Apache Spark-Based Analysis of Electricity Generation and Emissions

Systems and Architectures for Big Data, A.Y. 2024-2025

Alessandro Cortese
Computer Engineering
University of Rome
Tor Vergata
Roma, Roma
alessandro.cortese@students.uniroma2.eu

Chiara Iurato
Computer Engineering
University of Rome
Tor Vergata
Vittoria, Ragusa
chiara.iurato@students.uniroma2.eu

Luca Martorelli
Computer Engineering
University of Rome
Tor Vergata
Firenze, Firenze
luca.martorelli@students.uniroma2.eu

***Abstract*—This project presents an end-to-end pipeline for analyzing historical electricity production and direct carbon dioxide emissions data using the Apache Spark. It demonstrates the integration of big data tools to support environmental data analysis and performance benchmarking in a fully containerized architecture.**

I. Introduction

This project analyzes electricity production data and related CO_2 emissions using *Apache Spark* [1], focusing on carbon intensity and the share of low/zero-emission energy sources. The dataset, provided by *Electricity Maps* [2], contains hourly data from 2021 to 2024 for different countries, including key indicators such as carbon intensity and the percentage of carbon-free energy. The objectives of the project are:

- perform analytical queries using different Spark Abstraction (RDD, DataFrame, SQL);
- benchmark and compare their performance under varying runtime configurations.

II. System Architecture

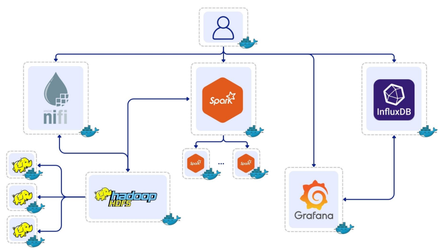


Fig. 1: System Architecture

The diagram above illustrates the system architecture, highlighting the interaction between the client and the core components of the infrastructure. The main elements are:

- **Client**: it initiates the entire workflow by interacting with *Apache Nifi* [3] to trigger data ingestion. Once the data is processed, it is stored in *HDFS* [4]. Subsequently, the client retrieves the data from HDFS and performs the required Spark queries. The query results are then written to *InfluxDB* [5] to enable seamless integration with *Grafana* [6] for visualizing the analytical outcomes.
- **Apache NiFi**: handles the ingestion of raw data from the local directory "/dataset", which contains multiple CSV files downloaded from the *Electricity Maps* source. NiFi merges all the files into a single dataset, removes columns that are not relevant for the upcoming queries, and stores the resulting data in a single Parquet file. The choice of the Parquet format is motivated by its efficiency in both storage and processing: the original CSV files occupy approximately 464 MB, whereas the resulting Parquet file is only 12.5 MB, significantly reducing storage overhead.
- **Apache Hadoop (HDFS)**: is a distributed storage system designed to run on commodity hardware. In this architecture, HDFS is used to store the dataset in different formats, including the merged Parquet file generated by NiFi. Its API allows for easy interaction, enabling both reading from and writing to the file system with minimal complexity. The HDFS setup consists of a master node and three slave nodes, which reflect the default configuration parameters typically used for small-scale deployments.
- **Apache Spark**: is the core processing engine of the project, and its configuration is dynamic at runtime. Through the execution script, it is possible to specify up to 8 worker nodes, allowing performance benchmarking and identification of the optimal configuration. Due to its simplicity and flexibility, Spark also performs an additional preprocessing phase.
- **InfluxDB**: is a powerful time-series database specifically designed to handle data indexed by time. Unlike systems like Redis, InfluxDB automatically appends a timestamp to each record upon insertion, which is essential to create dynamic and responsive time-series visualizations in Grafana. After Spark executes the queries, the final results—exported as CSV files—are written into InfluxDB.

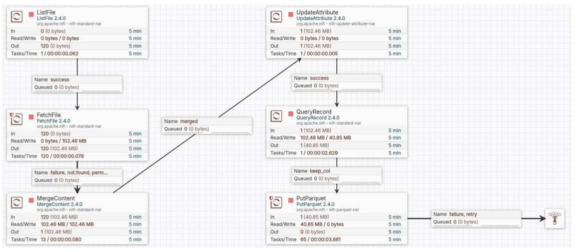
- **Grafana**: is a powerful open-source platform for querying, visualizing, and alerting on time-series data. In this system, it connects to InfluxDB through a dedicated plugin, enabling users to build interactive dashboards using the Flux query language. Grafana supports the import of dashboard configurations in JSON format, allowing automated chart generation and streamlined dashboard deployment.
  To enable the export of visualizations as images, a separate container was added to the infrastructure using the Grafana Image Renderer plugin.

