Relazione 1° progetto SABD

Giuseppe Marseglia
matricola n. 0350066
Università di Roma Tor Vergata

***Abstract*—Questo documento ha lo scopo di illustrare il design, i dettagli implementativi e i risultati ottenuti di un sistema di batch data processing basato in grado di rispondere a delle query sui dati forniti da Electricity Maps. In particolare, il sistema è stato realizzato con il framework Apache Spark e Python.**

I. Introduzione

A. Dataset

Electricity Maps mette a disposizione dei dataset che contengono diverse informazioni riguardanti la produzione ed il consumo di energia elettrica di diversi paesi. Le informazioni di interesse per il sistema sono:

- "quantità di gas serra emessi per unità di elettricità, espressa in grammi di CO_2 equivalenti per kilowattora ($gCO2eq/kWh$)". In particolare la statistica legata ai fattori di emissioni dirette. Questa informazione verrà indicata come `carbon-intensity`.
- "percentuale di elettricità disponibile sulla rete da fonti a basse o nulle emissioni di CO_2." Questa informazione verrà indicata come `cfe`.

I dati sono disponibili con diverse granularità temporali (ore, giorni, mesi, anni) e spaziali (per stato, per regione). Nel sistema sono stati adottati i dati con granularità temporale delle ore ed entrambe le granularità spaziali.

B. Specifica assegnata

Nella specifica assegnata sono indicate le query a cui il sistema deve rispondere:

- Q_1: "Facendo riferimento al dataset dei valori energetici di Italia e della Svezia, aggregare i dati su base annua. Calcolare la media, il minimo ed il massimo di `carbon-intensity` e `cfe` per ciascun anno dal 2021 al 2024."
- Q_2: "Considerando il solo dataset italiano, aggregare i dati sulla coppia (anno, mese), calcolando il valore medio di `carbon-intensity` e `cfe`. Calcolare la classifica delle prime 5 coppie `(anno, mese)` ordinando per `carbon-intensity` decrescente, crescente e `cfe` decrescente, crescente. In totale sono attesi 20 valori."

La specifica richiede che i dati vengano letti e scritti tramite HDFS. Il formato di lettura non è specificato, mentre il formato di scrittura è specificato in CSV.

Inoltre è richiesta la produzione di 4 grafici, 2 per query, tramite un framework di visualizzazione:

- Andamento su base annua del valore medio di `carbon-intensity` per Italia e Svezia.
- Andamento su base annua del valore medio di `cfe` per Italia e Svezia.
- Andamento su base mensile del valore medio di `carbon-intensity` per Italia.
- Andamento su base mensile del valore medio di `cfe` per Italia.

Infine è richiesta la valutazione sperimentale dei tempi di processamento delle query sulla piattaforma di riferimento usata per la realizzazione del progetto.

Tra le richieste opzionali della specifica, quelle implementate sono state:

- La scrittura dell'output verso un sistema di storage a scelta (diverso da HDFS).
- L'utilizzo di Spark SQL e la relativa valutazione delle performance.

II. Architettura

A. Componenti

Il sistema è composto da 4 componenti, come riportato in Fig.1:

1) **HDFS**: framework per File System distribuiti, usato per lo storage del dataset e del output in formato CSV.
2) **Apache Spark**: framework per batch e stream data processing, usato per il processamento batch dei dati.
3) **InfluxDB**: framework per DB a serie temporali, usato come storage secondario oltre a HDFS. La scelta di InfluxDB è stata influenzata dal tipo di dati di cui vanno prodotti i grafici e dalla nativa compatibilità con il framework di visualizzazione di dati citato sotto.
4) **Grafana**: framework per visualizzazione di dati, usato per produrre i grafici richiesti.

Le frecce in Fig.1 rappresentano i flussi di dati:

- Precedentemente all'avvio del processamento, i dati vengono inseriti in HDFS manualmente dall'esterno.
- All'avvio del processamento, Spark legge i dati da HDFS.
- Al completamento del processamento, Spark scrive i risultati delle query su HDFS e i dati di cui produrre i grafici su InfluxDB.
- All'accesso della Web UI, Grafana legge i dati da InfluxDB.

III. Deployment

A. Piattaforma

Il deployment dell'architettura è stato effettuato su un nodo standalone tramite l'uso di container via Docker Compose.

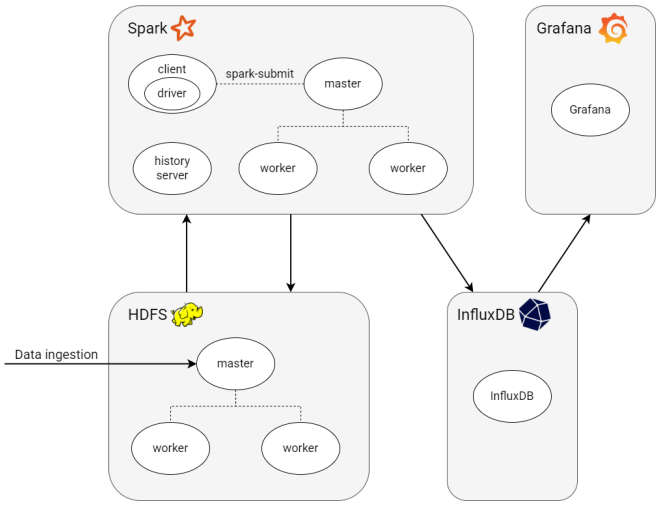

Fig. 1: Architettura del sistema

In particolare, il nodo standalone è una macchina Windows con supporto per Windows Linux Subsystem (WSL). L'orchestrazione del cluster è stata implementata tramite script bash.

B. Dettagli di deployment

Di seguito vengono riportati alcuni dettagli di deployment riguardanti i vari componenti del sistema visibili in Fig.1.

Per **HDFS**:

- HDFS è utilizzato in modalità cluster, con 1 NameNode e 2 DataNode.
- Il grado di replicazione dei chunk è stato impostato a *2*.
- La data ingestion viene effettuata al momento dell'avvio del NameNode: dal File System locale il dataset viene importato nel container tramite Bind Mounts, e poi caricato in HDFS da script.

