Analisi di dati energetici con Apache Spark

Leonardo Pompili - 0353499
leonardo.pompili@students.uniroma2.eu
Ingegneria Informatica
Università degli Studi di Roma "Tor Vergata"
Roma, Italia

Silvia Perelli - 0350110
silvia.perelli@students.uniroma2.eu
Ingegneria Informatica
Università degli Studi di Roma "Tor Vergata"
Roma, Italia

Sara Malaspina - 0350111
sara.malaspina@students.uniroma2.eu
Ingegneria Informatica
Università degli Studi di Roma "Tor Vergata"
Roma, Italia

***Abstract*—La presente relazione descrive l'architettura e l'implementazione di un sistema distribuito per l'analisi di dati energetici su larga scala. Il sistema sfrutta Apache Spark per l'elaborazione di query su dati storici forniti da Electricity Maps, relativi alla produzione di elettricità e alle emissioni di CO_2. L'architettura comprende un data lake basato su HDFS, una pipeline di data ingestion con Apache NiFi, l'elaborazione delle query con Spark, l'esportazione dei risultati su Redis e la visualizzazione tramite Grafana. L'intero sistema è containerizzato utilizzando Docker Compose per garantire portabilità e riproducibilità.**

***Index Terms*—Apache Spark, Big Data, HDFS, Docker, Apache NiFi, Redis, Grafana, DataFrame, RDD, Spark SQL, K-Means Clustering.**

I. Introduzione

L'analisi di grandi moli di dati (Big Data) è diventata cruciale in numerosi settori, inclusa l'energia, dove la comprensione dei pattern di consumo, produzione ed emissioni è fondamentale per la sostenibilità e l'efficienza. Il progetto proposto mira a utilizzare Apache Spark [1], un potente framework open-source per l'elaborazione distribuita, al fine di analizzare dati storici sull'elettricità e sulle emissioni di CO_2 forniti da Electricity Maps [2]. L'obiettivo è rispondere a specifiche query analitiche, aggregando e confrontando dati energetici di diverse nazioni (Italia e Svezia), conducendo anche un'analisi di clustering per identificare nazioni con comportamenti emissivi simili. Questa relazione dettaglia l'architettura del sistema implementato, le tecnologie utilizzate, il processo di data ingestion e le modalità di presentazione dei risultati. L'intera infrastruttura è stata containerizzata mediante Docker Compose [3], garantendo un ambiente di sviluppo e deployment isolato e facilmente replicabile. Vengono infine discussi i risultati ottenuti per le query, inclusa una valutazione delle prestazioni tra Spark API DataFrame, RDD e Spark SQL.

II. Dataset e Requisiti

Il dataset di input, fornito da Electricity Maps, contiene dati storici orari sulla produzione di elettricità e le relative emissioni di CO_2 per il periodo dal 1 gennaio 2021 al 31 dicembre 2024. Per ogni paese, il dataset originale consiste in circa 35.000 eventi. Ai fini del progetto, sono stati considerati i dati relativi all'Italia, alla Svezia e, per l'analisi di clustering, ad un insieme esteso di 30 nazioni (15 paesi europei e 15 extra-europei). Le metriche di interesse primario sono:

- **Intensità di carbonio diretta (gCO_2eq/kWh)**: quantità di gas serra emessi per unità di elettricità.
- **Percentuale di energia senza emissioni di carbonio (CFE%)**: percentuale di elettricità da fonti a basse o nulle emissioni di CO_2.

III. Architettura del Sistema

L'architettura proposta è stata progettata per gestire l'intero ciclo di vita dei dati, dalla loro acquisizione fino alla visualizzazione dei risultati analitici. Essa si basa su un insieme di componenti open-source, orchestrati tramite Docker Compose. Uno schema concettuale dell'architettura è mostrato in Fig. 1.

I componenti principali sono:

- **Apache NiFi**: Impiegato per la fase di data ingestion automatizzata.
- **Hadoop Distributed File System (HDFS)**: Utilizzato come data lake per lo storage persistente dei dati grezzi, processati e convertiti in formato Parquet. È configurato con un NameNode e un DataNode.
- **Apache Spark**: Framework centrale per l'elaborazione distribuita dei dati. È configurato in modalità cluster standalone con un Master e due Worker.
- **Redis**: Sistema di data storage in-memory utilizzato per memorizzare i risultati aggregati delle query Spark.
- **Grafana**: Piattaforma open-source per la visualizzazione e l'analisi dei dati, utilizzata per creare dashboard interattive basate sui risultati memorizzati in Redis.
- **Docker Compose**: Ha consentito di definire e orchestrare l'ambiente multi-container, includendo NiFi, HDFS, Spark, Redis e Grafana.

Tutti i servizi sono interconnessi tramite una rete Docker (`sabd_net`), garantendo la comunicazione isolata tra i container.

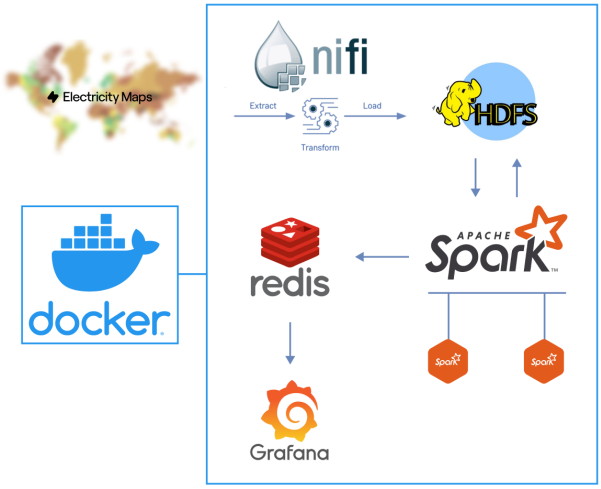

Fig. 1. Schema dell'architettura del sistema distribuito.