III. Pipeline

A. Data Ingestion

The NiFi flow is programmatically started and stopped using the REST APIs provided by Apache NiFi. Its main function is to retrieve the multiple CSV files that constitute the dataset, merge them into a single unified file, convert the result into the more efficient **Parquet** format and apply filtering operations to ensure data quality. The final dataset is then written to **HDFS**, where it becomes available for subsequent processing and analysis.

For simplicity, the CSV files are preloaded into the NiFi container, upon flow execution. The scheme used is shown below:



```
ListFile
ListFile 2.4.0
org.apache.nifi - nifi-standard-nar
In          0 (0 bytes)                5 min
Read/Write  0 bytes / 0 bytes          5 min
Out         120 (0 bytes)              5 min
Tasks/Time  1 / 00:00:00.062           5 min

Name success
Queued 0 (0 bytes)

FetchFile
FetchFile 2.4.0
org.apache.nifi - nifi-standard-nar
In          120 (0 bytes)              5 min
Read/Write  0 bytes / 102.46 MB        5 min
Out         120 (102.46 MB)            5 min
Tasks/Time  120 / 00:00:00.078         5 min

Name failure, not.found, perm...
Queued 0 (0 bytes)

MergeContent
MergeContent 2.4.0
org.apache.nifi - nifi-standard-nar
In          120 (102.46 MB)            5 min
Read/Write  102.46 MB / 102.46 MB      5 min
Out         1 (102.46 MB)              5 min
Tasks/Time  13 / 00:00:00.080          5 min

Name merged
Queued 0 (0 bytes)

UpdateAttribute
UpdateAttribute 2.4.0
org.apache.nifi - nifi-update-attribute-nar
In          1 (102.46 MB)              5 min
Read/Write  0 bytes / 0 bytes          5 min
Out         1 (102.46 MB)              5 min
Tasks/Time  1 / 00:00:00.005           5 min

Name success
Queued 0 (0 bytes)

QueryRecord
QueryRecord 2.4.0
org.apache.nifi - nifi-standard-nar
In          1 (102.46 MB)              5 min
Read/Write  102.48 MB / 40.85 MB       5 min
Out         1 (40.85 MB)               5 min
Tasks/Time  1 / 00:00:02.629           5 min

Name keep_col
Queued 0 (0 bytes)

PutParquet
PutParquet 2.4.0
org.apache.nifi - nifi-parquet-nar
In          1 (40.85 MB)               5 min
Read/Write  40.85 MB / 0 bytes         5 min
Out         0 (0 bytes)                5 min
Tasks/Time  65 / 00:00:03.661          5 min

Name failure, retry
Queued 0 (0 bytes)
```

Fig. 2: NiFi Flow

The process begins with the `ListFile` processor, which is responsible for scanning the specified directory and listing all the files it contains. The FlowFile generated by `ListFile`, containing only the metadata of the located file (such as name, absolute path, size, timestamp), is then sent to the FetchFile processor. The latter takes care of physically accessing the file on disk, using the attributes provided by `ListFile` to locate its exact path. Once read, the contents of the file are loaded inside the FlowFile, replacing its previously empty body. The original attributes are retained, however, thus preserving information about the source file. At this point, the FlowFile is complete: it contains both the metadata useful for tracking and management, and the actual binary content of the file, ready to be processed in subsequent stages of the pipeline.

Next, the `MergeContent` processor comes into play, which receives as input a set of FlowFiles, of which it waits for all files to accumulate. Once the condition is met, it proceeds to queue or concatenate the contents of the individual FlowFiles into a single aggregated FlowFile.

Once merged, a file format change will be performed, making it a parquet file through the use of the `UpdateAttribute` processor, after which it is passed to the `QueryRecord` processor. Here, a schema-aware SQL query is applied to extract only the relevant columns for the analysis:

```
SELECT "Datetime (UTC)", "Country",
    "Carbon intensity gCO2eq/kWh (direct)",
    "Carbon-free energy percentage (CFE%)"
FROM FLOWFILE
```

This step ensures that all unnecessary fields are discarded at ingestion time, thereby reducing data volume. The filtered data is then written into **HDFS** using the `PutParquet` processor, making it immediately ready for Spark-based operations.