Per **Spark**:

- La versione di Spark usata è la *3.5.5*.
- Spark è utilizzato in modalità Standalone Cluster, con 1 Master e 2 Worker.
- Il Client è all'interno di un container e lancia lo script `spark-submit` specificando come tipo di deployment `--deploy client`. Questa, oltre ad essere l'impostazione di default, è anche l'unica supportata per applicazioni Python. Ciò comporta che il driver è in esecuzione sul container Client. In questo specifico deployment ciò non rappresenta un problema, in quanto *"A common deployment strategy is to submit your application from a gateway machine that is physically co-located with your worker machines (e.g. Master node in a standalone EC2 cluster). In this setup, client mode is appropriate."* [1]
- È presente uno Spark History Server per la consultazione delle Web UI delle singole applicazioni di Spark (quelle la cui porta di default è la `4040`). Il History Server è necessario poiché al completamento di `spark-submit` le Web UI vengono chiuse.
  - I log di tutti i container di Spark sono condivisi attraverso un Named Volume di Docker.
  - La Web UI delle applicazioni permette di consultare i DAG e statistiche come i tempi di esecuzione di ciascuno stage.
- Le dipendenze delle librerie di Python richieste sono risolte installandole nel momento della creazione dell'immagine utilizzata da tutti i container di Spark.
- Le dipendenze dal codice sorgente sono risolte a runtime utilizzando l'opzione `--py-files` e indicando il file `.zip` della cartella contenente i sorgenti. [2]

Per **InfluxDB**:

- InfluxDB è utilizzato in modalità standalone.
- È stata abilitata la persistenza dei dati tramite Named Volume di Docker.

Per **Grafana**:

- Grafana è utilizzato in modalità standalone.
- È stata abilitata la persistenza della dashboard tramite Named Volume di Docker.

IV. Data ingestion

È possibile riassumere la fase di data ingestion in 2 momenti distinti:

1) Acquisizione dei dati in locale.
2) Caricamento dei dati in HDFS.

La fase di acquisizione dei dati in locale va eseguita precedentemente alla creazione del cluster di container. È costituita dai seguenti passi, implementati tramite combinazione di script bash e Python:

1) Vengono prodotti gli URL dei dataset in formato CSV da scaricare.
   - Gli URL seguono un formato fisso e sono semplici da generare.
   - In questo step è possibile decidere se utilizzare il dataset con granularità spaziale del paese o delle regioni.
   - La granularità temporale è sempre quella dell'ora.
2) I CSV relativi agli URL vengono scaricati nella directory `dataset/raw`.
3) I CSV vengono combinati in un unico CSV per paese e salvati nella directory `dataset/combined`
   - L'header del file viene rimosso. La scelta avvantaggia l'uso dell'API degli RDD, che non richiede il filtraggio della prima riga, ma svantaggia l'uso dell'API dei DataFrame e SQL che richiedono uno schema delle colonne esplicito.
   - La scelta di combinare in un file per paese è stata presa in quanto la query Q_2 richiede solo i dati relativi al dataset dell'Italia. In questo modo è possibile ridurre alla sorgente la quantità di dati che Spark deve leggere da HDFS.

4) I CSV vengono convertiti e salvati anche in formato Avro e Parquet nella directory `dataset/combined`.
   - La conversione è effettuata tramite Spark.
   - Nella versione di Spark usata, la compatibilità con Avro richiede una dipendenza esterna, scaricabile in automatico da Spark o manualmente inserita nell'immagine del container e riferita nella creazione della Spark Session.

All'avvio del container NameNode di HDFS, questo ha accesso tramite Bind Volume al contenuto della directory `dataset`, che può caricare tramite script in HDFS.

- Successivamente al caricamento in HDFS, il NameNode imposta la ACL per l'utente `spark` in modo che abbia permessi di lettura e scrittura.

V. Processamento

Sia la query Q_1 sia la query Q_2 sono state implementate utilizzando le API degli RDD, dei DataFrame e di SparkSQL.

L'implementazione tramite API degli RDD può utilizzare in input solo i dati in formato CSV, mentre l'implementazione tramite API dei DataFrame e di SparkSQL è in grado di utilizzare anche i dati in formato Avro e Parquet.

A. Query 1

La query Q_1 ritorna un solo RDD/DataFrame. Di seguito vengono riassunte le implementazioni della query Q_1 con le diverse API.

RDD In figura Fig.2 è riportato lo schema completo delle trasformazioni necessarie per produrre un RDD contenente l'output finale, che può essere riassunto così:

1) Vengono prodotti due RDD a partire dai CSV di Italia e Svezia, che vengono poi uniti in unico RDD.
2) Il RDD viene preparato per il processamento portandolo ad una forma di tipo chiave-valore, dove la chiave è la coppia `(paese, anno)` e i valori sono `(carbon-intensity, cfe, 1)`. Il valore `1` predispone il RDD per il pattern del calcolo della media.
3) Media, minimo e massimo vengono calcolati parallelamente tramite tre diverse trasformazioni di `reduceByKey`.
4) Viene fatto il join degli RDD ottenuti, seguito da una `map` necessaria a fare il "flatten" di nuovo in un formato chiave-valore.
5) Viene fatto il sorting degli RDD in base alla chiave, e infine una `map` per portare l'output al formato richiesto dalla specifica.

DataFrame In figura Fig.3 è riportato lo schema completo delle operazioni necessarie per produrre un DataFrame contenente l'output finale, che può essere riassunto così:

1) Vengono prodotti due DataFrame a partire dai file di Italia e Svezia, che vengono poi uniti in un singolo DataFrame. È possibile scegliere il formato del file di input tra CSV, Avro e Parquet.
2) Viene aggiunta la colonna relativa all'anno, estraendo l'informazione dalla colonna che contiene il datetime e vengono selezionate solo le colonne utili alla computazione della query.
3) I dati vengono aggregati per la coppia `(paese, anno)` utilizzando le funzioni built-in di Spark per media, minimo e massimo e rinominando opportunamente le colonne.
4) I dati vengono ordinati e portati nel formato richiesto.

SparkSQL In figura Fig.4 è riportato lo schema completo per produrre un DataFrame contenente l'output finale, che può essere riassunto così:

1) Vengono prodotti due DataFrame a partire dai file di Italia e Svezia, che vengono poi uniti in un singolo DataFrame. È possibile scegliere il formato del file di input tra CSV, Avro e Parquet.
2) In un'unica query SQL vengono prodotti i dati, aggregandoli per la coppia `(paese, anno)`, ordinati e portati nel formato richiesto dalla specifica.

B. Query 2

La query Q_2 ritorna 5 RDD/DataFrame. Di seguito vengono riassunte le implementazioni della query Q_2 con le diverse API.