A. Data Ingestion e Preprocessing con Apache NiFi e Spark

La fase di data ingestion è cruciale per popolare il data lake HDFS [4] con i dati grezzi necessari per le analisi. A tale scopo, si è realizzato un flusso Apache NiFi [5] che si pone l'obiettivo di automatizzare il processo di recupero, trasformazione e archiviazione dei dati da una fonte esterna (electricitymaps.com) su un cluster HDFS. Nello specifico, il flusso orchestra dinamicamente il download di molteplici file CSV, li converte in formato Parquet e li organizza in HDFS in base al paese e all'anno di riferimento. Di seguito sono elencati i connettori che compongono la pipeline NiFi (mostrata in Fig. 2) per la fase di data ingestion:

- **GenerateFlowFile**: Questo processore funge da trigger iniziale per il flusso e genera un FlowFile a intervalli regolari.
- **ExecuteScript**: Riceve il FlowFile trigger da GenerateFlowFile ed esegue uno script per generare dinamicamente una serie di FlowFile, ognuno rappresentante una richiesta di download specifica. Lo script definisce un URL base per costruire gli URL di download, un suffisso `hourly.csv`, una lista di `countries` (`IT`, `SE`, ecc.) e una lista di `year` (2021, 2022, 2023, 2024). Il codice cicla poi su ogni combinazione di `countries` e `year` per costruire l'URL di download completo creando un nuovo FlowFile per ogni URL.
- **InvokeHTTP**: Per ogni FlowFile ricevuto da ExecuteScript, questo processore effettua una richiesta `HTTP GET` all'URL specificato nell'attributo `download.url` per scaricare il file CSV corrispondente. Il contenuto del FlowFile viene sostituito dalla risposta HTTP (cioè, i dati CSV scaricati).
- **ConvertRecord**: Converte i dati dal formato CSV (letto tramite un `CSVReader`) al formato Parquet (scritto tramite un `ParquetRecordSetWriter`). Il formato dati Parquet risulta particolarmente adatto per l'ottimizzazione dello storage e delle query su HDFS.
- **UpdateAttribute**: Questo processore modifica l'attributo filename del FlowFile per riflettere il nuovo formato Parquet.
- **PutHDFS**: Scrive il contenuto del FlowFile (ora in formato Parquet) nel sistema HDFS. Il percorso di destinazione in HDFS è costruito dinamicamente usando gli attributi `data.country` e `data.year` impostati da ExecuteScript. Ad esempio, i dati per l'Italia 2024 andranno in `/nifi_data/electricity_maps/IT/2024/`.

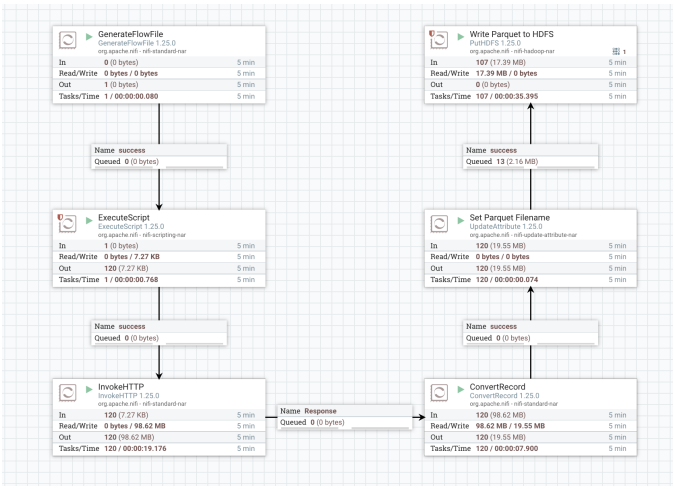

Fig. 2. Pipeline NiFi.

Successivamente, uno script PySpark (`preprocess_data.py`) viene eseguito per unificare e processare ulteriormente i dati Parquet caricati da NiFi. Questo script:

1) Legge i dati in formato Parquet da HDFS relativi a tutti i paesi e anni scaricati.
2) Effettua operazioni di pulizia rimuovendo le colonne non necessarie per le analisi richieste.
3) Converte i campi rilevanti nei tipi di dato appropriati.
4) Estrae componenti temporali come anno, mese, giorno e ora dalla colonna `datetime`.
5) Salva il DataFrame processato nuovamente su HDFS, partizionato per `country` (codice paese), sempre in formato Parquet. Questo dataset pre-processato costituisce la base per tutte le query successive.

Questo approccio a due stadi (NiFi per l'acquisizione e la conversione iniziale, Spark per il raffinamento) garantisce flessibilità e efficienza.

B. Elaborazione delle Query con Apache Spark

Le quattro query richieste sono state implementate in script PySpark separati, i quali utilizzano le API DataFrame di Spark, Spark SQL e RDD. Tutti gli script leggono dati in formato Parquet partizionati da HDFS tramite `spark_session.read.parquet()` e salvano i riultati in CSV sempre su HDFS. L'uso di `coalesce(1)` ha consentito di ottenere un singolo file per query.

C. Esportazione dei Risultati su Redis

I risultati in formato CSV prodotti da ciascuna query e salvati su HDFS vengono successivamente letti da un apposito script PySpark (`export_to_redis.py`) e caricati su un'istanza Redis [6]. Per ogni query, è stata definita una strategia di naming delle chiavi per facilitare il recupero dei dati. Il valore associato alla chiave è un oggetto JSON contenente tutte le metriche aggregate.