B. Pre-processing

Before the execution of analytical queries, another crucial pre-processing phase is carried out to ensure the quality and consistency of the dataset, supported by the Spark Controller. This phase includes the removal of rows with missing values and duplicates, this will guarantee a data volume reduction and will simplify downstream operations.

To enhance temporal analysis, the dataset is enriched with additional time-based attributes such as `Year`, `Month`, `Day`, and `Hour`, which are extracted from the timestamp associated with each observation. Only the fields relevant to the analytical goals are retained and uniformly renamed, resulting in a clean and compact dataset.

Once transformed, the dataset is stored back in **HDFS** using the format specified for the upcoming analysis.

Ingested data are typically written in **Parquet format** using the `PutParquet` processor, but if requested, the dataset can be converted to a different format (**AVRO** or **CSV**) for benchmarking purposes. This conversion takes place directly in Spark at runtime, rather than in NiFi, to simplify the preprocessing workflow.

This pre-processing pipeline sets the foundation for efficient, format-independent query execution. It is managed by the `SparkController`, which handles the full execution lifecycle for each query: loading the appropriate data format from HDFS, applying the selected Spark API (DataFrame, RDD, or SparkSQL), and collecting the results for further evaluation.

C. Processing: Queries

Before running the queries, a few practical decisions were made to ensure consistent and comparable execution across the different processing modes.

The dataset is always initially loaded as a DataFrame. From this base representation:

- RDD-based queries convert the DataFrame into an RDD to allow low-level operations;
- SQL-based queries register the DataFrame as a temporary view (`ElectricityData`);
- DataFrame-based queries directly use the loaded structure with high-level transformations.

Caching was deliberately avoided in all implementations. The dataset is relatively small and accessed only once per query, so caching would offer little benefit. More importantly, enabling caching could affect the execution time of subsequent runs, especially for benchmarking, by hiding data loading overhead and skewing comparisons between query types.

In Spark, transformations are evaluated according to the lazy evaluation model and are not executed until an action is applied. The `collect()` method is an action that forces the execution of transformations and returns the results to the driver. In the end, it is used to trigger the actual execution of Spark queries and retrieve the data in local memory.

Once the query is executed, the results are encapsulated in a *QueryResult* object, which ensures:

- consistent writing across multiple executions;
- uniform sorting and formatting of output;
- tracking of the average execution time;
- integration with logging, exporting, and visualization modules.

Query 1. *Referring to the dataset of energy values of Italy and Sweden, aggregate the data on an annual basis. Calculate the average, minimum and maximum of "Carbon intensity gCO_2eq/kWh (direct)" and "Carbon-free energy percentage (CFE%)" for each year from 2021 to 2024.*

RDD

For the execution of the first query, the following operation are performed:

- apply `filter` to select "Italy" and "Sweden";
- each record is mapped to a key-value pair where the key is `(Country, Year)`, and the value is a tuple containing:
  - sum of CI and CFE values;
  - element counts;
  - partial minima and maxima for CI and CFE;
- aggregation by key is applied: values are aggregated by `(Country, Year)` using `reduceByKey()`.
- final statistics are calculated, specifically for each aggregate data is calculated:
  - average of CI: sum CI / count;
  - average of CFE: sum CFE/ count;
  - Min/Max CI and CFE already calculated with aggregation;

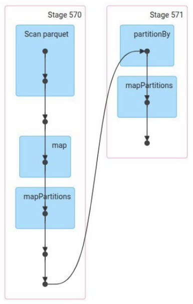


Fig. 3: Dag of Q1 with RDD

DataFrame

The following operations are performed:

- applies the same country filter as in the RDD version;
- for each pair `(Country, Year)` is calculated:
  - mean, minimum and maximum of carbon intensity;
  - mean, minimum and maximum of the percentage of CFE;
- data are sorted for ease of reading;

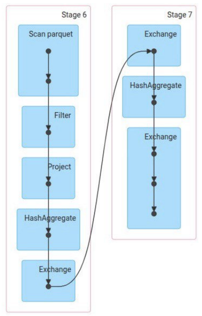


Fig. 4: Dag of Q1 with Dataframe

SQL

Among the three approaches, the first SQL implementation was the most straightforward to express, thanks to the natural mapping between the query requirements and standard SQL operations. Familiarity with SQL syntax also made it quick to develop:

- a `WHERE` clause to filter records for Italy and Sweden;
- a `GROUP BY` on `(Country, Year)` to aggregate data;
- aggregate functions `AVG`, `MIN`, and `MAX` applied to both carbon intensity and CFE percentage;
- an `ORDER BY` clause to ensure the results are sorted by country and year.

