Sistemi e Architetture per Big Data - A.A. 2024/25

Report Progetto 1: Analisi di dati energetici con Apache Spark

Simone Niro
Matricola: 0350119
Università di Roma "Tor Vergata"

Lorenzo Brunori
Matricola: 0353481
Università di Roma "Tor Vergata"

Luigi Ciuffreda
Matricola: 0351898
Università di Roma "Tor Vergata"

***Abstract*—Questo report descrive un'architettura distribuita per l'analisi di dati energetici, progettata per rispondere a specifiche query usando Apache Spark, su dati storici forniti da Electricity Maps, relativi all'elettricità e alle emissioni di CO_2 associate alla sua produzione.**

***Index Terms*—Batch processing, Apache NiFi, Apache Spark, HDFS, Redis, Grafana, Docker Compose, Big Data**

I. Introduzione

Lo scopo del progetto è usare il framework di data processing Apache Spark per rispondere ad alcune query su dati storici relativi all'elettricità e alle emissioni di CO_2 per produrla, forniti da Electricity Maps.

Electricity Maps fornisce un dataset per ogni paese e per diverse granularità temporali. Il dataset di ogni paese contiene 35065 eventi, che descrivono la produzione di elettricità dal 1 gennaio 2021 al 31 dicembre 2024. Ogni evento è caratterizzato dalle seguenti informazioni:

- **CI (*Carbon intensity*)**: quantità di gas serra emessi per unità di elettricità, espressa in grammi di CO_2 equivalenti per kilowattora (gCO_2eq/kWh). Sono forniti sia i fattori di emissione diretti che quelli relativi al ciclo di vita.
- **CFE% (*Carbon-Free Energy Percentage*)**: percentuale di elettricità disponibile sulla rete da fonti a basse o nulle emissioni di CO_2.
- **RE% (*Renewable Energy Percentage*)**: percentuale di elettricità consumata da fonti rinnovabili.

II. Dataset

Sono stati utilizzati i dataset di Electricity Maps:

- relativi a Italia e Svezia, con granularità oraria, per ciascun anno dal 2021 al 2024. In questo caso, sono state considerate la CI e la CFE%.
- relativi a 15 paesi europei (Austria, Belgio, Francia, Finlandia, Germania, Gran Bretagna, Irlanda, Italia, Norvegia, Polonia, Repubblica Ceca, Slovenia, Spagna, Svezia e Svizzera) e a 15 paesi extra-europei (Stati Uniti, Emirati Arabi, Cina, India, Giappone, Brasile, Messico, Canada, Australia, Sudafrica, Corea del Sud, Indonesia, Singapore, Argentina, Egitto), con granularità annua, per l'anno 2024. In questo caso, è stata considerata esclusivamente la CI.

Infine, si precisa che da ora in poi il termine CI verrà utilizzato per indicare la componente relativa alle emissioni di carbonio *dirette*, escludendo il valore riferito al ciclo di vita.

III. Query

Sono state implementate le seguenti query:

1) **Q1.** Aggregare i dati relativi a Italia e Svezia su base annua. Per ciascun paese, calcolare la media, il minimo e il massimo di CI e CFE% per ciascun anno dal 2021 al 2024. Inoltre, considerando il valor medio delle metriche aggregate su base annua, generare due grafici che consentano di confrontare visivamente l'andamento per Italia e Svezia.
2) **Q2.** Aggregare i dati relativi solo all'Italia sulla coppia (anno, mese). Calcolare il valor medio di CI e CFE% e calcolare la classifica delle prime 5 coppie (anno, mese) in ordine crescente e decrescente rispetto a ciascuna metrica. Inoltre, considerando il valor medio delle metriche aggregate sulla coppia (anno, mese), generare due grafici che consentano di valutare visivamente l'andamento delle due metriche.
3) **Q3.** Aggregare i dati relativi a Italia e Svezia sulla base delle 24 fasce orarie giornaliere. Per ciascun paese, calcolare il valor medio di CI e CFE%, per poi calcolare il minimo, 25-esimo, 50-esimo, 75-esimo percentile e massimo del valor medio per ciascuna metrica. Inoltre, considerando il valor medio delle metriche aggregate sulle 24 fasce orarie giornaliere, generare due grafici che consentano di confrontare visivamente l'andamento per Italia e Svezia.
4) **Q4.** Eseguire un'analisi di clustering sui valori annuali di CI per l'anno 2024, relativi a 30 paesi (15 europei e 15 extra-europei). 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. Generare un grafico che consenta di visualizzare il risultato del clustering.

Sono stati valutati sperimentalmente i tempi di processamento delle query. In particolare, le prime tre query sono state implementate sia tramite le API DataFrame di Spark sia utilizzando SparkSQL, al fine di confrontare le prestazioni in termini di tempi di processamento. Inoltre, sono stati condotti esperimenti variando la configurazione di default di Spark per analizzare l'impatto di tale cambiamenti sulle performance.

I risultati delle query sono stati esportati in formato `.csv` e visualizzati usando Grafana come framework di visualizzazione.

IV. Architettura

Il sistema è stato sviluppato su un nodo standalone, utilizzando container Docker per ogni componente ed orchestrando il tutto tramite Docker Compose.