RDD In figura Fig.5 è riportato lo schema completo delle trasformazioni necessarie per produrre un RDD contenente l'output finale, che può essere riassunto così:

1) Vengono prodotti un RDD a partire dal CSV di Italia.
2) Il RDD viene preparato per il processamento portandolo ad una forma di tipo chiave-valore, dove la chiave è la coppia `(anno, mese)` e i valori sono `(carbon-intensity, cfe, 1)`. Il valore `1` predispone il RDD per il pattern del calcolo della media.
3) La media viene calcolata tramite `reduceByKey`.
4) Qui il flusso si separa in 3: nel 1º i dati vengono ordinati per `datetime`, nel 2º i dati vengono ordinati per `carbon-intensity` decrescente e nel 3º per `cfe` decrescente. Il 1º flusso è già pronto come valore di ritorno, mentre sul 2º e 3º sono eseguite indipendentemente le seguenti operazioni.
5) Viene usata l'operazione `zipWithIndex()`. [3] Questa operazione associa ad ogni elemento un indice che corrisponde alla sua posizione nel RDD in base all'attuale configurazione delle partizioni. Se la `zipWithIndex()` è preceduta da una operazione di sorting, questo indice corrisponde alla posizione nel RDD ordinato, anche in presenza di più partizioni, come mostrato in Listing.1.
   - L'approccio con `.zipWithIndex()` permette di prendere i primi e gli ultimi n elementi di un RDD senza dover utilizzare una azione e convogliare i dati sul driver. Però, come riportato nella documentazione, in caso di più partizioni, *"needs to trigger a spark job"*. [4]

- Un approccio alternativo è quello di utilizzare `.takeOrdered(...)`, che essendo una azione ritorna i dati direttamente al driver, e successivamente `.parallelize()` in caso volessimo i dati come RDD.
- L'approccio con `.zipWithIndex()` permette di evitare un secondo costoso sorting crescente su tutti i dati per ottenere gli ultimi n dati.
- L'approccio con `.takeOrdered(...)` potrebbe essere vantaggioso in deployment come quello implementato nel progetto, in quanto driver e Spark Cluster sono vicini e la quantità di dati è molto ridotta.

6) Qui il flusso si separa nuovamente: nel flusso `top` vengono presi i primi n elementi e ne viene poi rimosso l'indice, mentre nel flusso `bottom` vengono presi gli ultimi n elementi, ne viene rimosso l'indice, e vengono ordinati in maniera crescente.

DataFrame

In figura Fig.6 è riportato lo schema completo delle operazioni necessarie per produrre un DataFrame contenente l'output finale, che può essere riassunto così:

1) Viene prodotto un DataFrame a partire dal file dell'Italia. È possibile scegliere il formato del file di input tra CSV, Avro e Parquet.
2) Vengono aggiunte le colonne relative all'anno e al mese, estraendo l'informazione dalla colonna che contiene il datetime e vengono selezionate solo le colonne utili alla computazione della query.
3) I dati vengono aggregati per la coppia `(anno, mese)` utilizzando le funzioni built-in di Spark per media e rinominando opportunamente la colonna.
4) I dati vengono portati nel formato richiesto.
5) I dati vengono poi ordinati tramite `orderBy` e opportunamente limitati con `limit` per produrre gli output richiesti.
   - Non è possibile fare la stessa ottimizzazione dell'implementazione con l'API degli RDD, vista la mancanza dell'operazione `zipWithIndex()` per l'API dei DataFrame.

SparkSQL

In figura Fig.7 è riportato lo schema completo per produrre i DataFrame contenenti gli output finali, che può essere riassunto così:

1) Viene prodotto un DataFrame a partire dal file dell'Italia. È possibile scegliere il formato del file di input tra CSV, Avro e Parquet.
2) Una prima query SQL produce i dati, aggregandoli per la coppia `(anno, mese)` e li porta nel formato richiesto dalla specifica.
3) 5 query SQL producono i DataFrame contenente gli output richiesti, ordinando e limitando opportunamente.

C. Salvataggio dei risultati

Il salvataggio dell'output delle query ha le seguenti caratteristiche:

- È possibile indicare al momento dell'avvio dell'applicazione se utilizzare una operazione di `.coalesce(1)` prima della scrittura per avere il risultato in un singolo file.
- Tutti i dati prodotti dalle query vengono salvati in HDFS nella directory `results/{query}/{api}-{formato di lettura}-{coalesce o no}`.
- Solo i dati necessari alla produzione grafici vengono salvati in InfluxDB.

D. Grafici

I grafici sono organizzati in una Dashboard Grafana, e sono riportati in Fig.8, Fig.9, Fig.10 e Fig.11.

VI. Uso del sistema

A. Opzioni generali

All'avvio dell'applicazione, l'utente ha la possibilità di scegliere:

- Il tipo di modalità tra `local` o `composed`. Questo impatta i path dei file di input e di output, l'utilizzo del FS locale o di HDFS e la scelta tra il deployment locale o Standalone Cluster per Spark.
- Quale query eseguire.
- Con quale API eseguire la query.
- Quale formato di input utilizzare.
- Se utilizzare o meno la cache durante le query.
- Se salvare o meno su File System.
- Se utilizzare o meno la `coalesce` prima della scrittura su File System.
- Se salvare o meno su InfluxDB.
- Se misurare e salvare i tempi di esecuzioni dei vari step dell'applicazione.
- Se stampare o meno informazioni di debug durante l'esecuzione.

VII. Esperimenti

A. Introduzione

Gli esperimenti condotti sono mirati a valutare l'impatto sul tempo di processamento delle 2 query di:

1) Le differenti API: RDD, DataFrame e SparkSQL.
2) I differenti formati: CSV, Avro[1] e Parquet[1].
3) L'uso del caching[2].

[1]: solo DataFrame e SparkSQL.
[2]: solo RDD e DataFrame.

Per ogni esperimento è stato condotto anche uno studio dell'impatto della variazione della grandezza del dataset[3]. Questo porta il totale a 6 esperimenti condotti.

[3]: il dataset più piccolo ha granularità spaziale del paese (35064 entry per Italia e Svezia) , mentre quello più grande ha granularità spaziale della regione (210384 entry per Italia,

140256 entry per Svezia). Il dataset per la Q_1 aumenta di 5 volte, mentre quello per la Q_2 aumenta di 6 volte.

I tempi di processamento sono:

- Raccolti direttamente dentro il driver, utilizzando le funzioni built-in di Python.
- Calcolati da prima dell'inizio della query fino a dopo una `.collect()` per assicurare che il processamento venga effettivamente compiuto.
- Salvati e analizzati direttamente in InfluxDB tramite query in Flux, come quella in Listing.2.

B. Risultati

Esperimento 1: API

Nell'esperimento 1.a, vengono confrontati i tempi di processamento ottenuti:

- Variando l'API tra: RDD, DataFrame, SparkSQL e una baseline costituita da una implementazione in Python nativo.
- Utilizzando il formato di lettura a CSV.
- Non utilizzando il caching.
- Utilizzando il dataset con granularità per paese.

Query	**API**	Run	Media in s	StdDev
1	baseline	10	0.772	0.6
1	rdd	10	7.398	0.421
1	sql	10	12.607	0.471
1	df	10	13.948	0.765
2	baseline	10	0.303	0.034
2	rdd	10	7.497	0.2
2	sql	10	13.372	0.431
2	df	10	13.713	0.843

TABLE I: Risultati dell'esperimento 1.a

Dai risultati ottenuti si può dedurre che:

- La baseline è notevolmente più veloce di qualsiasi implementazione con Spark. Ciò probabilmente è dovuto ai ritardi introdotti da Docker e WSL. *Si lascia come sviluppo futuro quello di rivalutare questo risultato su una piattaforma e un deployment diverso.*
- Per entrambe le query l'implementazione con l'API degli RDD risulta la più veloce.
- Per entrambe le query l'implementazione con l'API di SparkSQL risulta leggermente più veloce di quella con l'API dei DataFrame.