D. Visualizzazione con Grafana

Grafana [7] è stato utilizzato per creare dashboard interattive. Per prima cosa, è stato configurato un data source Redis al fine di leggere i dati esportati, poi, per ogni query, è stato creato un pannello grafico per le metriche d'interesse. La configurazione dei pannelli in Grafana ha richiesto l'uso delle query Redis tramite Command Line Interface, specificando il comando `MGET` seguito dalle chiavi contenenti i dati da recuperare. Sono state, inoltre, utilizzate delle trasformazioni interne a Grafana per plasmare i dati nel formato richiesto dalle visualizzazioni, quali `Extract fields`, `Join by field`, `Organize fields by name` e `Convert fields type`.

IV. Query

Di seguito sono riportate le quattro richieste di analisi da effettuare sui dati raccolti:

- **Query 1**: Facendo riferimento al dataset dei valori energetici dell'Italia e della Svezia, aggregare i dati su base annua. Calcolare la media, il minimo ed il massimo di "Carbon intensity gCO2eq/kWh (direct)" e "Carbon-free energy percentage (CFE%)" per ciascun anno dal 2021 al 2024.
- **Query 2**: Considerando il solo dataset italiano, aggregare i dati sulla coppia (anno, mese), calcolando il valor medio di "Carbon intensity gCO2eq/kWh (direct)" e "Carbon-free energy percentage (CFE%)". Calcolare la classifica delle prime 5 coppie (anno, mese) ordinando per "Carbon intensity gCO2eq/kWh (direct)" decrescente, crescente e "Carbon-free energy percentage (CFE%)" decrescente, crescente. In totale sono attesi 20 valori.
- **Query 3**: Facendo riferimento al dataset dei valori energetici dell'Italia e della Svezia, aggregare i dati di ciascun paese sulle 24 ore della giornata, calcolando il valor medio di "Carbon intensity gCO2eq/kWh (direct)" e "Carbon-free energy percentage (CFE%)". Calcolare il minimo, 25-esimo, 50-esimo, 75-esimo percentile e massimo del valor medio di "Carbon intensity gCO2eq/kWh (direct)" e "Carbon-free energy percentage (CFE%)".
- **Query 4**: eseguire un'analisi di clustering sui dati relativi al "Carbon intensity gCO2eq/kWh (direct)", aggregati su base annua e per l'anno 2024. I dati fanno riferimento ai valori medi annui per ciascun paese. L'obiettivo è individuare insiemi di nazioni con comportamenti simili in termini di emissioni di carbonio, usando l'algoritmo di clustering K-Means. Oltre ad applicare l'algoritmo di clustering sui dati, determinare un valore ottimale di k (numero di cluster) mediante l'uso di metriche appropriate, come ad esempio elbow method o l'indice di silhouette. Si considerino i seguenti 15 paesi europei: Austria, Belgio, Francia, Finlandia, Germania, Gran Bretagna, Irlanda, Italia, Norvegia, Polonia, Repubblica Ceca, Slovenia, Spagna, Svezia e Svizzera. Si scelgano inoltre 15 paesi extra-europei, tra cui Stati Uniti, Emirati Arabi, Cina, India, per un totale di 30 paesi a livello mondiale.

Successivamente viene discussa più nel dettaglio l'implementazione di ciascuna delle quattro query richieste. Ad eccezione dell'analisi di clustering, oltre all'implementazione tramite API DataFrame e RDD, le query 1, 2 e 3 sono state svolte anche con l'utilizzo di Spark SQL, al fine di confrontare le prestazioni ottenute.

A. Query 1

Negli *Algoritmi 1* e *2* sono riportate le implementazioni in pseudocodice della query 1 tramite API DataFrame e RDD.

Algoritmo 1 Query 1 - DataFrame

Input: *spark_session*, *paths_to_read*
1: Lettura dei file Parquet su HDFS (`spark_session.read.parquet`)
2: Aggregazione per anno e paese (`groupBy`):
3: Calcolo media, minimo e massimo (`F.avg, F.min, F.max`) per *carbon_intensity* e *carbon_free_percentage*

Algoritmo 2 Query 1 - RDD

Input: *spark_session*, *paths_to_read*
1: Lettura dei file Parquet su HDFS (`spark_session.read.parquet`)
2: Conversione del DataFrame letto in RDD (`.rdd`)
3: Mapping (`map`) tra chiave (*year, country_code*) e valori aggregati (*carbon_intensity, carbon_free_percentage, count*)
4: Calcolo di somma, minimo, massimo e conteggio (`reduceByKey`)
5: Calcolo delle medie (`map`)

Nelle Fig. 3 e 4 sono mostrati i grafici relativi ai risultati ottenuti con la query 1.

B. Query 2

Negli *Algoritmi 3* e *4* sono riportate le implementazioni in pseudocodice della query 2 tramite API DataFrame e RDD.

Nelle Fig. 5 e 6 sono mostrati i grafici relativi ai risultati ottenuti con la query 2.

C. Query 3

Negli *Algoritmi 5* e *6* sono riportate le implementazioni in pseudocodice della query 3 tramite API DataFrame e RDD.

Il DataFrame e l'RDD, aggregati per le 24 fasce orarie, sono stati memorizzati in cache al fine di eseguire in modo più efficiente le operazioni richieste (minimo, massimo e percentili dei valori medi). I percentili sono stati calcolati tramite delle

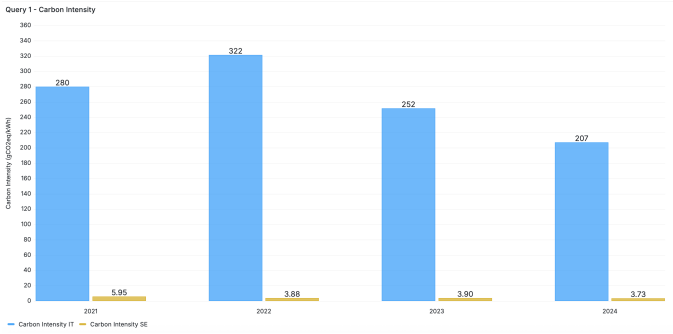

Fig. 3. Grafico Bar Chart per query 1 che mostra l'andamento del valor medio per la metrica "Carbon intensity gCO2eq/kWh (direct)" su base annua per l'Italia (in blu) e la Svezia (in giallo).

