SABD - Progetto 1

Esecuzione di Query Spark su dataset “TLC Trip Record Data”

Danilo Dell’Orco
Facoltà di Ingegneria
Università di Roma Tor Vergata
Roma, Lazio, Italia
danilodellorcolp@gmail.com

Jacopo Fabi
Facoltà di Ingegneria
Università di Roma Tor Vergata
Roma, Lazio, Italia
jacopo.fabi1997@gmail.com

Michele Salvatori
Facoltà di Ingegneria
Università di Roma Tor Vergata
Roma, Lazio, Italia
michelesalvv@gmail.com

ABSTRACT

L’obiettivo del progetto è quello di analizzare i dati forniti dal Taxi and Limousine Commission (TLC) riguardo le corse effettuate dai taxi, in particolare utilizzando il dataset *“TLC Trip Record Data”* fornito in formato parquet. Si applica tramite Nifi una fase di preprocessamento e caricamento del dataset, si utilizza HDFS come storage per mantenere il dataset, mentre l’analisi e l’esecuzione delle query avviene tramite Spark. I risultati delle query vengono inseriti su MongoDB, e salvati anche in formato CSV su HDFS. Infine, i risultati delle query vengono visualizzati tramite una dashboard Grafana.

1. INTRODUZIONE

Ai fini del progetto è necessario eseguire 3 operazioni di analisi sul dataset messo a disposizione.

1. Calcolare per ogni mese la percentuale media della mancia rispetto al costo della corsa.
2. Calcolare per ogni ora la distribuzione in percentuale del numero di corse rispetto alle zone di partenza, la mancia media, la sua deviazione standard ed il metodo di pagamento più diffuso.
3. Calcolare per ogni giorno le 5 zone di destinazione più popolari, indicando per ciascuna di esse il numero medio di passeggeri, la media e la deviazione standard della tariffa pagata.

Questo documento ha lo scopo di descrivere nel dettaglio l’architettura utilizzata nel progetto ed i punti principali della sua implementazione, motivando le varia scelte progettuali effettuate.

Il documento è strutturato come segue: la sezione 2 descrive i diversi componenti/frameworks del sistema e relativi dettagli implementativi, la sezione 3 descrive la realizzazione delle query, la sezione 4 descrive il testing e l’analisi delle prestazioni, la sezione 5 conclude il documento con le considerazioni finali sul sistema proposto.

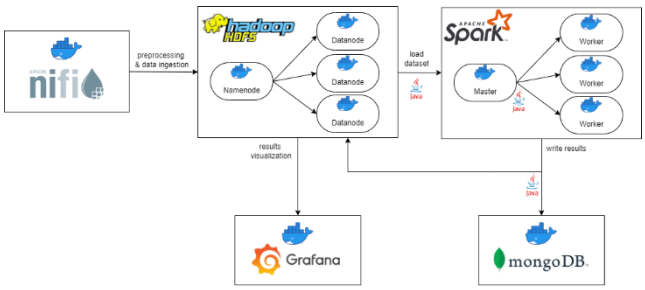


Figura 1 - Architettura di Sistema

2. ARCHITETTURA E IMPLEMENTAZIONE

Per l’analisi del dataset si utilizzano diversi framework per obiettivi differenti. L’esecuzione delle query avviene sul sistema locale, sfruttando la virtualizzazione a livello di sistema operativo offerta da **docker**. Ognuno dei framework dello stack utilizzato viene quindi hostato su uno o più container docker, e tutti questi container sono interconnessi tra loro, facendo parte della stessa rete virtuale. Si analizzano quindi nel dettaglio i componenti architetturali del sistema considerato *(figura1)*.

- Nifi (*apache/nifi:latest* [1])
- Hadoop File System (*matnar/hadoop:3.3.2* [2])
- Apache Spark (*bitnami/spark:3.2.1* [3])
- MongoDB (*mongo:latest* [4])
- Grafana (*grafana/grafana:latest* [5])

2.1. Nifi

Apache Nifi è un framework di data injection che permette di recuperare e aggregare i dati provenienti da diverse sorgenti per caricarli all'interno dei vari frameworks che formano la pipeline di processamento.