Nell'esperimento 1.b, vengono confrontati i tempi di processamento ottenuti:

- Variando l'API tra: RDD, DataFrame, SparkSQL e baseline.
- Utilizzando il formato di lettura CSV.
- Utilizzando il caching, dove possibile.
- Variando il dataset tra: con granularità per paese e con granularità per regione.

Query	**API**	**Dataset**	Run	Media in s	StdDev
1	baseline	country	10	0.772	0.6
1	baseline	region	10	3.294	0.812
1	rdd	country	10	7.262	0.568
1	rdd	region	10	7.654	0.295
1	sql	country	10	12.607	0.471
1	sql	region	10	14.394	0.818
1	df	country	10	13.535	0.795
1	df	region	10	15.408	0.888
2	baseline	country	10	0.303	0.034
2	baseline	region	10	1.523	0.064
2	rdd	country	10	7.348	0.132
2	rdd	region	10	7.797	0.475
2	sql	country	10	13.372	0.431
2	sql	region	10	15.709	0.65
2	df	country	10	17.448	0.549
2	df	region	10	19.603	0.805

TABLE II: Risultati dell'esperimento 1.b

Dai risultati ottenuti si può dedurre che:

- Come atteso, la baseline è estremamente più suscettibile alla grandezza del dataset, aumentando proporzionalmente con costante proporzionalità quasi pari a 1.
- Per entrambe le query, l'incremento assoluto del tempo di processamento delle implementazioni con l'API degli RDD è minore di quello della baseline.
- Per la Q_1, l'incremento assoluto del tempo di processamento per **tutte** implementazioni con l'API degli RDD è minore di quello della baseline.

Esperimento 2: Formato

Nell'esperimento 2.a, vengono confrontati i tempi di processamento ottenuti:

- Variando l'API tra: DataFrame e SparkSQL.
- Variando il formato di lettura tra: CSV, Avro e Parquet.
- Non utilizzando il caching.
- Utilizzando il dataset con granularità per paese.

Query	API	**Formato**	Run	Media in s	StdDev
1	df	avro	10	9.855	0.409
1	df	parquet	10	10.977	0.396
1	df	csv	10	13.948	0.765
1	sql	avro	10	9	0.181
1	sql	parquet	10	9.708	0.437
1	sql	csv	10	12.607	0.471
2	df	avro	10	11.54	1.091
2	df	parquet	10	11.614	0.69
2	df	csv	10	13.713	0.843
2	sql	parquet	10	11.565	0.848
2	sql	avro	10	11.875	0.441
2	sql	csv	10	13.372	0.431

TABLE III: Risultati dell'esperimento 2.a

Dai risultati ottenuti si può dedurre che:

- In tutte le query e le implementazioni, i risultati ottenuti con Avro o Parquet sono migliori di quelli ottenuti con CSV.
- In 3 combinazioni su 4 Avro è leggermente migliore di Parquet.

Nell'esperimento 2.b, vengono confrontati i tempi di processamento ottenuti:

- Utilizzando l'API SparkSQL.

- Variando il formato di lettura tra: CSV, Avro e Parquet.
- Non utilizzando il caching.
- Variando il dataset tra: con granularità per paese e con granularità per regione.

Query	**Formato**	**Dataset**	Run	Media in s	StdDev
1	csv	country	10	12.607	0.471
1	csv	region	10	14.394	0.818
1	avro	country	10	9	0.181
1	avro	region	10	10.203	0.51
1	parquet	country	10	9.708	0.437
1	parquet	region	10	10.041	0.487
2	csv	country	10	13.372	0.431
2	csv	region	10	15.709	0.65
2	avro	country	10	11.875	0.441
2	avro	region	10	12.186	0.596
2	parquet	country	10	11.565	0.848
2	parquet	region	10	12.626	0.965

TABLE IV: Risultati dell'esperimento 2.b

Dai risultati ottenuti si può dedurre che:

- In entrambe le query, CSV è il formato più suscettibile alla variazione della grandezza del dataset.

Esperimento 3: Caching

Nell'esperimento 3.a, vengono confrontati i tempi di processamento ottenuti:

- Variando l'API tra: RDD e DataFrame[4].
- Utilizzando il formato di lettura CSV.
- Variando tra: utilizzo del caching e non utilizzo del caching.
- Utilizzando il dataset con granularità per paese.

[4]: L'implementazione di Q_1 con API dei DataFrame non usa l'operazione di `.cache()`.

Query	API	**Caching**	Run	Media in s	StdDev
1	rdd	TRUE	10	7.262	0.568
1	rdd	FALSE	10	7.398	0.421
2	df	FALSE	10	13.713	0.843
2	df	TRUE	10	17.448	0.549
2	rdd	TRUE	10	7.348	0.132
2	rdd	FALSE	10	7.497	0.2

TABLE V: Risultati dell'esperimento 3.a

Dai risultati ottenuti si può dedurre che:

- L'utilizzo del caching ha un impatto diverso per le due query.
- Quando l'utilizzo del caching ha un impatto negativo questo è maggiore rispetto a quando l'utilizzo del caching ha un impatto positivo.

Nell'esperimento 3.b, vengono confrontate le differenze di tempi di processamento ottenute:

- Utilizzando l'API degli RDD.
- Utilizzando il formato di lettura CSV.
- Variando tra: utilizzo del caching e non utilizzo del caching.
- Variando il dataset tra: con granularità per paese e con granularità per regione.

Query	**Caching**	**Dataset**	Run	Media in s	StdDev
1	FALSE	country	10	7.398	0.421
1	FALSE	region	10	8.12	0.253
1	TRUE	country	10	7.262	0.568
1	TRUE	region	10	7.654	0.295
2	FALSE	country	10	7.497	0.2
2	FALSE	region	10	7.844	0.4
2	TRUE	country	10	7.348	0.132
2	TRUE	region	10	7.797	0.475

TABLE VI: Risultati dell'esperimento 3.b

Dai risultati ottenuti si può dedurre che:

- L'utilizzo del caching ha un impatto diverso per le due query.

REFERENCES

[1] "Launching applications with spark-submit." (), [Online]. Available: https : / / spark . apache . org / docs / latest / submitting - applications . html # launching - applications - with-spark-submit.

[2] "Bundling your application's dependencies." (), [Online]. Available: https : / / spark . apache . org / docs / latest / submitting - applications . html # bundling - your - applications-dependencies.

[3] "Apache spark — rdd zipwithindex." (), [Online]. Available: https://bigdataenthusiast.medium.com/spark-scala-rdd-zipwithindex-d08626333032.