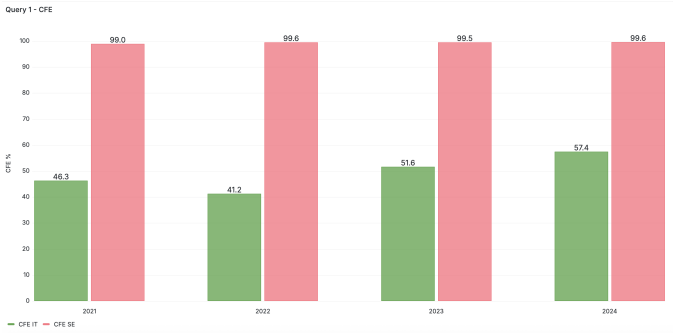

Fig. 4. Grafico Bar Chart per query 1 che mostra l'andamento del valor medio per la metrica "Carbon-free energy percentage (CFE%)" su base annua per l'Italia (in verde) e la Svezia (in rosa).

funzioni ausiliarie raggruppando i dati delle rispettive strutture per `country_code`. Nel caso dei DataFrame, le statistiche sono state ottenute utilizzando la funzione `F.expr()` di PySpark SQL, mentre per gli RDD è stata implementata un'interpolazione lineare.

Nelle Fig. 7 e 8 sono mostrati i grafici relativi ai risultati ottenuti con la query 3.

D. Query 4

Nell' *Algoritmo 7* viene mostrata l'implementazione in pseudocodice della query 4 tramite API DataFrame e algo-

Algoritmo 3 Query 2 - DataFrame

Input: *spark_session*, *path_to_read*
1: Lettura dei file Parquet su HDFS (`spark_session.read.parquet`)
2: Aggregazione per anno e mese (`groupBy`):
3: Calcolo media (`F.avg`) per *carbon_intensity* e *carbon_free_percentage*
4: Memorizzazione in cache del DataFrame aggregato
5: Estrazione classifiche (`orderBy` + `limit`):
6: 5 migliori e peggiori coppie (anno, mese) per *carbon_intensity* e *carbon_free_percentage*
7: Unione dei risultati in un unico DataFrame (`unionAll`)
8: Rilascio della cache

Algoritmo 4 Query 2 - RDD

Input: *spark_session*, *path_to_read*
1: Lettura dei file Parquet su HDFS (`spark_session.read.parquet`)
2: Conversione del DataFrame letto in RDD (`.rdd`)
3: Mapping (`map`) tra chiave (*year, month*) e valori aggregati (*carbon_intensity, carbon_free_percentage, count*)
4: Calcolo di somma e conteggio (`reduceByKey`)
5: Calcolo delle medie (`map`)
6: Memorizzazione in cache dell'RDD aggregato
7: Estrazione classifiche (`sortBy` + `take`):
8: 5 migliori e peggiori coppie (anno, mese) per *carbon_intensity* e *carbon_free_percentage*
9: Unione dei risultati in un unico RDD
10: Rilascio della cache

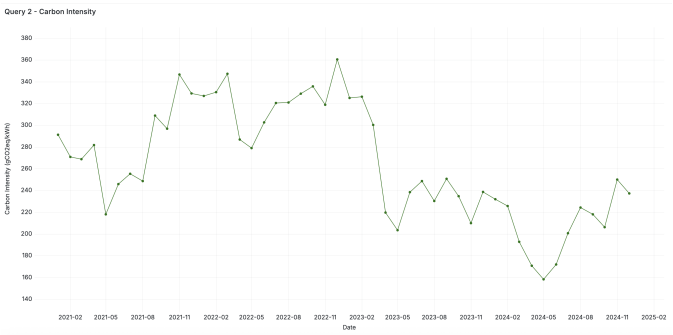

Fig. 5. Grafico Time Series per query 2 che mostra l'andamento del valor medio per la metrica "Carbon intensity gCO2eq/kWh (direct)" sulla coppia (anno, mese) per l'Italia.

ritmo K-Means. Oltre ai 15 paesi europei richiesti, sono stati considerati i seguenti 15 paesi extra-europei: USA, Argentina, Canada, India, Corea del Sud, Brasile, Australia, Sud Africa, Cina, Messico, Marocco, Thailandia, Emirati Arabi, Senegal e Singapore. Per determinare il valore ottimale del numero di cluster k, è stata svolta preliminarmente una fase di tuning, sia tramite *Elbow Method* che *Silhouette Score*, in modo da poter confrontare i risultati ottenuti. L'intervallo di valori testati per

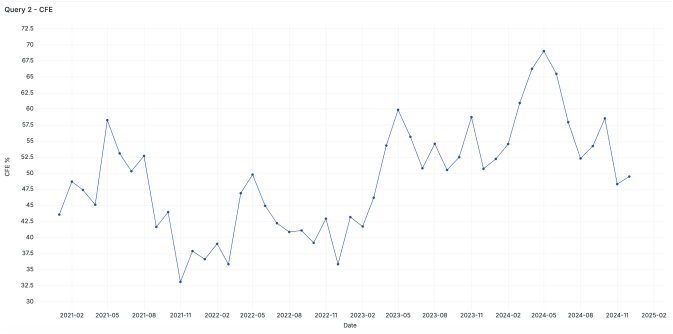

Fig. 6. Grafico Time Series per query 2 che mostra l'andamento del valor medio per la metrica "Carbon-free energy percentage (CFE%)" sulla coppia (anno, mese) per l'Italia.