A. Schema Architetturale

La Figura 1 illustra lo schema architetturale del sistema, evidenziando le interazioni tra i vari servizi e il percorso seguito dai dati. In particolare, i componenti principali che la compongono sono:

- **Apache NiFi**: si occupa dell'acquisizione dei dati dal sito di Electricity Maps, del loro pre-processamento e della loro ingestione su HDFS.
- **Hadoop Distributed File System (HDFS)**: il file system distribuito per la memorizzazione dei dati.
- **Apache Spark**: esegue le query, ne misura le prestazioni e scrive il tutto su HDFS.
- **Redis**: recupera da HDFS i risultati delle query e le performance, per poi esportarli in locale.
- **Grafana**: consente la visualizzazione interattiva dei risultati e delle prestazioni, salvate in locale.
- **Locale**: tramite una CLI è possibile gestire, in maniera user-friendly, tutte le operazioni del sistema oltre alla gestione dei container Docker.

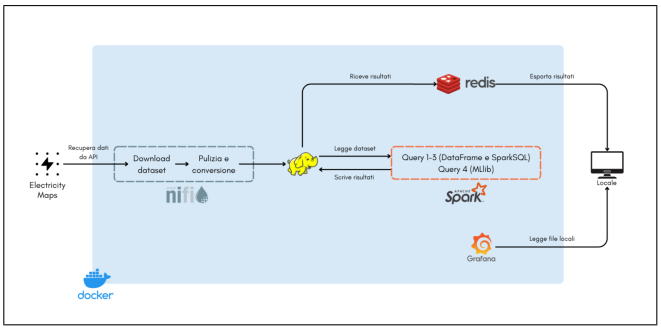


Fig. 1: Architettura del sistema

B. Cluster Docker Compose

Tra i vari container Docker che compongono il sistema, quelli relativi ad HDFS e Spark sono quelli che meritano una vista più dettagliata, in quanto costituiscono due cluster distinti.

Come mostrato in Figura 2, il cluster HDFS è composto da un *NameNode* e due *DataNode*, mentre il cluster Spark da uno *Spark Master* e due *Spark Worker*. Ogni componente è eseguito all'interno di un container dedicato.

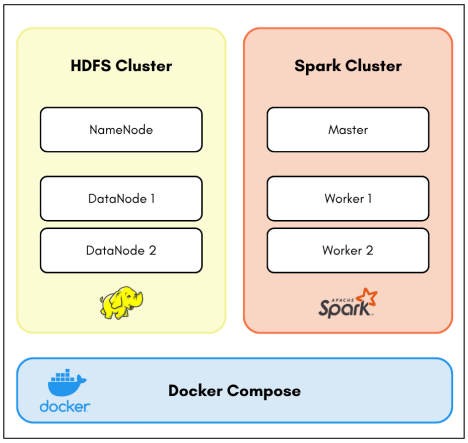


Fig. 2: Cluster di HDFS e Spark

Per quanto riguarda i restanti componenti (i.e. NiFi, Redis e Grafana), essi non presentano una struttura particolare da evidenziare.

C. HDFS

I file di configurazione necessari per l'esecuzione di HDFS sono montati tramite volumi condivisi, e risiedono nella directory locale `hdfs/conf`. Nel dettaglio, `hdfs-site.xml` definisce il fattore di replicazione (pari a 2, coerente con il numero di DataNode disponibili), i path alle directory dei dati nei container e i permessi, mentre `core-site.xml` specifica l'endpoint del NameNode, che è `hdfs://namenode:9000`.

Il container del NameNode esegue uno script di avvio (`start-hdfs.sh`) che formatta il NameNode e lancia il servizio, introducendo un'attesa iniziale di 60 secondi per garantire l'avvio corretto prima che altri componenti accedano al cluster. Inoltre, si occupa della creazione di due directory principali (modificandone i permessi per consentire la scrittura):

- `/data`: contiene i dati ricevuti da NiFi;
- `/results`: raccoglie i risultati delle query Spark e i file relativi alle performance.

Dunque, non è prevista alcuna logica applicativa attiva all'interno di HDFS.

D. Apache Spark

Nel container dello Spark Master sono configurati:

- La `SPARK_MODE=master`, che indica che il container esegue il ruolo di master nel cluster Spark.

- L'endpoint HDFS, definito tramite la variabile d'ambiente `HDFS_BASE=hdfs://namenode:9000`.
- Una porta per la gestione delle richieste di esecuzione delle query da parte dei client ed una per l'accesso alla web UI di Spark.

Inoltre, sono stati montati i volumi:

- `hdfs/conf`: contiene i file di configurazione di HDFS.
- `spark/app`: include gli script necessari per l'esecuzione delle query e per la valutazione delle performance.
- `spark/conf`: contiene un file di configurazione personalizzato per Spark.

I nodi worker sono configurati per connettersi al master tramite l'hostname specificato e non espongono alcuna interfaccia web.

E. Redis

L'infrastruttura di Redis include due container distinti:

- `redis`: il database Redis vero e proprio, esposto sulla porta `6379`.
- `redis-loader`: container dedicato al caricamento e all'esportazione dei file CSV da e verso Redis, tramite gli script `load.py` e `export.py` situati nella directory `redis/loader/`.

Più nel dettaglio, la gestione dei file CSV in Redis si articola principalmente in due fasi:

1) *Caricamento da HDFS a Redis*: lo script `load.py` esplora ricorsivamente la directory dei risultati su HDFS, individua i file `.csv` e li carica in Redis. Ogni file viene salvato con una chiave del tipo `hdfs_data:<nome_directory>.csv`.
2) *Esportazione da Redis in locale*: lo script `export.py` recupera tutte le chiavi che iniziano con `hdfs_data:`, ne legge i contenuti e salva i file corrispondenti nella cartella locale `results` della macchina host.