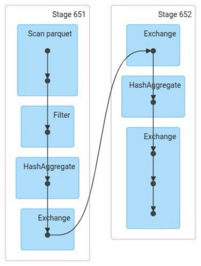


Fig. 5: Dag of Q1 with SQL

Query 2. *Considering only the Italian dataset, aggregate the data on the pair (year, month), calculating the average value of "Carbon intensity gCO_2eq/kWh (direct)" and "Carbon-free energy percentage (CFE%)." Calculate the ranking of the top 5 pairs (year, month) by sorting by "Carbon intensity gCO_2eq/kWh (direct)" decreasing, increasing and "Carbon-free energy percentage (CFE%)"decreasing, increasing.*

RDD
The implementation is structured as follows:

- apply `filter` to select “Italy”;
- each row is transformed into a key-value pair, where the key is `(Year, Month)` and the value is a tuple containing:
  - carbon-intensity;
  - carbon-free-energy;
  - a count equal to 1;
- the data is aggregated by key, summing the CI and CFE values and counting the number of occurrences per month;
- the averages are then computed by dividing the accumulated sums by the respective counts;
- the resulting dataset is reused to extract the top and bottom 5 records for both CI and CFE through four distinct sorting operations.

During development, several alternatives were tested:

- the use of `combineByKey` instead of `reduceByKey` for the grouping step, but no measurable performance gain was observed despite the greater flexibility;
- replacing the use of `sortBy(...).take()` with a single `collect()` followed by local Python `sorted()` operations on the driver. This change reduced the overhead of multiple distributed sorting steps and improved performance in practice, thanks to the relatively small result size.

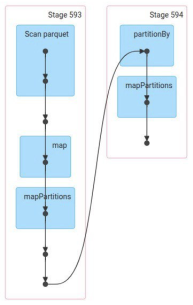


Fig. 6: Dag of Q2 with RDD

Dataframe
Similarly to RDD, the execution involves:

- filtering the dataset to include only records corresponding to `"Italy"`;
- applying a `groupBy` on the `Year` and `Month` columns;
- computing the average values for carbon intensity and carbon-free energy via the `avg()` function;
- sorting the result according to each metric in ascending and descending order to retrieve the top-5 and bottom-5 values;

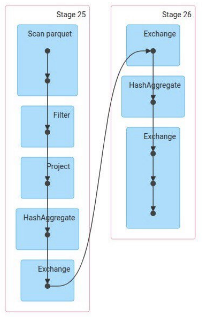


Fig. 7: Dag of Q2 with Dataframe

SQL
The execution flow is as follows:

- A common table expression (CTE) named `aggregated` performs the filtering for `"Italy"` and computes monthly averages for carbon intensity and CFE using `AVG()` and `GROUP BY`;
- Four additional CTEs are used to select the top-5 and bottom-5 values of each metric by ordering the results in ascending or descending order;
- These partial results are combined using `UNION ALL`, producing a final table with twenty rows: five for each ordering of the two metrics;

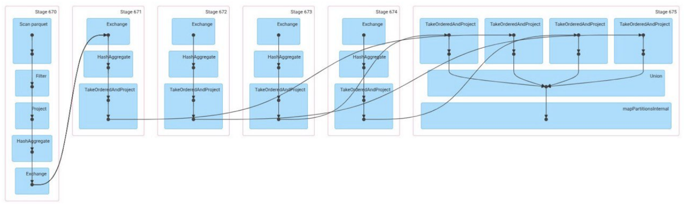


Fig. 8: Dag of Q2 with SQL

Query 3. *Aggregate the data over a 24-hour period, calculating the average value of carbon intensity and carbon-free energy (CFE) share. Compute the minimum, 25th percentile, 50th percentile (median), 75th percentile, and maximum*
RDD
The execution of the third query using RDDs involves the following steps:

- rows corresponding only to the countries “Italy” and “Sweden” are retained through a filtering operation;
- a mapping phase is then performed, where each record is converted into a key-value pair with key `(Country, Hour)` and value containing:
  - carbon intensity (CI) value;
  - carbon-free energy (CFE) value;
  - a count equal to 1 to track the number of elements;
- these key-value pairs are then combined using `reduceByKey()`, which produces, for each `(Country, Hour)`:
  - the total sum of CI and CFE values;
  - the total count of entries;

- the resulting aggregates are used to compute the average CI and CFE per hour, by dividing the accumulated sums by the corresponding counts;
- next, records are grouped by country in order to gather all hourly averages for each nation;
- from these grouped values, a statistical summary is produced by computing the empirical percentiles (0th, 25th, 50th, 75th, and 100th) for both CI and CFE;
- finally, the results are formatted as rows containing:
  - country code (“IT” or “SE”);
  - metric type (“carbon-intensity” or “cfe”);
  - the five percentile values for the selected metric.