Algoritmo 5 Query 3 - DataFrame

Input: *spark_session*, *paths_to_read*

1: Lettura dei file Parquet su HDFS (`spark_session.read.parquet`)
2: Aggregazione per paese e ora del giorno (`groupBy`):
3: Calcolo media (`F.avg`) per *carbon_intensity* e *carbon_free_percentage*
4: Memorizzazione in cache del DataFrame aggregato
5: Calcolo statistiche per paese:
6: Minimo, percentili (25°, 50°, 75°), massimo su *carbon_intensity* e *carbon_free_percentage*
7: Unione dei risultati in un unico DataFrame (`unionByName`)
8: Rilascio della cache

Algoritmo 6 Query 3 - RDD

Input: *spark_session*, *paths_to_read*

1: Lettura dei file Parquet su HDFS (`spark_session.read.parquet`)
2: Conversione del DataFrame letto in RDD (`.rdd`)
3: Mapping (`map`) tra chiave (*hour, country_code*) e valori aggregati (*carbon_intensity, carbon_free_percentage, count*)
4: Calcolo di somma e conteggio (`reduceByKey`)
5: Calcolo delle medie (`map`)
6: Memorizzazione in cache del DataFrame aggregato
7: Mapping (`map`) tra chiave (*country_code*) e valori aggregati (*avg_ci, avg_cfe*)
8: Raggruppamento per paese (`groupByKey`)
9: Calcolo statistiche per paese (`flatMap`):
10: Minimo, percentili (25°, 50°, 75°), massimo su *carbon_intensity* e *carbon_free_percentage*
11: Rilascio della cache

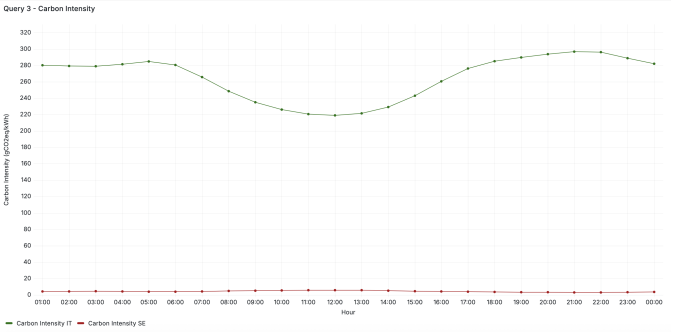

Fig. 7. Grafico Time Series per query 3 che mostra l'andamento del valor medio per la metrica "Carbon intensity gCO2eq/kWh (direct)" sulle 24 fasce orarie giornaliere per l'Italia (in verde) e la Svezia (in giallo).

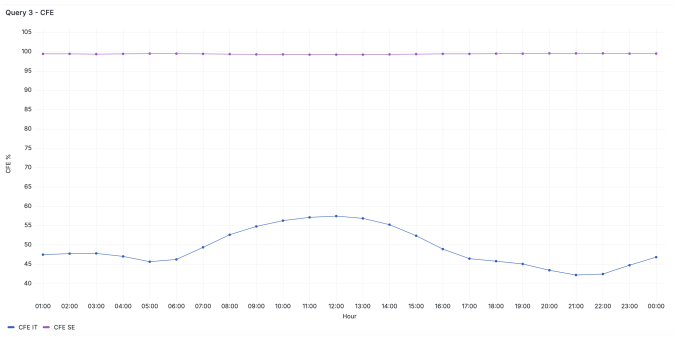

Fig. 8. Grafico Time Series per query 3 che mostra l'andamento del valor medio per la metrica "Carbon-free energy percentage (CFE%)" sulle 24 fasce orarie giornaliere per l'Italia (in blu) e la Svezia (in viola).

k va da 2 fino al minimo tra 15 e il numero di campioni.

*1) **Elbow method**:* Per stimare il numero ottimale di cluster k, si calcola la Within-Cluster Sum of Squares (WCSS) per diversi valori di k. Per ciascun k, si addestra un modello K-Means e si ottiene la WCSS tramite `summary.trainingCost` di Spark MLlib. Il valore ottimale di k è individuato tramite un approccio geometrico: si traccia il segmento tra il primo e l'ultimo punto del grafico WCSS vs k (mostrato in Fig. 9), e si seleziona il punto con la massima distanza perpendicolare da questo segmento, corrispondente al "gomito" della curva.

Dallo studio emerge che il valore ottimale del numero di cluster risulta essere pari a 5.

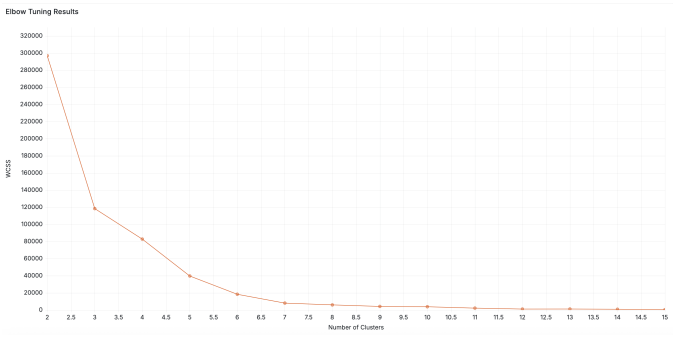

Fig. 9. Grafico WCSS vs k che mostra i risultati ottenuti dal tuning *Elbow Method*.

*2) **Silhouette Score**:* Per ciascun k, si addestra un modello K-Means, si generano le predizioni e si valuta la qualità del clustering utilizzando la metrica Silhouette Score con distanza euclidea al quadrato. Il Silhouette Score misura la separabilità e la coesione dei cluster: valori prossimi a +1 indicano una buona struttura, mentre valori negativi segnalano assegnazioni errate. Il valore ottimale di k corrisponde a quello che massimizza il Silhouette Score tra quelli validamente calcolati.