V. Data Ingestion and Preprocessing

A. Data Acquisition

Come mostrato in Figura 3, il workflow definito in Apache NiFi inizia con il processor `ListenHTTP` che espone un endpoint (`localhost:1406`) al quale uno script Python invia richieste contenenti gli URL da cui scaricare i dati.

Il processor `ExtractText` estrae dinamicamente l'URL dal corpo della richiesta HTTP, mentre `InvokeHTTP` effettua il download del corrispondente file CSV.

A seguito del download, un `RouteOnAttribute` consente di suddividere i dati in base alla loro granularità (identificata dalla presenza dei termini `hourly` o `yearly` nell'URL), il che ha permesso di differenziare le operazioni di preprocessamento.

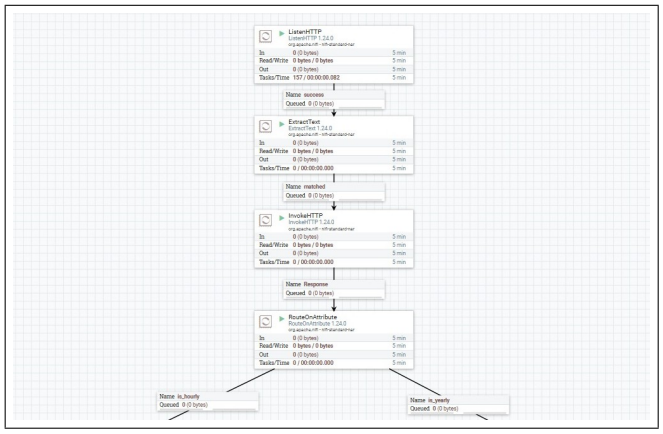


Fig. 3: Parte del workflow NiFi dedicata all'acquisizione dei dati

B. Preprocessing and Data Ingestion

Per i dati orari: (i) sono state selezionate solo le colonne rilevanti; (ii) i file dei diversi anni sono stati aggregati, per ciascun paese, in un unico file; (iii) i dati sono stati convertiti in formato Parquet.

Per i dati annuali: (i) è stato selezionato un sottoinsieme di colonne differente rispetto a quello dei dati orari; (ii) i file dei vari paesi per il 2024 sono stati aggregati in un unico file; (iii) i dati sono stati mantenuti in formato CSV.

Nel dettaglio (Figura 4), ciascun ramo successivo al `RouteOnAttribute` è composto, nell'ordine, dai seguenti processor:

1) `QueryRecord`: seleziona le colonne rilevanti per l'analisi.
2) `UpdateAttribute`: genera una *merge key* basata sul nome del file (e.g. "{Zone id}_hourly" e "{Year}_yearly").
3) `MergeRecord`: unisce i record in un singolo file, utilizzando la *merge key*, e lo converte nel formato adeguato tramite il rispettivo *Controller Service* (Parquet o CSV).
4) `UpdateAttribute`: definisce il nome finale del file di output.

Infine, entrambi i rami convergono nel processor `PutHDFS`, che salva i file preprocessati nella cartella `data` su HDFS.

C. Considerazioni sul Preprocessamento

Le scelte relative al pre-processamento sono state dettate da diverse considerazioni (a volte frutto di alcuni tentativi).

- La rimozione delle colonne non rilevanti, in base alla granularità, ha permesso di ridurre il volume complessivo dei dati. Nonostante le dimensioni originali non fossero eccessive, si è ritenuto opportuno effettuare questa operazione per garantire un dataset più inerente alle analisi previste.

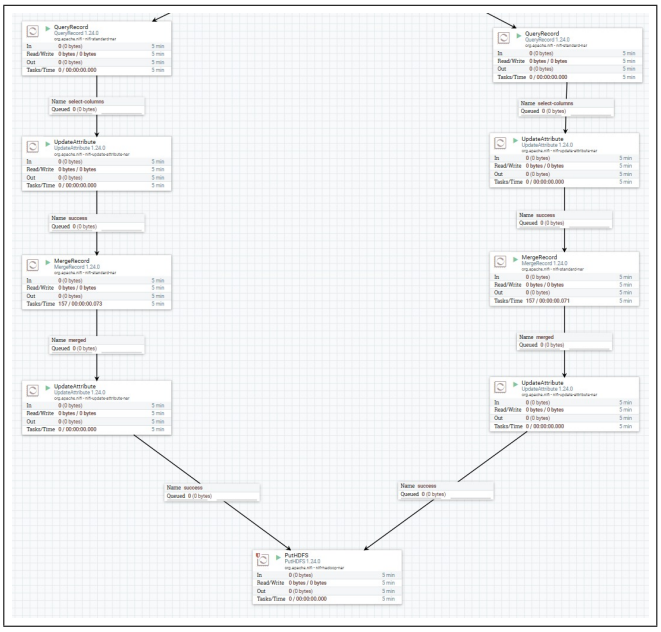


Fig. 4: Seconda parte del workflow di NiFi

- La conversione in formato Parquet per i dati orari ha migliorato le prestazioni di lettura/scrittura su HDFS, grazie all'organizzazione colonnare.
- La scelta di non convertire i dati annuali in formato Parquet è stata consapevole: il numero esiguo di record già aggregati per il 2024 non giustificava l'overhead introdotto dalla conversione.

Questa fase, quindi, ha permesso di ottenere un dataset già pertinente e ottimizzato per le esigenze specifiche del progetto.

D. Gestione del Template NiFi

Per facilitare la fase di deploy e rendere più agevole l'istanziazione del flusso su ambienti diversi, è stato realizzato uno script denominato `import-template.sh`. Lo script consente, una volta eseguito, di importare, istanziare e avviare automaticamente il template NiFi specificato nella cartella `nifi/conf` del progetto, senza richiedere ulteriori interventi manuali.