[4] "Pyspark.rdd.zipwithindex." (), [Online]. Available: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.zipWithIndex.html.

VIII. CODICI E IMMAGINI

Di seguito il codice Python che mostra il funzionamento della `zipWithIndex()` in relazione alla `sortBy(...)`.

```
>>> base = sc.parallelize([
        90, 70 ,50, 30, 10, 0, 20, 40, 60, 80], 5
        )
>>> base.glom().collect()
[[90, 70], [50, 30], [10, 0], [20, 40], [60, 80]]

>>> base.zipWithIndex().glom().collect()
[[(90, 0), (70, 1)], [(50, 2), (30, 3)], [(10, 4), (0, 5)], [(20, 6), (40, 7)], [(60, 8), (80, 9)]]

>>> base.sortBy(lambda x: x).glom().collect()
[[0, 10, 20], [30, 40], [50, 60], [70, 80], [90]]

>>> zipped_and_sorted = base \
        .sortBy(lambda x: x) \
        .zipWithIndex()
>>> zipped_and_sorted.glom().collect()
[[(0, 0), (10, 1), (20, 2)], [(30, 3), (40, 4)], [(50, 5), (60, 6)], [(70, 7), (80, 8)], [(90, 9)]]

>>> zipped_and_sorted \
        .filter(lambda x: x[1] >= 5) \
        .map(lambda x: x[0]) \
        .collect()
[50, 60, 70, 80, 90]
```

Listing 1: Codice Python che mostra funzionamento di `zipWithIndex`.

Di seguito il codice Flux che mostra la query per l'esperimento 2.a.
I risultati dei vari esperimenti vengono prodotti cambiando `groupCols` e `sortGroupCols`.

```
import "array"

groupCols = ["_measurement", "api", "format"]
keepCols = groupCols |> array.concat(v: ["mean", "stddev", "count"])
sortGroupCols = ["_measurement", "api"]

base = from(bucket: "mybucket")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r.api == "df" or r.api == "sql")
  |> filter(fn: (r) => r.cache != "True")
  |> filter(fn: (r) => r.custom == "country")

meanData = base
  |> group(columns: groupCols)
  |> aggregateWindow(every: 1y, fn: mean, createEmpty: false)

stdDevData = base
  |> group(columns: groupCols)
  |> aggregateWindow(every: 1y, fn: stddev, createEmpty: false)

meanAndStdDevData = join(
    tables: {t1: meanData, t2: stdDevData},
    on: groupCols,
    method: "inner"
)
  |> rename(columns: {_value_t1: "mean", _value_t2: "stddev"})
  |> keep(columns: keepCols)

countData = base
  |> group(columns: groupCols)
  |> aggregateWindow(every: 1y, fn: count, createEmpty: false)
  |> rename(columns: {_value: "count"})
  |> keep(columns: keepCols)

finalData = join(
    tables: {t1: meanAndStdDevData, t2: countData},
    on: groupCols,
    method: "inner"
)
  |> group(columns: sortGroupCols)
  |> sort(columns: ["mean"], desc: false)
  |> yield()
```

Listing 2: Codice Flux della query dell'esperimento 2.

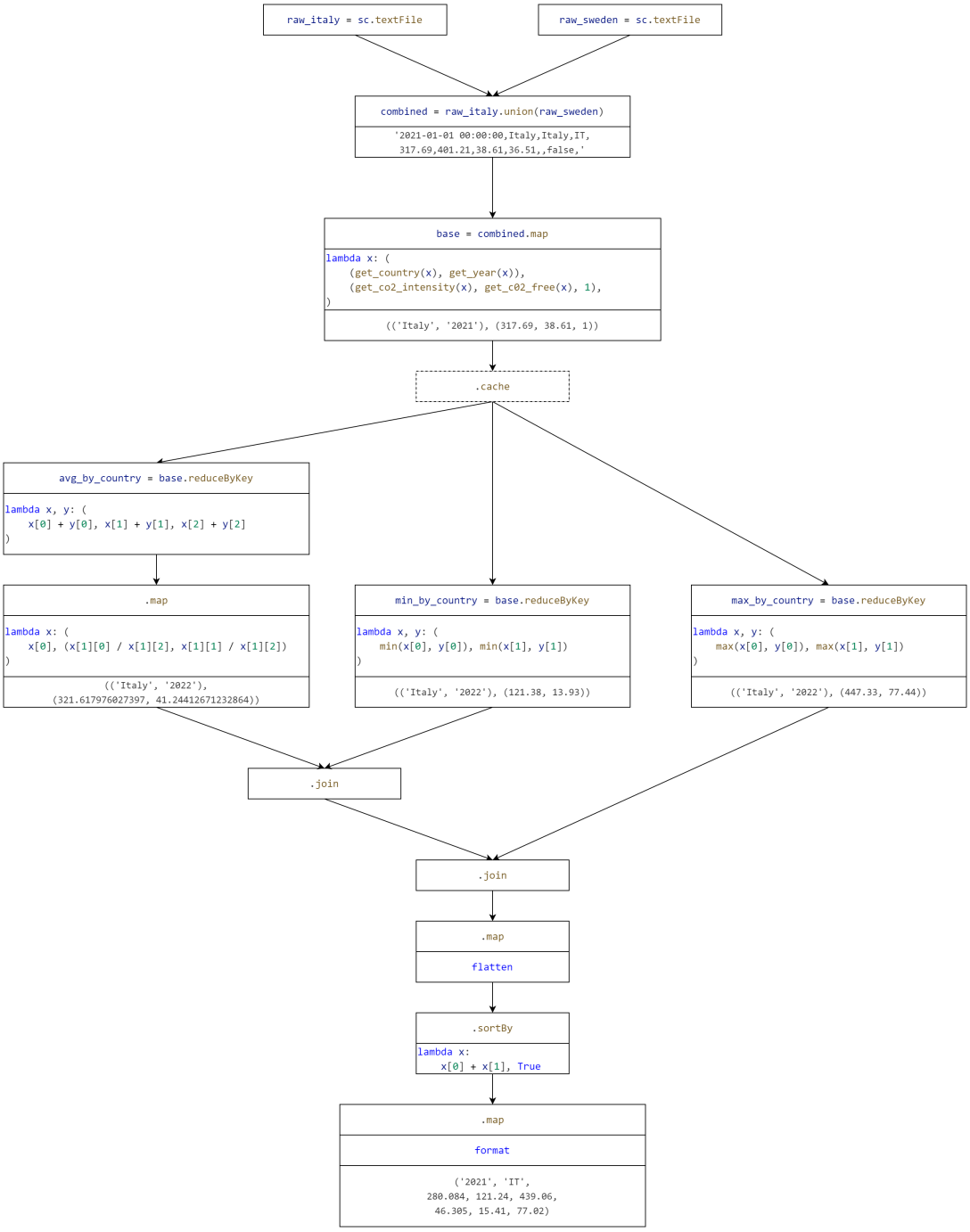

Fig. 2: Implementazione di Q_1 tramite RDD.