In Fig. 10 è mostrato l'andamento del Silhouette Score al variare del numero di cluster. Dallo studio emerge che il valore ottimale risulta essere pari a 11 cluster.

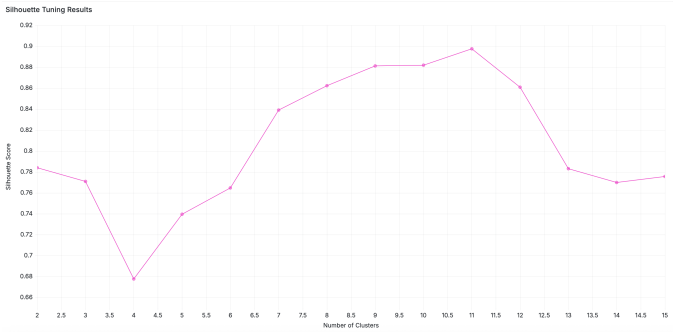

Fig. 10. Grafico che mostra l'andamento del *Silhouette Score* al variare del numero di cluster.

Algoritmo 7 Query 4 - Clustering

Input: *spark_session*, *paths_to_read*, *k*

1: Lettura dei file Parquet su HDFS (`spark_session.read.parquet`)
2: Filtraggio per l'anno target 2024 (`where`)
3: Aggregazione per paese (`groupBy`):
4: Calcolo della media annua (`F.avg`) di *carbon_intensity*
5: Preparazione feature vettoriale per clustering (`VectorAssembler`)
6: Addestramento modello K-Means con *k* cluster
7: Assegnazione dei cluster ai paesi

Nelle Fig. 11 e 12 sono mostrati i grafici relativi ai risultati ottenuti con la query 4 con k=5.

Fig. 11. Grafico GeoMap che mostra i cluster ottenuti con k=5.

Nelle Fig. 13 e 14 sono mostrati i grafici relativi ai risultati ottenuti con la query 4 con k=11.

V. Analisi delle Prestazioni

È stata condotta una valutazione sperimentale dei tempi di processamento delle query per analizzare l'impatto del parallelismo e confrontare l'efficienza delle diverse API offerte da Apache Spark: l'API DataFrame, RDD e Spark SQL.

A. Metodologia di Misurazione

Per ogni query e per ciascuna implementazione è stata eseguita una serie di test. La metodologia adottata per la misurazione delle prestazioni è la seguente:

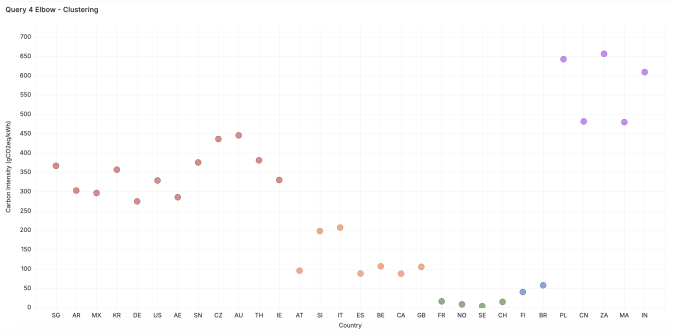

Fig. 12. Grafico XY Chart che mostra i cluster ottenuti con k=5.

Fig. 13. Grafico GeoMap che mostra i cluster ottenuti con k=11.

1) **Configurazione di SparkSession:** Per ogni esecuzione, è stata creata una nuova SparkSession. Il cluster Spark è stato configurato con due worker node, ciascuno con 2 core a disposizione, per un totale di 4 core disponibili nel cluster. Ogni executor Spark è stato configurato per utilizzare 1 core (`spark.executor.cores="1"`). Il parametro `spark.cores.max` è stato fatto variare da 1 a 4, controllando così il numero totale di executor (e quindi di core attivi) utilizzati dall'applicazione. La memoria assegnata a ciascun executor è stata mantenuta costante e pari ad 1 GB (`spark.executor.memory="1g"`), poichè ogni worker node è stato configurato per avere a disposizione 2 GB ciascuno.

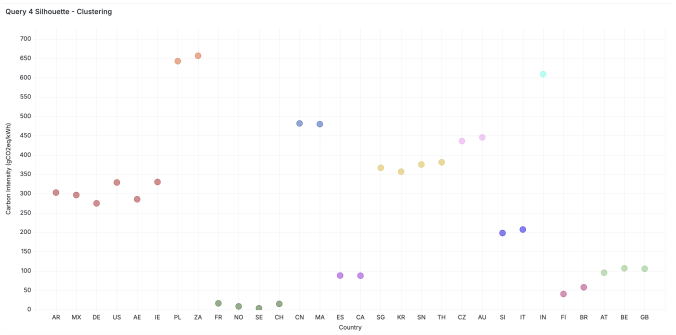

Fig. 14. Grafico XY Chart che mostra i cluster ottenuti con k=11.

2) **Esecuzioni multiple:** Ciascuna combinazione query-API-configurazione è stata eseguita 10 volte, includendo una prima esecuzione di warm-up. Questa ripetizione permette di mitigare la variabilità e ottenere una stima più robusta del tempo medio di esecuzione.
3) **Misurazione del tempo:** Il tempo di esecuzione per ogni singolo run è stato misurato in Python utilizzando `time.time()` prima dell'inizio della logica della query e subito dopo il completamento dell'azione Spark. Il tempo registrato include quindi:
   - La lettura dei dati di input in formato Parquet da HDFS.
   - Tutte le trasformazioni Spark definite dalla logica della query (DataFrame, SQL o RDD).
   - L'esecuzione di un'azione Spark per forzare il calcolo dell'intero piano di esecuzione.
4) **Raccolta e analisi dei dati:** I tempi di esecuzione individuali sono stati raccolti e utilizzati per calcolare il tempo medio e la deviazione standard. I tempi medi sono stati poi salvati su file CSV memorizzati su HDFS, esportati su Redis, e utilizzati per generare i grafici presentati di seguito tramite Grafana.