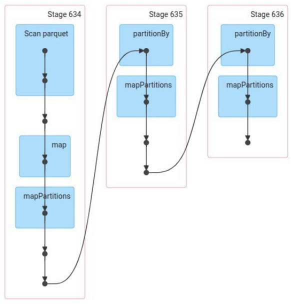


Fig. 9: Dag of Q3 with RDD

DataFrame
The query logic is composed of the following steps:

- rows are filtered to include only records from the countries “Italy” and “Sweden”;
- a grouping by `(Country, Hour)` is performed to compute hourly average values of:
  - carbon intensity;
  - carbon-free energy percentage (CFE);
- for each country and for both metrics (CI and CFE), the following percentiles are computed using `percentile_approx()`:
  - minimum (0th);
  - 25th, 50th, and 75th percentiles;
  - maximum (100th);

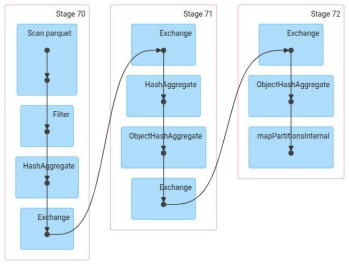


Fig. 10: Dag of Q3 with Dataframe

SQL
The following operations are performed:

- data is filtered using the `WHERE` clause to include only records from “Italy” and “Sweden”;
- hourly averages for carbon intensity and CFE percentage are calculated using `AVG()` grouped by `Country` and `Hour`;
- two intermediate queries are then used to compute the approximate percentiles (0%, 25%, 50%, 75%, 100%) for both metrics using `percentile_approx()`, grouped by country;
- ISO country codes (`IT`, `SE`) are assigned via a `CASE WHEN` expression for consistency with previous queries;
- final results for carbon intensity and CFE are merged using `UNION ALL` and formatted with alias columns for percentiles: `Min`, `P25`, `P50`, `P75`, `Max`;

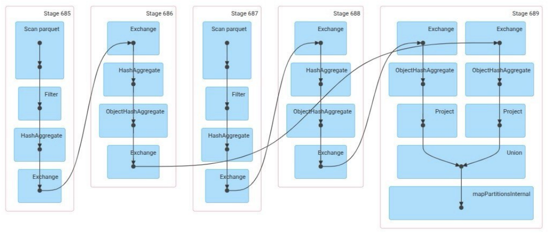


Fig. 11: Dag of Q3 with SQL

Query 4. *use the K-means clustering algorithm on a selected set of countries to group them according to carbon intensity calculated on a yearly basis, specifically for 2024. The countries used for clustering are those listed above.*

DataFrame
This query performs a clustering task using the KMeans algorithm, with the aim of grouping countries according to their average carbon intensity for the year 2024. The implementation relies on Spark DataFrame transformations and MLlib for clustering. The following steps are executed:

- the dataset is filtered to include only a predefined list of countries and only records from the year 2024;
- for each country, the average carbon intensity is computed using the `avg()` aggregation function;
- a feature vector is created by applying a `VectorAssembler` to the `Avg_Carbon_Intensity` column, resulting in a DataFrame suitable for MLlib processing;
- a model selection phase is performed where the optimal number of clusters k is determined by computing the *silhouette score* for values of k ranging from 2 to 14. For each k, a KMeans model is trained and evaluated;
- once the best k is identified, a final KMeans model is trained using that value. If the query is executed in parallel mode, the initialization method `k-means||` is used;
- clustering predictions are computed for each country and materialized using `collect()`, which also triggers execution;
- the final results are formatted as pairs of `(Country, Cluster)`.

