Esempi di Apache Spark

Programmazione

Apache Spark è un framework open-source per l’elaborazione distribuita dei dati, progettato per gestire in modo estremamente rapido grandi volumi di informazioni su cluster di macchine.

Nasce come evoluzione del modello MapReduce di Hadoop, migliorandone drasticamente le prestazioni grazie all’uso massiccio della memoria RAM: invece di scrivere continuamente su disco tra un’operazione e l’altra, Spark conserva i dati in memoria, riducendo la latenza e rendendo possibile l’analisi interattiva, il machine learning e lo streaming in tempo reale.

La sua architettura si basa su un driver che coordina il lavoro e su più executor che processano i dati in parallelo, distribuendo automaticamente il carico e gestendo la tolleranza ai guasti.

0. SparkSession + DataFrame

La SparkSession è l’interfaccia principale per lavorare con Apache Spark: rappresenta il punto d’ingresso per creare DataFrame, leggere file, configurare l’applicazione e utilizzare tutti i moduli (SQL, Streaming, MLlib).

Un DataFrame è una collezione distribuita di dati organizzati in colonne, con schema e tipi, ottimizzato tramite il motore Catalyst e progettato per gestire grandi volumi in modo parallelo e fault-tolerant.

0. Setup: creare la SparkSession e un DataFrame di esempio

```
from pyspark.sql import SparkSession

# Creazione della SparkSession (punto di ingresso di Spark)
spark = SparkSession.builder \
    .appName("EsempiSpark") \
    .getOrCreate()

# Dati di esempio in memoria (lista di tuple)
data = [("Mario", 30, "IT"),
        ("Luca", 25, "FR"),
        ("Anna", 40, "IT")]

# Creazione di un DataFrame con schema esplicito
df = spark.createDataFrame(data, ["nome", "eta", "paese"])

df.show()
df.printSchema()
```

Cosa fa:

- Crea la sessione Spark.
- Costruisce un DataFrame a partire da una lista Python.
- Mostra i dati e lo schema (tipi delle colonne).

1. Select e Filter

Le operazioni di selezione (select) e filtraggio (filter/where) permettono di estrarre specifiche colonne o righe da un DataFrame.

Sono trasformazioni lazy: Spark non le esegue subito, ma costruisce un piano logico ottimizzato che verrà calcolato solo quando servirà un’azione.

Queste operazioni sono fondamentali per pulizia dati, pre-processing e creazione di viste ridotte.

1. Selezionare colonne e filtrare righe (select, filter / where)

```
from pyspark.sql.functions import col

# Selezionare solo alcune colonne
df_nome_eta = df.select("nome", "eta")

# Filtrare le righe con eta > 30
df_over_30 = df.filter(col("eta") > 30)

df_nome_eta.show()
df_over_30.show()
```

Cosa fa:

- **select sceglie un sottoinsieme di colonne.**
- **filter (o where) applica una condizione sulle righe tramite col("eta") > 30.**

2. withColumn (nuove colonne e trasformazioni)

withColumn permette di aggiungere una nuova colonna o sovrascriverne una esistente.

È essenziale per creare feature derivate, normalizzazioni, conversioni di tipi e calcoli vettoriali.

Spark applica queste trasformazioni in modo distribuito su tutte le partizioni, senza modificare il DataFrame originale (immutabilità funzionale).

2. Aggiungere una colonna calcolata (withColumn)

Cosa fa:

- **withColumn crea (o sovrascrive) una colonna.**
- **lit crea una costante (qui +10).**
- **L’operazione è vettoriale su tutte le righe.**

```
from pyspark.sql.functions import lit

# Aggiungere una colonna "eta_tra_10_anni"
df_eta_future = df.withColumn("eta_tra_10_anni", col("eta") + lit(10))

df_eta_future.show()

```

3. GroupBy e Aggregazioni

Il groupBy permette di raggruppare i dati in base al valore di una o più colonne, mentre agg consente di applicare funzioni di aggregazione (conteggio, media, somma, min/max).

Si tratta di un'operazione chiave nell’analisi dei dati e nei sistemi distribuiti, perché richiede uno shuffle: Spark redistribuisce le righe in base ai gruppi e combina i risultati parallelamente.