B. Risultati

I grafici seguenti illustrano i tempi medi di esecuzione per le query 1, 2, 3 e 4 al variare del numero di executor attivi.

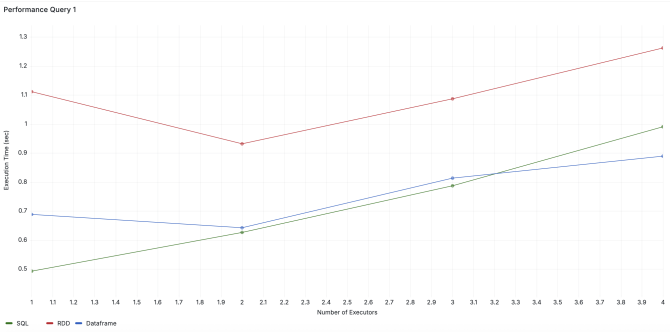

Fig. 15. Prestazioni per query 1 (aggregazione annuale) al variare del numero di executor per le API DataFrame, RDD e Spark SQL.

Query 1 (Fig. 15): La query 1, che esegue un'aggregazione standard, mostra che Spark SQL (linea verde) offre le migliori prestazioni, scalando bene fino a 2 executor, dopodiché i benefici del parallelismo aggiuntivo diminuiscono e il tempo tende ad aumentare leggermente. L'API DataFrame (linea blu) mostra un comportamento simile, con il punto ottimale a 2 executor. L'API RDD (linea rossa), pur migliorando significativamente da 1 a 2 executor, diventa meno efficiente con 3 e 4 executor. Questo suggerisce che per aggregazioni semplici, le ottimizzazioni del Catalyst Optimizer per SQL e DataFrame sono efficaci. L'aumento dei tempi oltre i 2 executor per tutte le API potrebbe indicare che per questo volume di dati e tipo di query, l'overhead di gestione di più task e la comunicazione iniziano a superare i vantaggi del parallelismo aggiuntivo, o che il dataset non è sufficientemente grande per beneficiare appieno di 3 o 4 core.

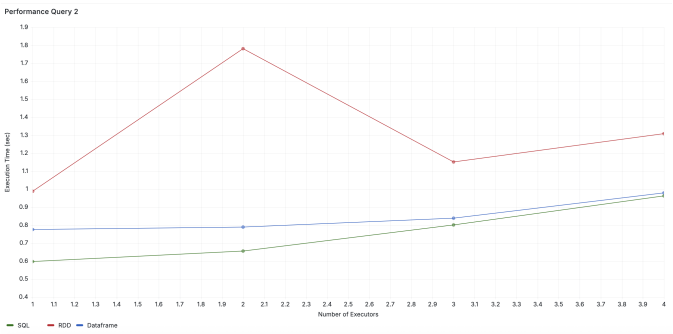

Fig. 16. Prestazioni per query 2 (aggregazione mensile e classifiche) al variare del numero di executor per le API DataFrame, RDD e Spark SQL.

Query 2 (Fig. 16): Per la query 2, più complessa a causa delle multiple operazioni di ordinamento e limitazione per le classifiche, Spark SQL (linea verde) si dimostra di nuovo l'API più effciente, con tempi di esecuzione costantemente bassi all'aumentare del numero di executor. Anche l'API DataFrame mostra performance simili. L'API RDD (linea rossa), invece, presenta un comportamento anomalo con un marcato picco negativo nelle prestazioni a 2 executor, per poi migliorare a 3 e peggiorare di nuovo a 4. Questo picco per l'RDD potrebbe essere dovuto a inefficienze nella gestione degli shuffle indotti dalle multiple operazioni di `sortBy` su un RDD memorizzato in cache, che potrebbero portare a contesa di risorse o a una cattiva distribuzione del lavoro con quella specifica configurazione di parallelismo.

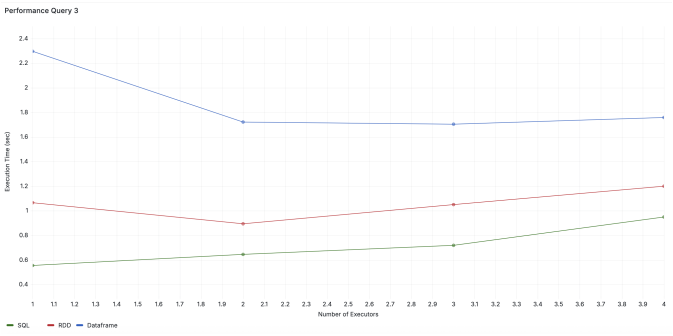

Fig. 17. Prestazioni per query 3 (analisi oraria e statistiche percentili) al variare del numero di executor per le API DataFrame, RDD e Spark SQL.

Query 3 (Fig. 17): Anche per la query 3, Spark SQL si dimostra l'approccio più efficiente, seguito stavolta dall'API RDD. L'implementazione DataFrame è notevolmente più lenta, pur mostrando un miglioramento all'aumentare degli executor. La performance inferiore dell'API DataFrame potrebbe essere dovuta al modo in cui le funzioni `F.expr("percentile(...)")` vengono tradotte e ottimizzate dal Catalyst Optimizer in questo specifico piano di esecuzione, che include due fasi di aggregazione (prima per le medie orarie, poi per i percentili su queste medie).