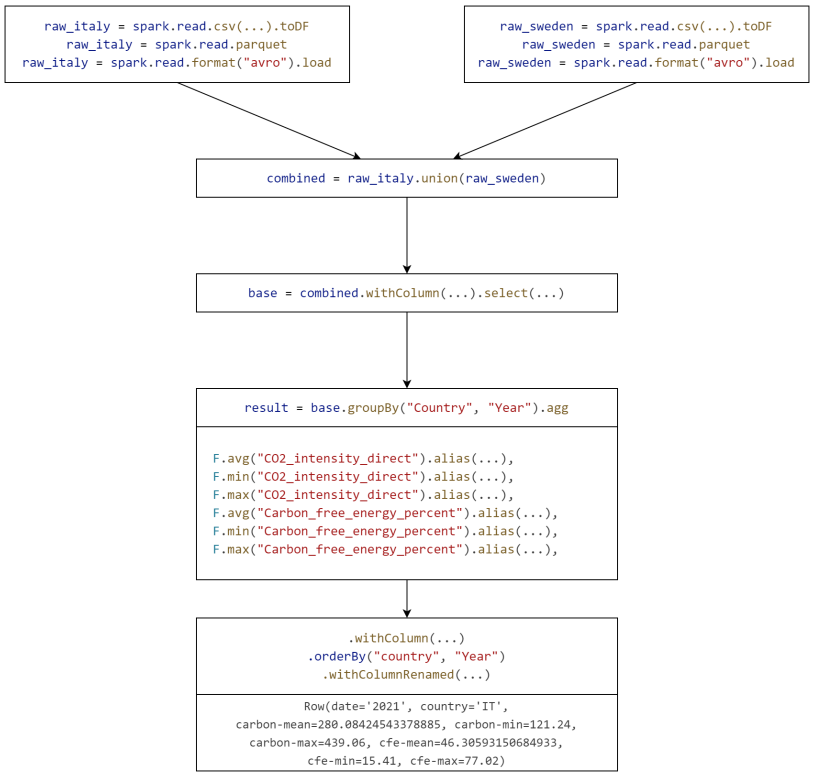

Fig. 3: Implementazione di Q_1 tramite DataFrame.

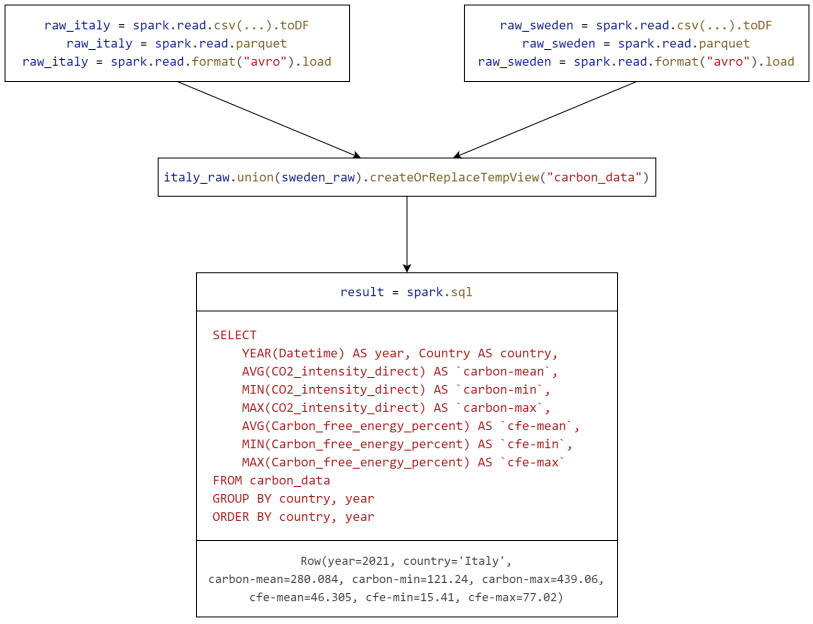

Fig. 4: Implementazione di Q_1 tramite SparkSQL.