La particolarità di Nifi è quella di supportare il pre-processamento dei dati prima di effettuare l'ingestion nel sistema; questo ci permette di filtrare il dataset rimuovendo eventuali colonne inutili e dati corrotti che andrebbero

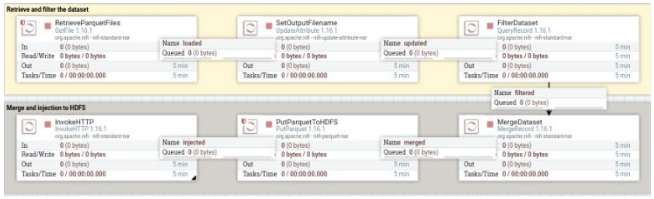


Figura 2 – NiFi Flow

soltanto ad occupare memoria durante il processamento, non essendo rilevanti nel calcolo delle diverse query.

Per NiFi si utilizza un singolo nodo su container docker, il dataset viene caricato nello storage locale del container NiFi e a seguito dell'elaborazione viene caricato su HDFS.

2.1.1. Implementazione

Il flusso di Nifi (*figura2*) progettato viene avviato e stoppato tramite le `REST API` messe a disposizione da Apache e si occupa di recuperare i tre diversi files parquet che rappresentano il dataset per filtrarli e unirli, producendo un unico file di output che viene poi salvato su `HDFS`.

I files parquet, per semplicità, sono caricati sul container che ospita il nodo NiFi, e all'avvio del flusso vengono recuperati dalla precisa directory tramite il processore `GetFile`, che produce tre diversi `flowfiles`.

Il processore `UpdateAttribute` viene utilizzato per settare la proprietà filename, che descrive il nome del file da assegnare al flusso NiFi in esecuzione.

Il processore `QueryRecord` viene utilizzato per il filtraggio dei dati sfruttando una precisa proprietà che descrive la query sql da utilizzare, nel caso specifico vengono rimosse tutte le colonne non necessarie alla realizzazione delle query e tutte le righe con valori anomali.

Il processore `MergeRecord` viene utilizzato per unire i tre diversi flowfiles generati in un singolo flowfile prima di procedere con la scrittura, così da scrivere su `HDFS` un singolo file che rappresenta il dataset finale. Per realizzare il merge, il processore è stato configurato per produrre una singola `.bin`, ovvero un oggetto che combina tutti i flowfiles in un singolo flowfile di output. Il numero minimo e massimo di records di una bin è stato impostato pari al numero totale di record del dataset dopo il filtraggio, così da rendere il processore bloccante fino a quando non sono stati uniti tutti i records in un singolo flowfile di output, evitando così di produrre più bins e quindi più flowfiles.

Il processore `PutFile` viene utilizzato per scrivere il flowfile che rappresenta il dataset finale su `HDFS` fornendo le varie configurazioni di Hadoop richieste.

Al termine della scrittura su `HDFS`, il processore `InvokeHTTP` effettua una `POST` all'indirizzo della macchina che ospita i container per notificare la fine del flusso. Il server in ascolto risponde tramite le `REST API` per stoppare il flusso e ripristinarlo per una nuova esecuzione.

Inoltre, viene eseguito lo shutdown del container di NiFi al termine dell'elaborazione, per non saturare ulteriori risorse utili per la successive fase di processamento di Spark.

2.2. Hadoop File System

Si utilizza `HDFS` per diversi scopi nell'architettura:

- Mantenere il dataset filtrato da NiFi, in modo che questo risulti visibile a tutti i nodi del sistema.
- Mantenere il `jar` dell'applicazione da fornire allo script `spark-submit` per lanciare le query Spark.
- Scrittura dei risultati di output sfruttando le API di Spark.

Si utilizza un container docker che svolge il ruolo di namenode `HDFS`, ed ulteriori tre containers che svolgono il ruolo dei datanodes `HDFS`.

2.3. MongoDB

Per il salvataggio dei risultati delle query su un datastore NoSQL si utilizza MongoDB. In particolare, per ogni query si crea una collezione dedicata, ed ogni linea dell'output del processamento viene salvata come documento nella precisa collezione.

Il server Mongo, per semplicità, è ospitato su un singolo container docker che viene raggiunto da Spark tramite il mongo-java-driver, istanziando un client a runtime.

2.4. Apache Spark

Per l'esecuzione delle query sul dataset si utilizza Spark tramite `Java API`. A seguito dell'analisi delle performance (4.1) sono stati individuati quattro nodi workers come numero ottimale, considerando anche i diversi limiti prestazionali della macchina fisica.