VI. Data Processing

Il processamento dei dati è stato realizzato utilizzando *PySpark*, che offre un'interfaccia in linguaggio Python per Apache Spark. Le query vengono eseguite tramite `functions.py`, che lancia `spark-submit` all'interno del container `spark-master`. Questo comando avvia il *Driver Program*, che si connette allo `spark-master` per coordinare l'elaborazione distribuita sui nodi *Worker* del cluster. I parametri principali specificati per ogni esecuzione sono i file di input, la directory di output, il file Python per lo script di valutazione delle performance e il numero di esecuzioni.

A. Struttura delle Query

Lo script di implementazione di ciascuna query presenta la seguente struttura:

1) Blocco `if __name__ == "__main__"`: raccoglie i parametri, inizializza la SparkSession, istanzia la classe `Evaluation` per la misurazione delle prestazioni, invoca la funzione `main_<query_id>`, e salva i risultati delle performance.
2) Funzione `main_<query_id>`: esegue la query e ne salva i risultati in HDFS.

B. RDD vs DataFrame

Per tutte le query si è scelto di utilizzare l'API DataFrame. Questa scelta è stata dettata dalla natura strutturata dei dati di input, che ha reso i DataFrame una soluzione efficace e intuitiva. Gli RDD offrono una maggiore flessibilità grazie alle loro operazioni a basso livello, ma tale vantaggio risulta più utile in presenza di dati non strutturati o semi-strutturati, un contesto operativo diverso da quello affrontato.

I DataFrame consentono a Spark di sfruttare il *Catalyst Optimizer*, che è in grado di ottimizzare l'esecuzione delle query su dati strutturati generando piani di esecuzione efficienti, che spesso superano le prestazioni ottenibiili con gli RDD. Un ulteriore beneficio risiede nella loro API più intuitiva e leggibile, che si presta bene a una possibile integrazione con SparkSQL. Infatti, per ogni query, il passaggio verso SparkSQL è stato semplice, limitandosi alla sostituzione delle trasformazioni con una corrispondente query SQL.

C. Implementazione Query 1

La *Query 1* opera su due distinti file in formato Parquet, uno per ciascuna nazione. La funzione `process_file`, invocata dal main, è responsabile delle seguenti operazioni:

1) Il file Parquet specificato viene caricato in un DataFrame Spark, verificando la presenza delle colonne essenziali per l'analisi.
2) Dal timestamp, viene estratta la colonna "year". Il DataFrame viene raggruppato per quest'ultima e vengono calcolati la media, il minimo e il massimo.
3) Al DataFrame risultante viene aggiunta la colonna "country" contenente l'identificativo della nazione.

Elaborati i dati da `process_file` per entrambe le nazioni, i due Dataframe vengono uniti per poi essere scritti in HDFS in formato CSV.

Nella versione SparkSQL, dopo aver caricato i dati in un DataFrame, le operazioni precedentemente specifiche per i DataFrame vengono sostituite da una query SQL che opera su una vista temporanea dei dati.

D. Implementazione Query 2

La *Query 2* prende in input solo il file Parquet dei dati italiani. La funzione `process_file` esegue le seguenti operazioni:

1) Il file Parquet specificato viene caricato in un DataFrame Spark verificando la presenza delle colonne essenziali per l'analisi.

2) Dal timestamp vengono estratti mese e anno, creando nuove colonne dedicate. Il DataFrame viene poi raggruppato per le coppie (anno, mese) e per ciascun gruppo vengono calcolate le medie.

3) Il DataFrame risultante diventa input della funzione `calculate_rankings`, che identifica le 5 coppie (anno, mese) richieste dalla traccia per ciascuna metrica e crea un DataFrame per ciascuna di esse.

I quattro DataFrame creati vengono aggregati in un unico DataFrame, che viene poi scritto in HDFS in formato CSV.

Il passaggio a SparkSQL segue un approccio del tutto analogo a quello visto per Q1, con la differenza che vengono utilizzate due viste temporanee: una per il DataFrame iniziale e una per il DataFrame aggregato per coppie (anno, mese).

E. Implementazione Query 3

La *Query 3*, come *Query 1*, opera su due file Parquet, uno per l'Italia e uno per la Svezia. La funzione `process_file` è responsabile delle seguenti operazioni:

1) Il file Parquet specificato viene caricato in un DataFrame Spark verificando la presenza delle colonne essenziali per l'analisi-

2) Dal timestamp viene estratta l'ora, creando una nuova colonna. Il DataFrame viene quindi raggruppato per fascia oraria e, per ciascun gruppo, vengono calcolate le medie.

3) Infine, vengono calcolate le statistiche descrittive desiderate sulle medie ottenute.

Elaborati i dati da `process_file` per entrambe le nazioni, i due Dataframe vengono uniti per poi essere scritti in HDFS in formato CSV.

Come per le query precedenti, la versione SparkSQL segue lo stesso flusso logico, ma traduce le operazioni in sintassi SQL.

F. Implementazione Query 4

La *Query 4* implementa un'analisi di clustering K-Means sfruttando le funzionalità di machine learning nel modulo `pyspark.ml`. L'elaborazione è orchestrata dalla funzione `process_clustering`:

1) Il file CSV specificato viene caricato in un DataFrame Spark verificando la presenza delle colonne essenziali per l'analisi.

2) Poiché il K-Means richiede che le feature siano consolidate in un singolo vettore, viene utilizzato `VectorAssembler` per trasformare la colonna `carbon-intensity` in un vettore di feature.