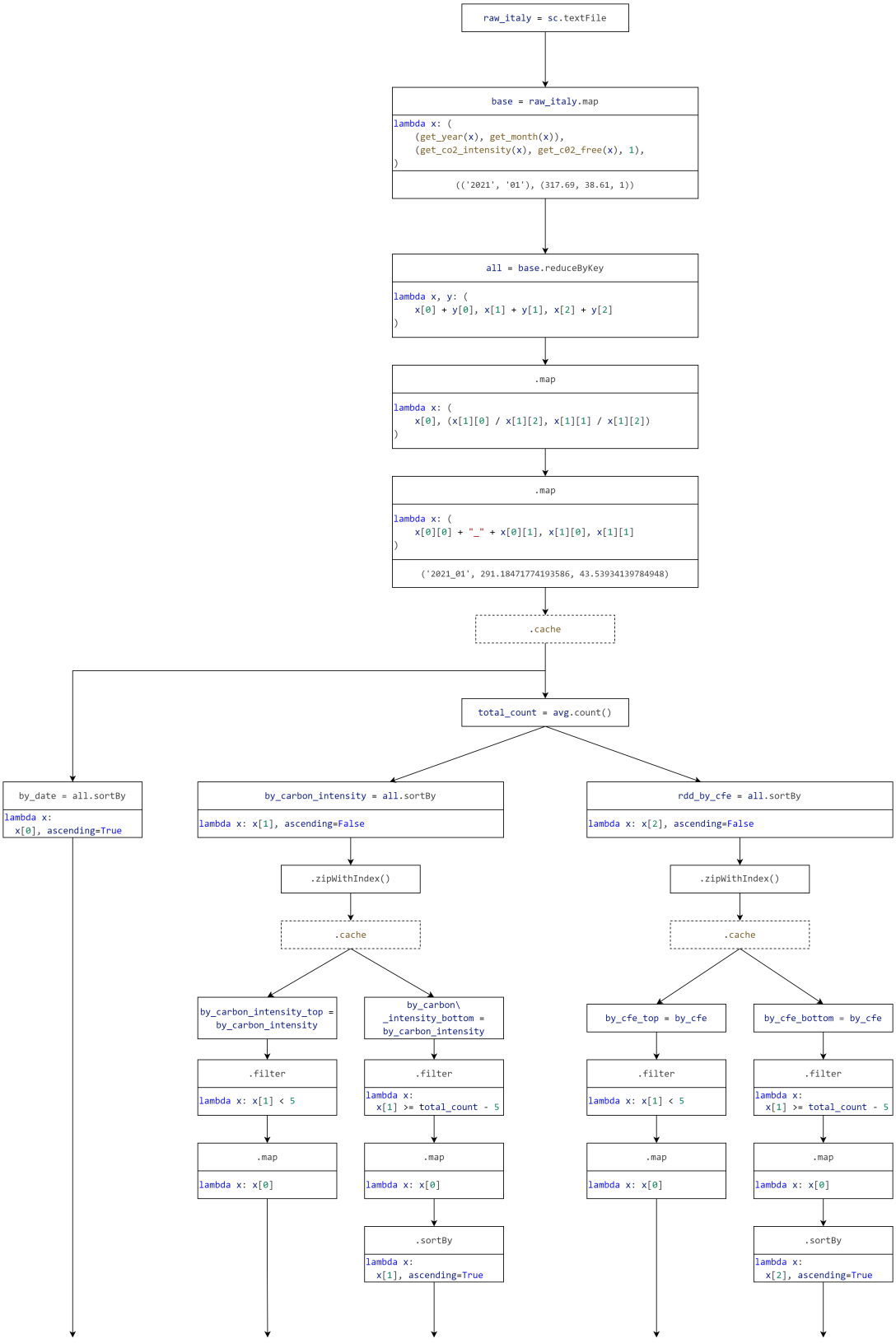

Fig. 5: Implementazione di Q_2 tramite RDD.

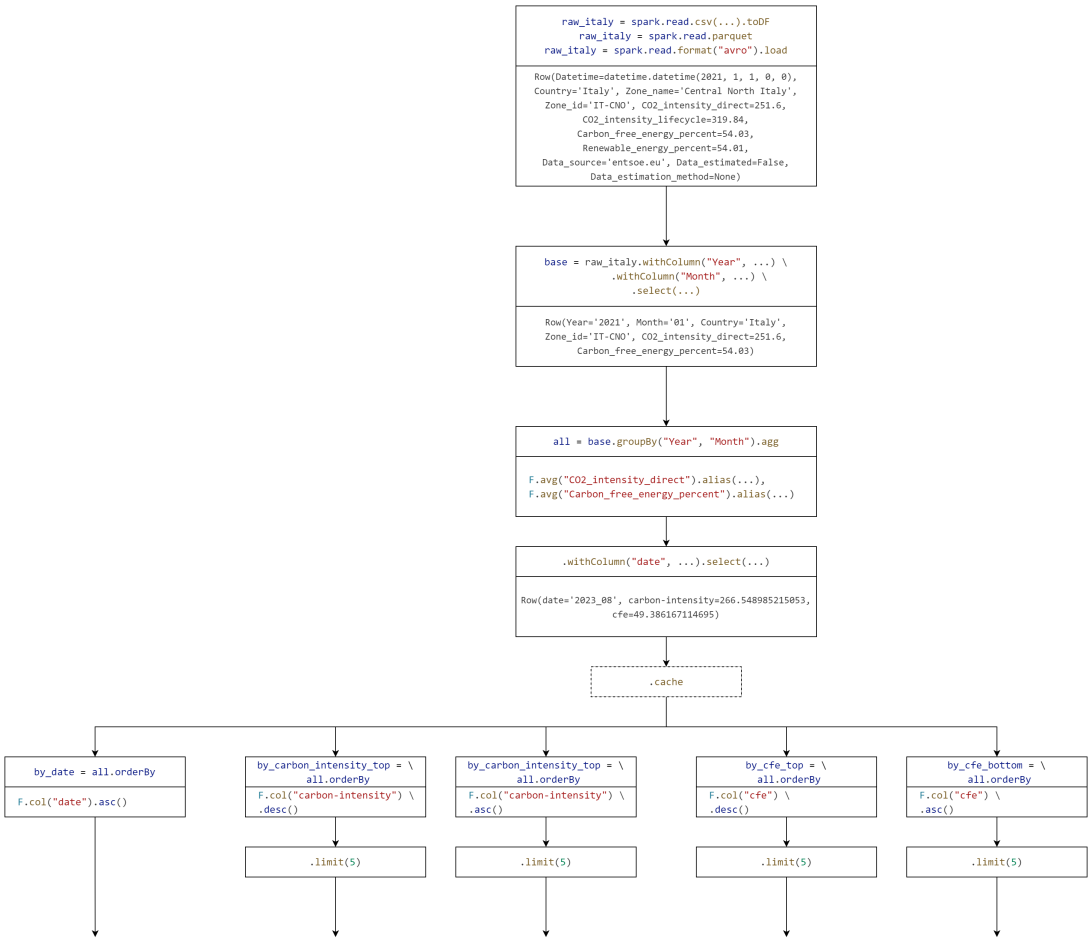

Fig. 6: Implementazione di Q_2 tramite DataFrame.

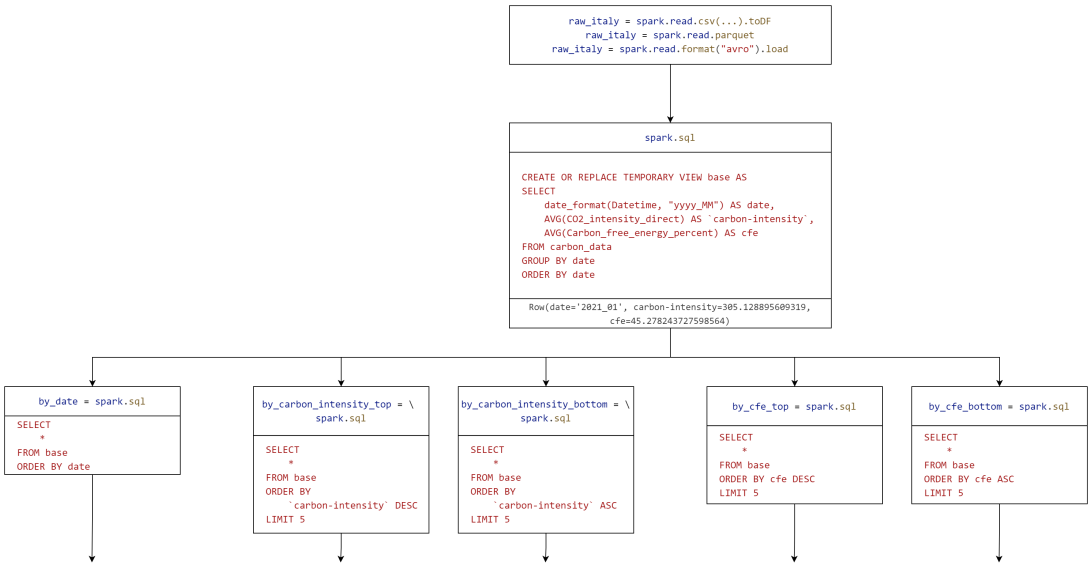

Fig. 7: Implementazione di Q_2 tramite SparkSQL.