Si utilizza quindi un container docker per il master Spark, più ulteriori quattro containers che svolgono il ruolo di workers nell'esecuzione dei jobs.

2.4.1. Implementazione

Per l'interazione con Spark si utilizzano i driver java messi a disposizione da Apache. Si istanzia l'oggetto `SparkSession` specificando l'url del container master, e successivamente si converte il dataset da `.parquet` in `Dataset`, e poi in `JavaRDD` in modo da poter operare tramite `RDD API`.

Per ogni Query `X` si istanzia un oggetto `QueryX`, e si effettua sempre una prima trasformazione di map mappando l'intero

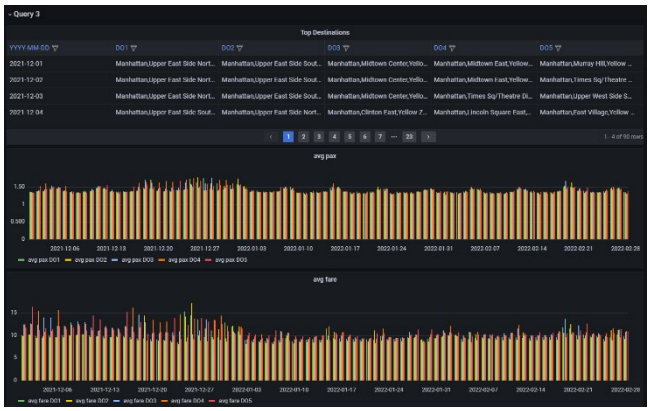


Figura 3 –Dashboard Grafana

dataset in un `JavaPairRDD<Chiave, ValQX>`. La chiave di questo RDD sarà data dal valore più idoneo alla query specifica mentre il valore è rappresentato da un oggetto `ValQX`, che contiene un campo per ogni valore richiesto nelle computazioni della `QueryX`. Utilizzando questo approccio è stato possibile effettuare un'unica fase di preprocessamento comune a tutte le query, rimuovendo le sole colonne non utilizzata da nessuna delle tre.

Per ogni query si crea una Java List `results` contenente i risultati finali, tramite l'azione di `collect()` sull'ultimo RDD prodotto.

2.4.2. Salvataggio dei Risultati

I risultati vengono salvati in parallelo su `HDFS`, `MongoDb` e `CSV` sul File System locale. I dati HDFS vengono salvati tramite l'azione `saveAsTextFile()` sull'RDD di output prodotto.

Per il salvataggio su `MongoDB` si scorre la lista `results`, e si genera un documento per ogni linea dei risultati; ogni campo dell'output corrisponde ad un particolare campo del documento mongo, mentre la chiave viene assegnata in modo automatico dallo storage. Successivamente si inserisce il documento generato nell'apposita collezione tramite `insertOne()`.

Per il salvataggio dei risultati su CSV si scorre la lista `results`, e si genera una colonna per ogni campo di interesse in output. Si scrive quindi il file CSV nella cartella `Results/`, sfruttando la classe `FileWriter`.

2.5. Grafana

Tramite un immagine docker di Grafana, viene offerta una visualizzazione dei dati tramite una semplice Dashboard, accessibile tramite: http://localhost:3001/d/QVfEthCnz/sabd-1?orgId=1 (*figura3*).

La dashboard riporta una rappresentazione in Bar Chart dei risultati delle query, sfruttando come Data Source i file csv prodotti in output da Spark, in modo da fornire all'utente una più semplice interpretazione dei risultati.

Tuttavia, come spiegato nelle *Limitazioni* (5.2), Grafana non offre un plugin gratuito per realizzare le sorgenti di dati a partire da un cluster HDFS o MongoDB. Pertanto, il fetching dei risultati è stato simulato caricando in locale nel container di Grafana i file csv risultanti dalla computazione di Spark.

3. QUERY

Vediamo le scelte effettuate durante la progettazione delle varie query.

3.1. Query 1

Per ogni mese solare, calcolare la percentuale media dell'importo della mancia rispetto al costo della corsa esclusi i pedaggi. Calcolare il costo della corsa come differenza tra l'importo totale (Total amount) e l'importo dei pedaggi (Tolls amount) ed includere soltanto i pagamenti effettuati con carta di credito.