3) Il vettore di feature viene scalato utilizzando `StandardScaler`, che normalizza i dati per evitare che feature con scale di valori molto diverse influenzino il calcolo della distanza. I dati non vengono centrati sulla media, ma solo scalati dalla deviazione standard.

4) Viene eseguita una ricerca del numero ottimale di cluster k utilizzando l'indice di Silhouette:
   a) Per ogni valore di k nell'intervallo specificato, viene creato un modello k-Means e addestrato sui dati scalati.
   b) Viene calcolato l'indice di Silhouette medio per il modello addestrato.
   c) Il valore di k che massimizza l'indice di Silhouette medio viene selezionato come ottimale.

5) Il modello k-Means viene riaddestrato sull'intero dataset scalato utilizzando il valore di k ottimale.

6) I centri dei cluster vengono estratti e scalati indietro nel dominio originale dei dati.

7) Viene costruito un DataFrame contenente il nome del paese, la CI, l'ID del cluster assegnato, il centro del cluster (in coordinate originali) e la distanza dal centro del cluster assegnato.

Elaborati i dati da `process_clustering`, il DataFrame risultante viene scritto in HDFS in formato CSV.

VII. Data Visualization

Il framework Grafana è stato eseguito all'interno di un container Docker, configurato per accedere ai risultati in formato CSV generati da Spark. Tali file sono accessibili nella cartella locale `results`, condivisa con Grafana tramite un volume Docker, sebbene l'accesso effettivo ai dati avvenga attraverso un processo manuale.

Per consentire l'interpretazione dei file CSV come sorgente dati, è stato utilizzato il plugin *Infinity*. L'installazione del plugin avviene automaticamente all'avvio del container, grazie a un file `grafana.ini` personalizzato collocato nella directory `grafana`.

A. Configurazione della sorgente dati

Per automatizzare la configurazione della sorgente dati, è stato creato un file YAML di provisioning nella cartella `provisioning/datasources`, che definisce e registra automaticamente la datasource `yesoreyeram-infinity-datasource` all'avvio del container.

Tuttavia, il plugin *Infinity* non supporta la lettura automatica di file locali dal file system del container. Pertanto, per ogni pannello della dashboard è stato necessario prima selezionare manualmente il file CSV desiderato dalla cartella locale `results` della macchina host, e poi configurare i campi e le query per ogni visualizzazione.

B. Dashboard e provisioning

Grafana è stato impostato per caricare dashboard personalizzate tramite un meccanismo di provisioning. In particolare, il file YAML `provisioning/dashboards.yml` espone la directory `grafana/dashboards`, contenente i file JSON che definiscono la struttura delle dashboard. Questa configurazione garantisce che le dashboard vengano caricate automaticamente e rese immediatamente disponibili.

Tuttavia, a ogni riavvio del sistema, i singoli pannelli potrebbero inizialmente mostrare lo stato *No data*: in tal caso, affinché i dati vengano visualizzati correttamente, è necessario effettuare un refresh manuale all'interno di ciascun pannello.

La limitazione sui file del plugin *Infinity* è stata accettata dopo aver valutato diverse alternative per la visualizzazione dei dati CSV. Altri plugin disponibili per Grafana, sebbene supportino la lettura automatica di file locali, presentavano limitazioni significative in termini di flessibilità nella rappresentazione grafica dei risultati. *Infinity*, nonostante richieda configurazione manuale, offre maggiore controllo sui formati di output e migliore compatibilità con i dati strutturati dei risultati delle query.

VIII. Risultati

Tutti i grafici generati a partire dai risultati delle query sono osservabili nelle Figure 5, 6, 7, 8, 9, 10 e 11.

IX. Prestazioni

A. Misurazione delle prestazioni

Le prestazioni vengono misurate da Spark grazie a `app/evaluation.py`, che definisce la classe `Evaluation` dedicata alla raccolta e all'analisi dei tempi di esecuzione.

La prima esecuzione di una query Spark tende ad avere tempi significativamente superiori rispetto alle successive, a causa delle inizializzazioni di basso livello (e.g. JVM, sessione Spark, caricamento librerie). In alcuni casi, il tempo di esecuzione può risultare fino a 20 volte maggiore. Per evitare che ciò influenzi le statistiche, viene sempre aggiunta un'esecuzione rispetto al numero indicato dall'utente, permettendo così di scartare la prima misura nel calcolo finale delle metriche.

Nello script di ciascuna query, la *SparkSession* viene inizializzata una sola volta all'inizio del processo e mantenuta attiva per tutte le esecuzioni. Dopo aver definito l'istanza di `Evaluation`, ne viene invocato il metodo `run()`, che esegue la query per il numero di volte specificato dall'attributo `runs`.

La misurazione del tempo avviene esclusivamente durante l'esecuzione effettiva della query e comprende: lettura dei file di input, elaborazione dei dati e scrittura dei risultati. Gli overhead di inizializzazione e chiusura della *SparkSession* sono esclusi dalla misurazione, garantendo che i risultati riflettano le prestazioni dell'elaborazione dati pura.

La misurazione avviene all'interno di un blocco `try-except`: se l'esecuzione ha esito positivo, il tempo viene registrato, altrimenti viene ignorato. Al termine di tutte le esecuzioni, viene invocato il metodo `evaluate()`, che verifica la presenza di almeno una run valida e, se i dati lo consentono, calcola tempo medio, massimo, minimo e deviazione standard.

Infine, le misurazioni vengono salvate in un file CSV all'interno della directory di output su HDFS, in una sottocartella dedicata chiamata `evaluation_<query_id>`.