L'implementazione RDD, pur con calcoli manuali, potrebbe aver beneficiato di una gestione più diretta dei dati intermedi su un numero limitato di valori (24 medie orarie per paese) per il calcolo finale delle statistiche.

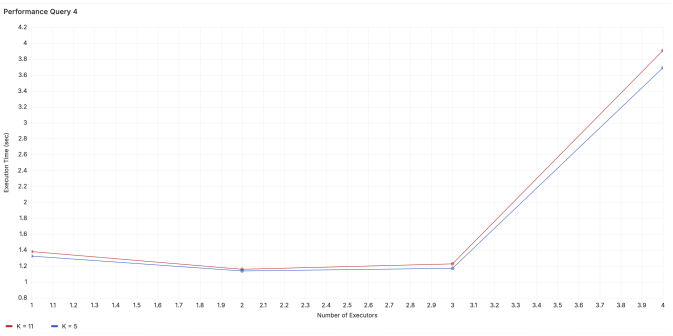

Fig. 18. Prestazioni per query 4 (clustering K-Means) al variare del numero di executor per k=11 e k=5.

Query 4 (Fig. 18): Il grafico delle prestazioni per la query 4 mostra i tempi di esecuzione dell'algoritmo K-Means per due valori di k: k=5 (ottenuto tramite Elbow Method) e k=11 (ottenuto tramite Silhouette Score). Per entrambi i valori, si osserva un comportamento simile, con una leggera diminuzione del tempo di esecuzione passando da 1 a 2 executor, e un andamento costante fino a 3. Entrambe le configurazioni mostrano un drastico aumento dei tempi di esecuzione quando si passa a 4 executor. Questo comportamento è tipico per algoritmi iterativi come K-Means quando applicati a dataset di dimensioni ridotte (30 paesi in questo caso) con un livello di parallelismo che supera il punto ottimale. Su dataset piccoli, l'overhead della distribuzione dei dati, della sincronizzazione delle iterazioni e della comunicazione tra un numero maggiore di task (anche se ognuno elabora pochissimi dati) può diventare predominante rispetto al tempo di calcolo effettivo, portando a un degrado delle prestazioni. Per questo problema, 2 o 3 executor sembrano essere sufficienti o addirittura ottimali.

VI. Conclusioni

I test evidenziano come Spark SQL sia generalmente l'API più performante e con il comportamento di scaling più prevedibile per le query analitiche strutturate, grazie alle ottimizzazioni del Catalyst Optimizer. L'API DataFrame si conferma una valida alternativa, anch'essa beneficiando di tali ottimizzazioni. L'API RDD, pur offrendo un controllo a basso livello, tende ad essere meno performante, specialmente per operazioni che hanno equivalenti ottimizzati nelle API di più alto livello o che richiedono una gestione manuale di logiche complesse e shuffle multipli. L'aumento del numero di executor ha mostrato benefici solamente fino a un certo punto (solitamente 2 executor), dopodiché i guadagni si riducono o i tempi possono addirittura aumentare. Questo indica che per questo volume di dati specifico e la complessità delle query, un parallelismo eccessivo non è sempre vantaggioso e l'overhead di gestione di Spark può diventare un fattore limitante. L'analisi dello scaling fornisce indicazioni utili per configurare le risorse Spark in modo ottimale per carichi di lavoro specifici.

Query	Type	Avg Execution Time (s)	Executors
Q1	dataframe	0.6881	1
Q1	rdd	1.1113	1
Q1	sql	0.4925	1
Q1	dataframe	0.6421	2
Q1	rdd	0.9316	2
Q1	sql	0.626	2
Q1	dataframe	0.8132	3
Q1	rdd	1.0867	3
Q1	sql	0.7869	3
Q1	dataframe	0.889	4
Q1	rdd	1.262	4
Q1	sql	0.9904	4

Query	Type	Avg Execution Time (s)	Executors
Q2	dataframe	0.7758	1
Q2	rdd	0.9885	1
Q2	sql	0.5977	1
Q2	dataframe	0.7893	2
Q2	rdd	1.7815	2
Q2	sql	0.656	2
Q2	dataframe	0.8388	3
Q2	rdd	1.1514	3
Q2	sql	0.8011	3
Q2	dataframe	0.9795	4
Q2	rdd	1.3088	4
Q2	sql	0.9627	4

Query	Type	Avg Execution Time (s)	Executors
Q3	dataframe	2.2978	1
Q3	rdd	1.0664	1
Q3	sql	0.5563	1
Q3	dataframe	1.7224	2
Q3	rdd	0.8951	2
Q3	sql	0.6464	2
Q3	dataframe	1.7052	3
Q3	rdd	1.0512	3
Q3	sql	0.7197	3
Q3	dataframe	1.7596	4
Q3	rdd	1.2007	4
Q3	sql	0.9502	4

Query	Clusters	Avg Execution Time (s)	Executors
Q4	5	1.3259	1
Q4	11	1.3828	1
Q4	5	1.139	2
Q4	11	1.1605	2
Q4	5	1.1725	3
Q4	11	1.2286	3
Q4	5	3.6932	4
Q4	11	3.9103	4

REFERENCES

[1] Apache Spark. [Online]. Available: https://spark.apache.org/
[2] Electricity Maps, "Carbon Intensity Data," 2025. [Online]. Available: https://portal.electricitymaps.com/datasets
[3] Docker. [Online]. Available: https://www.docker.com/
[4] Apache Hadoop. [Online]. Available: https://hadoop.apache.org/
[5] Apache NiFi. [Online]. Available: https://nifi.apache.org/
[6] Redis. [Online]. Available: https://redis.io/
[7] Grafana Labs, "Grafana - The open and composable observability and data visualization platform." [Online]. Available: https://grafana.com/grafana/