3.1.1. SparkRDD

La query è calcolata su base giornaliera, è quindi necessario il campo `tpep_dropoff_datetime` che rappresenta il timestamp relativo all'istante di arrivo del taxi, e quindi la fine di una corsa. Inoltre, per il calcolo della percentuale media della mancia rispetto al costo della corsa esclusi i pedaggi $\frac{tip}{total-tolls}$, sono necessari i campi `tip_amount`, `total_amount` e `tolls_amount`, oltre al campo `payment_type` per includere soltanto i pagamenti effettuati con carta di credito.

A partire dall'oggetto `JavaRDD<Row>` che mantiene il dataset si produce il nuovo RDD `taxiRows`, mappando ogni oggetto `Row` in una tupla tramite la funzione `mapToPair()`: la chiave di ogni tupla è data da una stringa associata al mese di interesse, mentre il valore è un oggetto `ValQ1`, che contiene i valori di `tip_amount`, `total_amount`, `tolls_amount`, `payment_type`, oltre a `trips_number=1`, usato poi per il conteggio delle occorrenze.

Successivamente si esegue una `filter()` per mantenere solamente le tratte con pagamento effettuato tramite carta di credito, e successivamente una `reduceByKey()` per aggregare i diversi valori ricavati dalla map. Essendo la chiave il mese, i valori vengono aggregati su base mensile; per ogni elemento dell'RDD si aggregano quindi i vari `ValQ1` sommando i diversi campi di interesse. Si ricava quindi l'RDD `reduced` dove per ogni mese, è presente il totale di `tip_amount`, `total_amount`, `tolls_amount` e numero di occorrenze mensili con pagamento tramite carta di credito.

Per il calcolo della percentuale media $\frac{tips}{total-tolls}$ si esegue una nuova `mapToPair()`; ogni mese viene mappato sul valore `tip_amount/total_amount-tolls_amount` e sul numero totale di corse con pagamento in carta di credito.

Per produrre l'output finale, sull'RDD prodotto dalla map viene applicata la `sortByKey()`, così da restituire il risultato ordinato dal primo all'ultimo mese solare.

Analisi del DAG

Analizzando il DAG prodotto da spark (8.1), si vede che si hanno due stage differenti. Il secondo stage è prodotto a seguito della `reduceByKey()`, in quanto è una wide transformation, che quindi richiede lo shuffle dei dati e la comunicazione globale tra i diversi nodi di Spark.

3.1.2. SparkSQL

Dopo aver formato il `DataFrame`, selezionando dal dataset solamente i campi di interesse per questa specifica query (`tpep_dropoff_datatime`, `tip_amount`, `tolls_amount`, `total_amount`, `payment_type`), viene effettuato un raggruppamento in base al timestamp `YYYY/MM`, attraverso lo statement `GROUP BY`, e calcolato per ogni gruppo individuate la somma delle mance, la somma degli importi dei pedaggi *(tolls)*, la somma degli importi totali addebitati ai passeggeri ed il numero di viaggi avvenuti in quel determinato mese dell'anno, filtrando i pagamenti avvenuti con carta di credito. È possibile realizzare tutto ciò in una singola query grazie alla funzione SQL `sum()`.

Successivamente, sul `Dataset` appena restituito, viene effettuata un'ultima query per il calcolo della percentuale media della mancia, rispetto al costo della corsa, tramite la formula $\frac{tips}{total\ amount - tolls}$. Infine, tramite lo statement `ORDER BY`, viene riportato l'output in ordine temporale crescente.

3.2. Query 2

Per ogni ora, calcolare la distribuzione in percentuale del numero di corse rispetto alle zone di partenza, la mancia media e la sua deviazione standard, il metodo di pagamento più diffuso.

3.2.1. SparkRDD

La query è calcolata su base oraria, quindi è necessario il campo `tpep_pickup_datetime` che rappresenta il timestamp relativo all'istante di partenza del taxi, e quindi l'inizio di una corsa. Inoltre, per il calcolo della quantità media di mancia, la sua deviazione standard della mancia è necessario il campo `tip_amount`, mentre per il tipo di pagamento si utilizza `payment_type`. Infine è necessario calcolare la distribuzione delle zone di pickup, per cui si utilizza anche la colonna `PULocationID`.