Le specifiche tecniche della macchina utilizzata per le esecuzioni sono riportate nella Tabella I.

B. Confronto delle prestazioni

Per esplorare al meglio le prestazioni del sistema, sono state utilizzate quattro diverse configurazioni, definite nel file `conf/spark-defaults.conf`. I risultati ottenuti sono riportati nelle tabelle II, III, IV e V.

Queste configurazioni sono state scelte per valutare l'impatto di alcune variazioni nei parametri di Spark sulle prestazioni delle query. Di default, Spark assegna 16 core a ciascun worker, valore che coincide con il numero massimo di thread disponibili sulla macchina di test: modificarlo avrebbe avuto un impatto diretto e potenzialmente distorsivo sulle performance e, pertanto, si è scelto di mantenerlo invariato.

Le variabili modificate nelle diverse configurazioni sono state il numero di executor, la memoria RAM assegnata a ciascuno di essi e la memoria riservata al driver. Questa scelta è stata motivata dal possibile overhead derivante dall'aumento del parallelismo, che richiede una maggiore disponibilità di risorse.

L'analisi dei risultati mostra come l'aumento del numero di executor comporta generalmente un miglioramento delle prestazioni in quasi tutte le query, grazie a un livello maggiore di parallelizzazione. In particolare, la *Query 4* ottiene i risultati migliori con 4 executor.

Tuttavia, un'analisi più approfondita evidenzia che l'incremento delle risorse non porta sempre a benefici significativi: in alcuni casi, le prestazioni rimangono pressoché invariate, mentre in altri casi possono addirittura peggiorare (e.g. *Query 1*). Quindi, è possibile concludere che, per le query implementate, la configurazione di default sembra rappresentare il miglior compromesso tra prestazioni e utilizzo delle risorse.

C. Confronto tra SQL e DataFrame API

Nel confronto tra le prestazioni delle query implementate in SQL e quelle sviluppate utilizzando le API DataFrame, si è osservato un leggero vantaggio a favore dell'implementazione in SQL. Questo risultato può essere attribuito alla natura fortemente strutturata dei dati e al fatto che le operazioni eseguite sono principalmente di filtraggio e aggregazione, particolarmente ottimizzate in SQL. Possiamo quindi affermare che, dato il dominio dei dati in analisi, questo comportamento era in parte atteso.

Le API DataFrame rappresentano comunque un'alternativa valida e flessibile, garantendo prestazioni paragonabili a quelle ottenute con SQL. Tuttavia, in presenza di dati ben strutturati e query semplici, l'utilizzo di SQL può consentire a Spark di