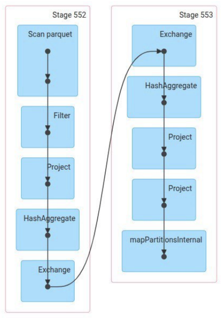


Fig. 12: Dag of Q4 with Dataframe

IV. EMPIRICAL RESULTS

The time of the query executions is measured using the Python `time` library, specifically by taking the time before the start of the first operation and after the invocation of a Spark action that can trigger the query execution, such as the `collect()` operation. Each query was executed multiple times to compute a reliable average runtime. The number of repetitions is defined by the `NUM_RUNS_PER_QUERY` parameter, set to 20 as a good trade-off between measurement stability and execution overhead. This choice also helps mitigate cold-start effects during the first iteration, such as JVM allocation, internal service initialization, and class loading. The number of worker nodes used varies from 2 to 8. The system architecture is turned off after collecting the metrics of interest and then allocated with an additional worker Spark node, until the maximum value of worker nodes indicated above is reached.

After each execution with a certain number of worker nodes, the Spark Session is shut down.

Each Spark node executes with the following characteristics:

- `SPARK_WORKER_MEMORY = 1G`;
- `SPARK_WORKERS_CORES = 1G`.

The analysis of the results is structured according to the following criteria:

- **Data format**: the format in which the dataset is stored and processed (e.g., CSV, Avro, Parquet);
- **Number of worker nodes**: the degree of parallelism used during Spark query execution;
- **Query execution type**: the Spark abstraction used—`RDD`, `DataFrame`, or `SQL`.

A. *Differences Between Formats (Avro vs. Parquet vs. CSV)*

We decided to compare the performance of different formats in order to compare the performance of queries performed with text and binary formats and with different access modes, particularly per row or per column.

Based on the considered queries, it can be seen that the worst storage format is CSV. This is due precisely to the characteristics of the format; in fact, fast reading is not guaranteed since no schema is supported, is used row-based access and the storage format used is textual.

The best storage format is Parquet. This is because the format uses a columnar access mode so that it reads only the columns of interest while avoiding reading the entire data set to perform fewer I/O operations, uses more efficient compression, supports predicate pushdown and is used a binary storage format. All of these features make reading the data of interest much faster and thus allow for better performance.

The use of Avro as a storage format performs better than CSV, but worse than Parquet. This is due to the fact that, in spite of the use of a binary storage format, access remains row-based and thus more operations are paid for than with Parquet. Compared to CSV, with Avro, the advantages of using schema, native data types and the binary format allow for better performance.

B. *Effect of the Number of Worker Nodes*

In the performance collection phase, was noted divergent behaviour as the number of workers used to execute the queries varied, regardless of the abstraction used.

For CSV and Avro, is noted a positive horizontal scalability, in fact as the number of workers used increases the average execution time decreases.

With Parquet, it is noticeable that is not obtained the same result, in fact after a certain number of worker nodes the average execution time tends to increase.

This behaviour is due to a combination of factors related to the nature of the formats and Spark's execution model. Parquet is an optimised columnar format and even in environments with few worker nodes it is able to provide very good performance. When the number of workers increases, a communication and task scheduling overhead is introduced, but since much less work is required with Parquet, the overhead can outweigh the benefits introduced with horizontal scalability and thus a deterioration in performance, this is because few logical blocks can be created resulting in limited parallelism.

The use of the CSV and Avro formats is more advantageous as it requires more parsing by having to access the data per row and thus each worker is able to process a number of rows independently, resulting in greater scalability.

In summary, Parquet remains the most efficient storage format, but the drop in performance above a certain threshold of parallelism is due to excessive overhead relative to task workload.

C. *Comparison Between RDD, SparkSQL and DataFrame*

From the benchmark results, DataFrame proved to be the most performant abstraction, with or without SQL.

This behaviour is mainly due to the use of Spark's Catalyst Optimizer, which restructures the application's logical plans to make them more efficient and translates queries into physical plans optimised for actual execution. RDDs, on the other hand, are a lower-level abstraction and are not subject to Catalyst, so the written code executes exactly without going through a transformation or optimisation phase.

Although RDDs can offer more control and flexibility, they have shown a higher cost in terms of performance, whereas Dataframes have shown greater efficiency due to their declarative nature and the optimisation mechanisms provided by Spark.