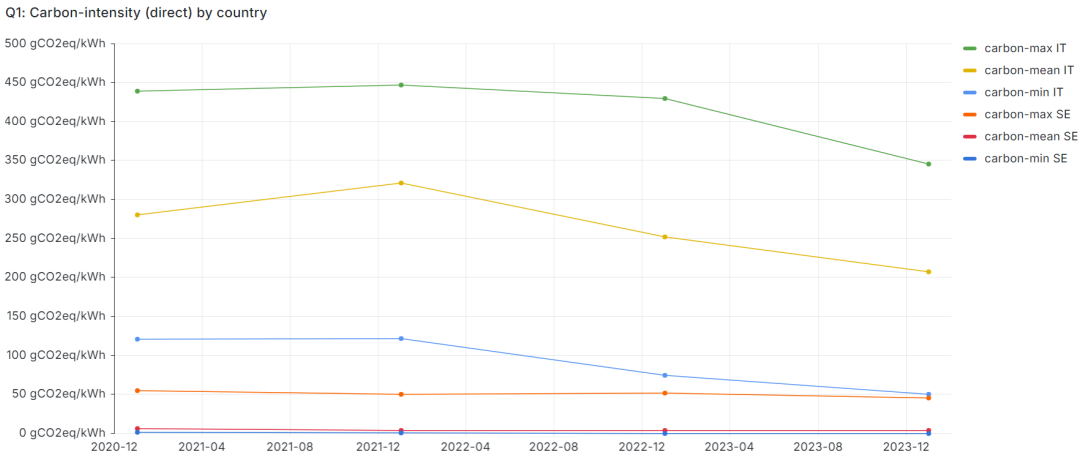

Fig. 8: Grafico di risultati Q_1 per `carbon-intensity`.

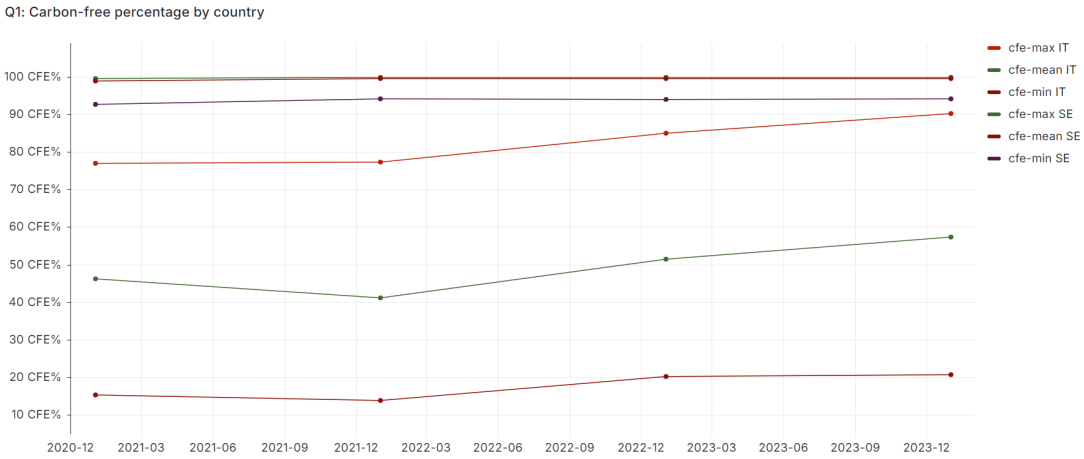

Fig. 9: Grafico di risultati Q_1 per `cfe`.

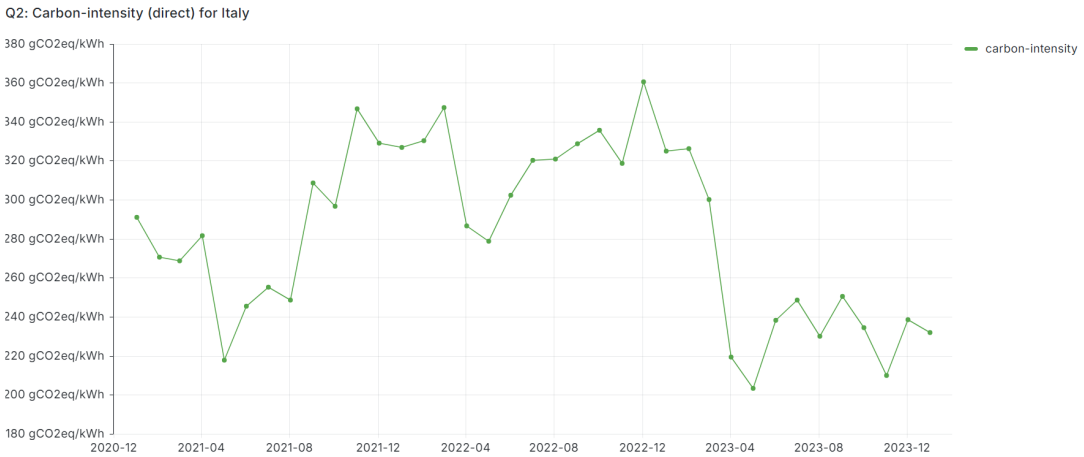

Fig. 10: Grafico di risultati Q_2 per `carbon-intensity`.

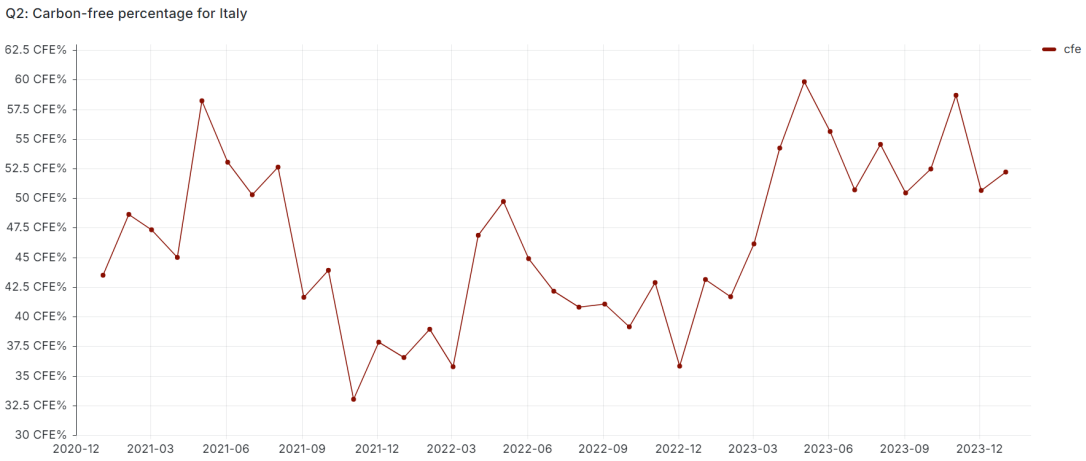

Fig. 11: Grafico di risultati Q_2 per `cfe`.