Fig. 5: Grafico della Query 1 per l'intensità di carbonio

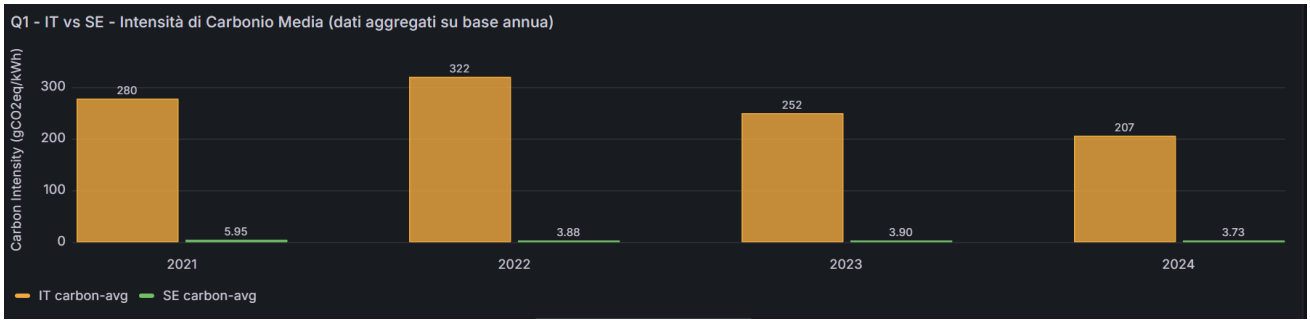


Fig. 6: Grafico della Query 1 per la CFE

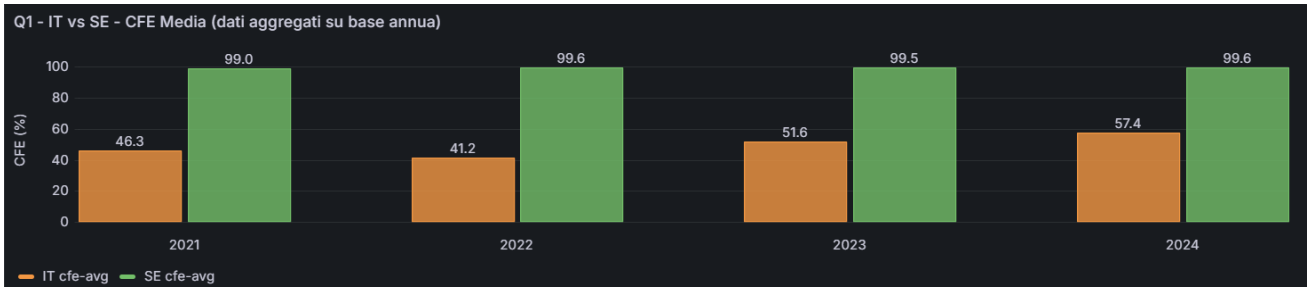


Fig. 7: Grafico della Query 2 per l'intensità di carbonio

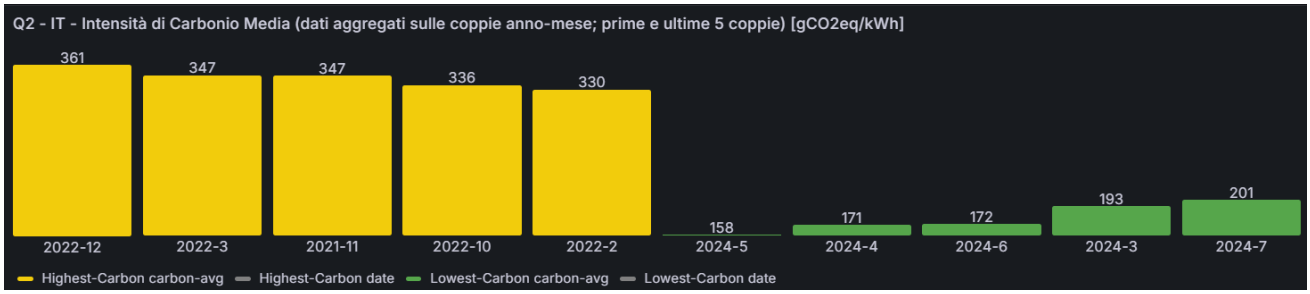
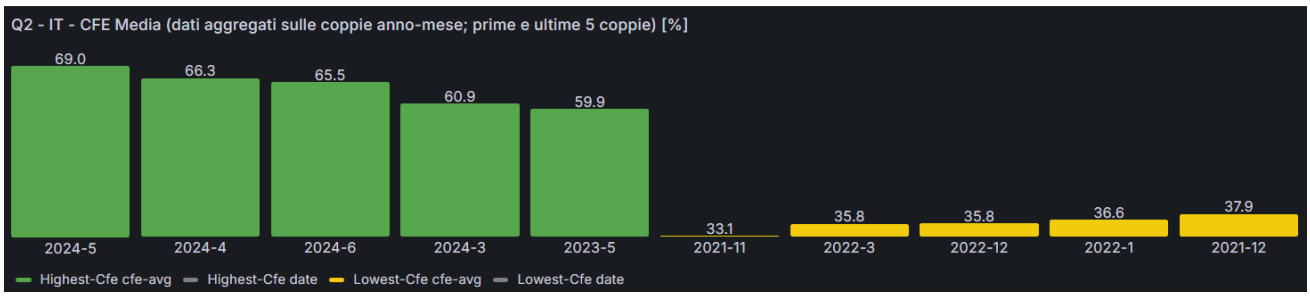


Fig. 8: Grafico della Query 2 per la CFE



generare piani di esecuzione leggermente più ottimizzati, con un leggero vantaggio prestazionale.