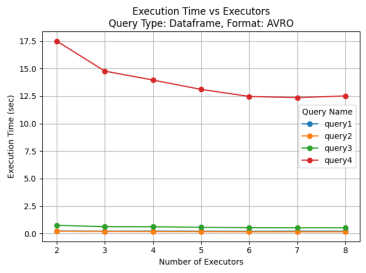


Fig. 13: Execution Time for Dataframe with Avro

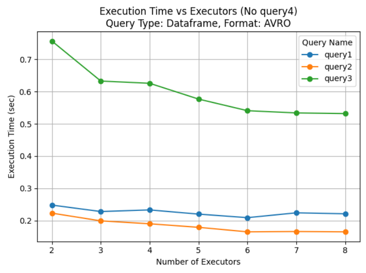


Fig. 14: Execution Time for Dataframe with Avro, without query 4

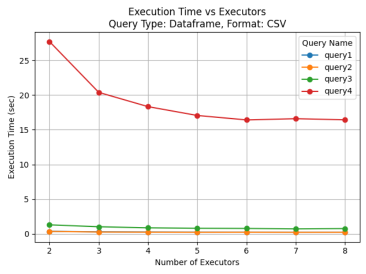


Fig. 15: Execution Time for Dataframe with CSV

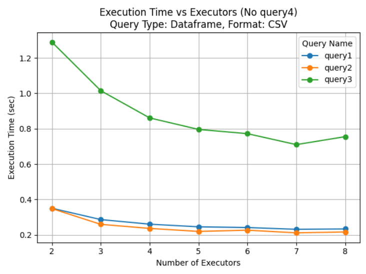


Fig. 16: Execution Time for Dataframe with CSV, without query 4

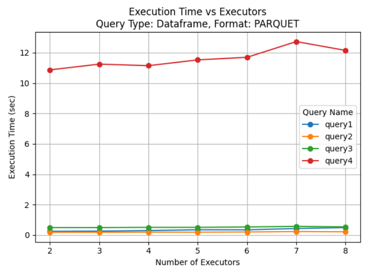


Fig. 17: Execution Time for Dataframe with Parquet

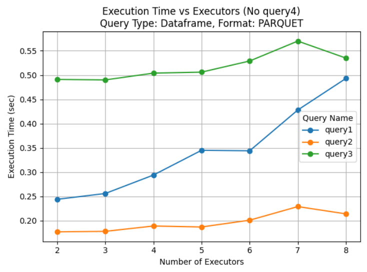


Fig. 18: Execution Time for Dataframe with Parquet, without query 4

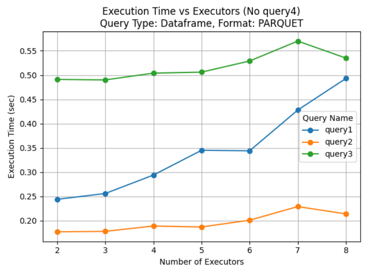


Fig. 19: Execution Time for Dataframe with Parquet, without query 4

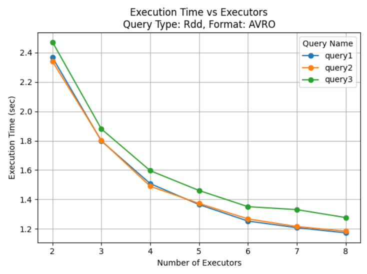


Fig. 20: Execution Time for RDD with Avro

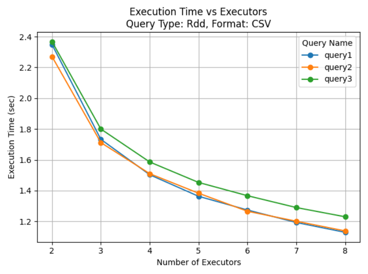


Fig. 21: Execution Time for RDD with CSV

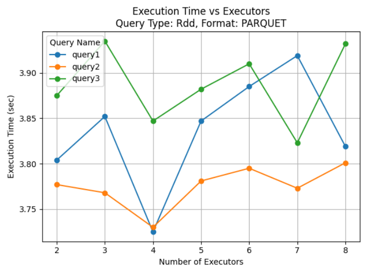


Fig. 22: Execution Time for RDD with Avro

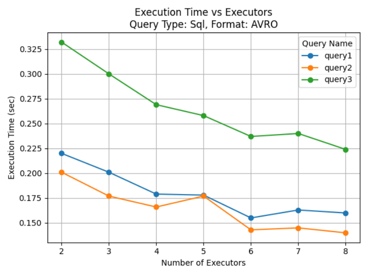


Fig. 23: Execution Time for SQL with Avro

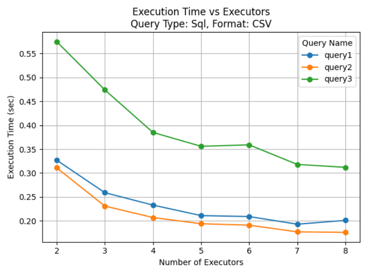


Fig. 24: Execution Time for SQL with CSV