3. Raggruppare e aggregare (groupBy, agg, funzioni di aggregazione)

Cosa fa:

- **groupBy("paese") crea gruppi per valore di paese.**
- **agg applica funzioni di aggregazione (avg, count).**
- **alias rinomina le colonne risultanti.**

```
from pyspark.sql.functions import avg, count

# Calcolare l'eta media per paese e quanti utenti per paese
df_stats_paese = df.groupBy("paese") \
    .agg(
        avg("eta").alias("eta_media"),
        count("*").alias("num_persone")
    )

df_stats_paese.show()
```

4. Join tra DataFrame

Le join consentono di combinare due DataFrame su una colonna comune, analogamente ai database relazionali.

Spark ottimizza automaticamente il tipo di join (broadcast, shuffle) in base alla dimensione delle tabelle.

Questo permette di collegare dataset diversi, come tabelle anagrafiche, log, lookup mantenendo coerenza con i paradigmi SQL ma con esecuzione distribuita.

4. Join tra DataFrame

Cosa fa:

- **Crea un secondo DataFrame con la “dimensione” paesi.**
- **Esegue una join (inner) sulla colonna comune paese.**
- **Aggiunge al DataFrame originale il nome esteso del paese.**

```
# Secondo DataFrame con info sui paesi
data_paesi = [("IT", "Italia"), ("FR", "Francia")]
df_paesi = spark.createDataFrame(data_paesi, ["paese", "nome_paese"])

# Join sui paesi (inner join)
df_join = df.join(df_paesi, on="paese", how="inner")

df_join.show()
```

5. Lettura CSV e scrittura Parquet

Spark supporta la lettura e scrittura di dati in molti formati (CSV, JSON, Parquet, Avro).

La lettura da CSV è utile per ingest iniziali e dati provenienti da sistemi legacy; il Parquet, invece, è un formato colonnare compresso progettato per query analitiche ad alte prestazioni.

Spark gestisce automaticamente inferenza dello schema, partizionamento e parallelizzazione dell'I/O.

5. Lettura da file CSV e scrittura su disco

Cosa fa:

- **spark.read.csv legge un file CSV, con intestazione e schema inferito.**
- **write.parquet salva i dati in formato Parquet, più efficiente di CSV per analisi.**

```
# Lettura da CSV (schema inferito)
df_csv = spark.read \
    .option("header", "true") \
    .option("inferSchema", "true") \
    .csv("path/dati_clienti.csv")

df_csv.show(5)

# Scrittura in formato Parquet (colonnare)
df_csv.write \
    .mode("overwrite") \
    .parquet("path/output_clienti_parquet")

```

6. Cache e Persistenza

La cache permette di memorizzare in RAM il risultato di una trasformazione per evitare ricalcoli.

È fondamentale per algoritmi iterativi (machine learning, analisi ricorsive) e per workflow che usano più volte lo stesso DataFrame.

Spark conserva i dati nelle partizioni dei nodi finché servono, migliorando le performance e riducendo i costi di computazione.

6. Cache / persistenza in memoria (cache)

Cosa fa:

- **Costruisce un DataFrame filtrato e con colonna calcolata.**
- **cache() chiede a Spark di mantenerlo in memoria dopo la prima azione.**
- **Successive action (come count() e show()) saranno più veloci.**

```
# Preparare un DataFrame derivato "pesante" e metterlo in cache
df_derivato = df.filter(col("paese") == "IT") \
                .withColumn("eta_x2", col("eta") * 2) \
                .cache()   # o .persist()

# Prima azione: esegue la pipeline e mette in cache
df_derivato.count()

# Le azioni successive riusano i dati in RAM
df_derivato.show()
```

7. Structured Streaming

Structured Streaming abilita l’elaborazione di flussi di dati in tempo reale usando la stessa API dei DataFrame batch.

Il flusso è trattato come una tabella che si aggiorna continuamente man mano che arrivano nuovi eventi.

Spark gestisce checkpointing, fault-tolerance e aggiornamento incrementale, consentendo casi d’uso come conteggio di parole, ingest real-time, dashboard e pipeline di log/telemetria.

7. Esempio super base di Structured Streaming

Solo per dare l'idea del pattern, non da lanciare così com'è in produzione.

Cosa fa:

- **Legge uno stream di testo da una socket (host:port).**
- **Divide le righe in parole, le raggruppa e le conta.**
- **Stampa le word count aggiornate continuamente sulla console.**

```
from pyspark.sql.functions import explode, split

# Lettura di uno stream di testo (es. socket)
lines = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Split delle righe in parole
words = lines.select(
    explode(split(col("value"), " ")).alias("word")
)

# Conteggio parole in tempo quasi-reale
word_counts = words.groupBy("word").count()

# Scrittura su console dello stream
query = word_counts.writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

# Attesa della terminazione (bloccante)
query.awaitTermination()
```


Buon Davante a tutti