Fig. 9: Grafico della Query 3 per l'intensità di carbonio

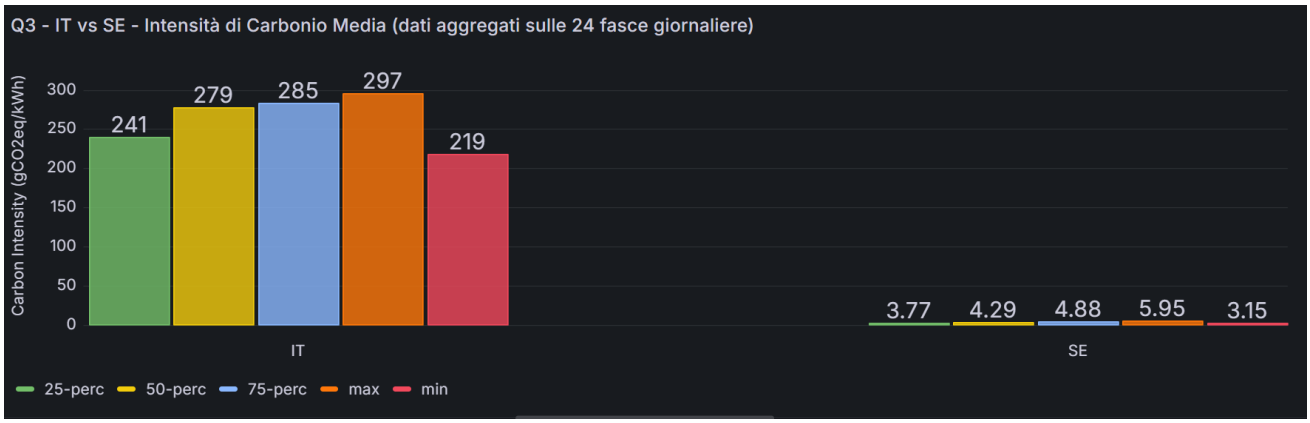


Fig. 10: Grafico della Query 3 per la CFE

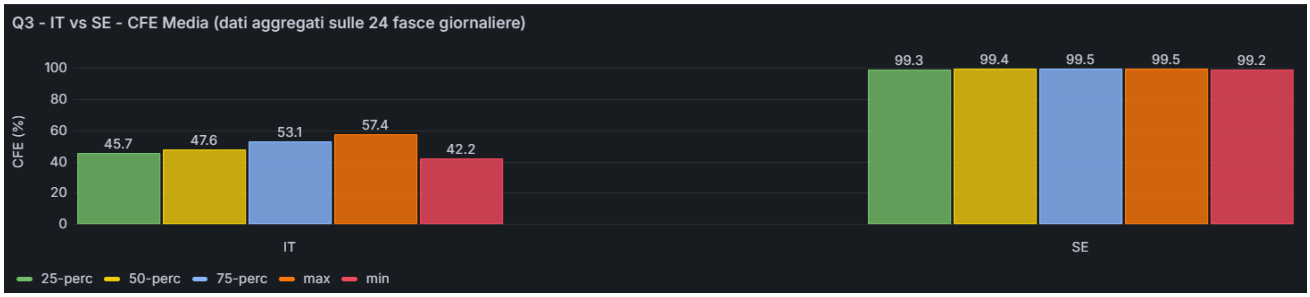


Fig. 11: Grafico della Query 4

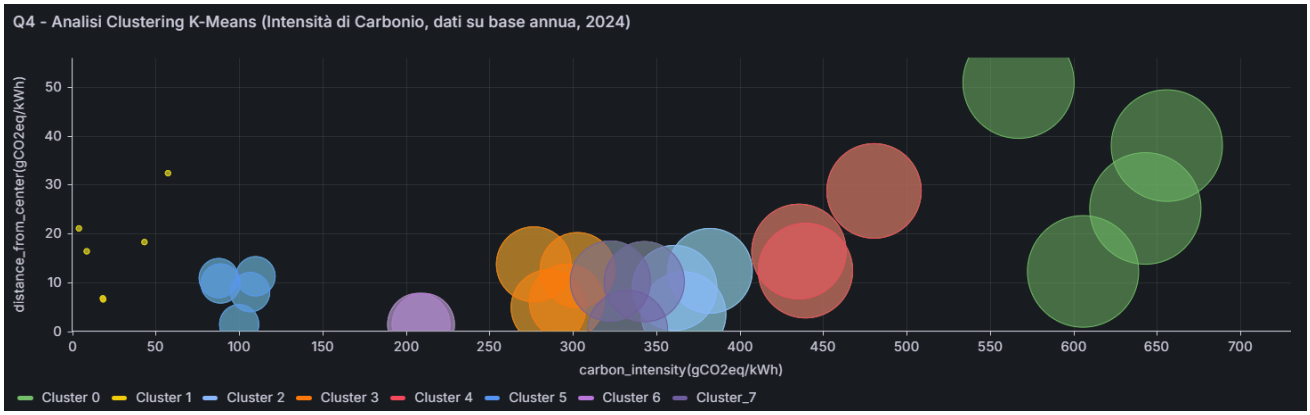


TABLE I: Specifiche tecniche della macchina utilizzata per i test.

RAM	CPU
32 GB DDR4	Ryzen 7 5700x3D 8c-16t

TABLE II: Tempi ottenuti con la configurazione di default di Spark: 1 *executor*. 20 esecuzioni per ciascuna query.

Query	Mode	Avg Time (s)	Min Time (s)	Max Time (s)	Std. Dev. (s)
Query 1	Spark	0.5844	0.4112	2.0094	0.3500
Query 1	SparkSQL	0.5590	0.3752	2.0638	0.3727
Query 2	Spark	0.5386	0.3394	1.9787	0.4266
Query 2	SparkSQL	0.5630	0.3222	3.6144	0.7236
Query 3	Spark	0.7222	0.5024	2.8251	0.5073
Query 3	SparkSQL	0.6383	0.4139	2.1297	0.4045
Query 4	Spark	14.032	12.3411	19.2202	1.6461

TABLE III: Tempi ottenuti con *Conf1*: 2 *executor* Spark, 4GB di RAM, 2GB di memoria per il driver. 20 esecuzioni per ciascuna query.

Query	Mode	Avg Time (s)	Min Time (s)	Max Time (s)	Std. Dev. (s)
Query 1	Spark	0.5623	0.4347	1.1418	0.1748
Query 1	SparkSQL	0.5595	0.3863	2.1059	0.3842
Query 2	Spark	0.6514	0.3563	2.1327	0.5543
Query 2	SparkSQL	0.6184	0.3224	2.7553	0.6354
Query 3	Spark	0.7454	0.5433	2.1603	0.3807
Query 3	SparkSQL	0.6421	0.4378	2.0692	0.3904
Query 4	Spark	14.2317	12.2402	19.5963	1.8416

TABLE IV: Tempi ottenuti con *Conf2*: 3 *executor* Spark, 4GB di RAM, 2GB di memoria per il driver. 20 esecuzioni per ciascuna query.

Query	Mode	Avg Time (s)	Min Time (s)	Max Time (s)	Std. Dev. (s)
Query 1	Spark	0.5956	0.4111	2.0995	0.3739
Query 1	SparkSQL	0.5916	0.3621	2.5168	0.4667
Query 2	Spark	0.5509	0.3470	1.9496	0.4208
Query 2	SparkSQL	0.5276	0.3131	1.9084	0.4173
Query 3	Spark	0.6492	0.5094	1.2807	0.1807
Query 3	SparkSQL	0.5724	0.4154	1.1546	0.1799
Query 4	Spark	13.8389	12.4094	18.7573	1.5336

TABLE V: Tempi ottenuti con *Conf3*: 4 *executor* Spark, 4GB di RAM, 2GB di memoria per il driver. 20 esecuzioni per ciascuna query.

Query	Mode	Avg Time (s)	Min Time (s)	Max Time (s)	Std. Dev. (s)
Query 1	Spark	0.6418	0.4152	2.2627	0.4425
Query 1	SparkSQL	0.5158	0.3813	1.0441	0.1670
Query 2	Spark	0.5145	0.3373	1.9485	0.3556
Query 2	SparkSQL	0.4913	0.3175	1.8470	0.3361
Query 3	Spark	0.6568	0.5289	1.3677	0.1964
Query 3	SparkSQL	0.5887	0.4148	1.3729	0.2570
Query 4	Spark	13.8007	12.4984	18.0333	1.4305