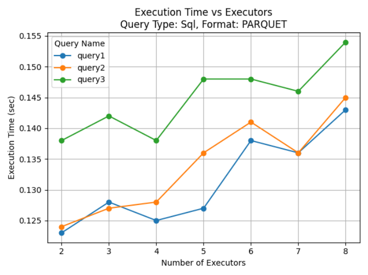


Fig. 25: Execution Time for SQL with Parquet

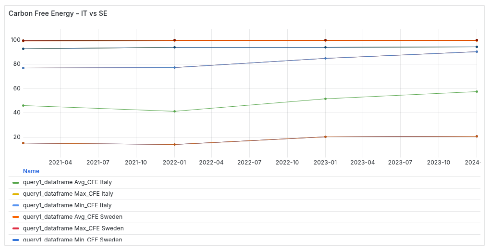


Fig. 26: Results of Q1 in Grafana

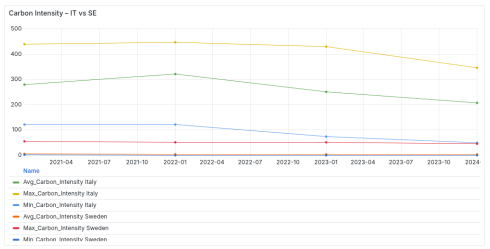


Fig. 27: Results of Q1 in Grafana

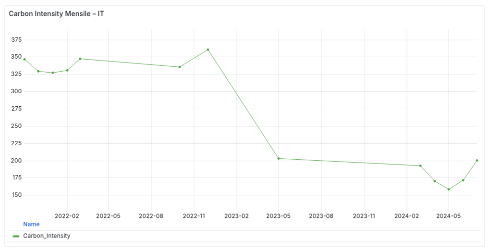


Fig. 28: Results of Q2 in Grafana

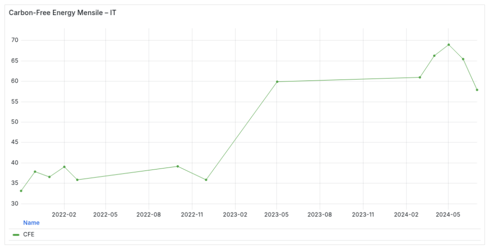


Fig. 29: Results of Q2 in Grafana

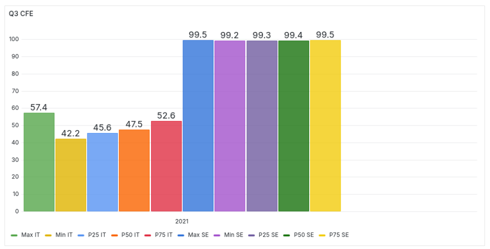


Fig. 30: Results of Q3 in Grafana

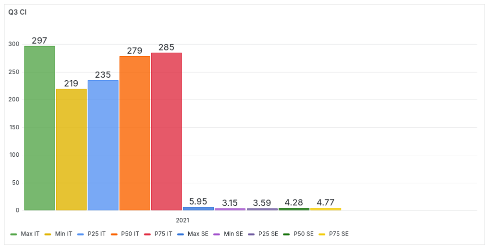


Fig. 31: Results of Q3 in Grafana

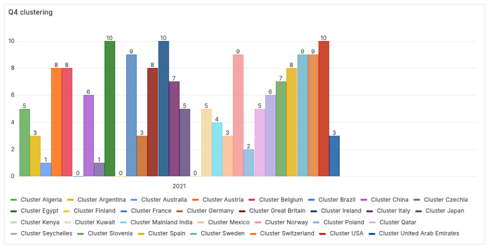


Fig. 32: Results of Q4 in Grafana

References

[1] Apache Software Foundation.
Apache Spark: Lightning-fast unified analytics engine.
URL: https://spark.apache.org/.

[2] *Electricity Maps*.
URL: https://www.electricitymaps.com/.

[3] Apache Software Foundation.
Apache NiFi.
URL: https://nifi.apache.org/.

[4] Apache Software Foundation.
Hadoop Distributed File System (HDFS).
URL: https://hadoop.apache.org/.

[5] InfluxData.
InfluxDB: Time series platform.
URL: https://www.influxdata.com/.

[6] Grafana Labs. *Grafana: The open observability platform*.
URL: https://grafana.com/.