A partire dall'oggetto `JavaRDD<Row>` che mantiene il dataset si produce il nuovo `RDD trips`, mappando ogni oggetto `Row` in una tupla tramite la funzione `mapToPair()`; la chiave di ogni tupla è data da una stringa che rappresenta l'orario in formato `YYYY-MM-DD-HH`. Il valore è un oggetto `ValQ2`, che contiene i valori di `tips`, `tips_stddev`, `payment_type`, oltre a `num_payments` e `num_trips` utilizzati poi per il conteggio delle occorrenze.

Si esegue successivamente una `reduceByKey()` per conteggiare il numero totale di viaggi per ogni ora e la quantità totale di mance in quell'ora, producendo l'RDD `aggregated`.

Distribuzione delle Zone

Si calcola innanzitutto, per ogni ora, il numero di viaggi totali ed il numero di viaggi di ogni zona, in modo da calcolarne la distribuzione oraria.

Successivamente si esegue una mapToPair() mappando il precedente RDD su un nuovo RDD avente come chiave un oggetto `KeyQ2PU`, ovvero la coppia `PULocationID` e `hour`, mentre come valore l'intero `1`. Tramite una successiva `reduceByKey()` si aggregano tali valori, calcolando quindi il numero di occorrenze di ogni zona di pickup nelle varie fasce orarie.

Tramite `groupBy()` si raggruppano i risultati sulla base dell'ora, avendo quindi un RDD in cui la chiave è l'ora, ed il valore è una lista del numero di viaggi effettuati per ogni zona. Eseguendo una join si aggiunge per ogni ora anche l'informazione del numero totale di viaggi, in modo da poter calcolarne la distribuzione, semplicemente dividendo il numero di viaggi su ogni zona per il numero totale di viaggi in quella fascia oraria. Si produce l'RDD `grouped_zones()`.

Metodo di Pagamento più Diffuso

Si esegue una `mapToPair()` sull'RDD `trips`, in modo da avere come chiave l'oggetto `KeyQ2Pay`, formato da orario e metodo di pagamento, e come valore l'intero `1`. Tramite una `reduceByKey()` si conta il numero di occorrenze dei metodi di pagamento per ogni ora, e tramite una `groupBy()` si raggruppano tali valori, avendo come chiave l'orario, e come valore la lista dei pagamenti in quell'ora con il relativo numero di occorrenze.

Per ottenere il metodo di pagamento più frequente si utilizza una `mapToPair()`, generando l'RDD `top_payments`; la chiave resta la stringa relativa all'ora, mentre il valore è il `PULocationID` del metodo di pagamento più frequentemente utilizzato in quell'ora. Per ricavare tale informazione si utilizza nella lambda function la funzione `getMostFrequentPayment`, che scorre la lista di iterable dell'RDD e ritorna l'identificativo del metodo con il maggior numero di occorrenze.

Media e Deviazione Standard delle mance

La deviazione standard è data dalla formula

$$tips_{stddev} = \sqrt{\frac{\sum(tip_{val} - tips_{mean})^2}{N}}$$

Si calcola innanzitutto, per ogni ora, la quantità totale di mance ed il numero totale di viaggi effettuati in quell'ora, tramite una `reduceByKey()` sull'RDD `trips`. Successivamente tramite una `mapToPair()` si calcola la media dei tips, dividendo il totale per il numero di viaggi in quell'ora, generando l'RDD `statistics`.

Per il calcolo della deviazione standard sono richiesti i singoli valori dei campi di interesse e i valori medi, motivo per cui si procede con la `join()` tra gli RDD `trips` e `statistics` in base alla chiave `hour`. Ogni elemento presenta quindi le statistiche per una specifica corsa, oltre alla media delle mance di quell'ora necessaria per il calcolo della deviazione standard.

A questo punto, procediamo con una `mapToPair()` sul nuovo RDD, mantenendo solamente i valori di interesse e calcolando i singoli elementi della sommatoria.

$$tips_{dev} = (tips_{amount} - tips_{mean})^2$$

Successivamente si esegue una `reduceByKey()` per calcolare la sommatoria, aggregando quindi tutti i valori $tips_{dev}$ calcolati in precedenza. Tramite una `mapToPair()` si genera l'RDD `deviation`, in cui viene calcolato il valore effettivo della deviazione standard, dividendo la sommatoria per il numero di occorrenze ed eseguendo la radice quadrata.

Risultato Finale

Per produrre l'output finale si esegue un join degli RDD `deviation`, `top_payments` e `grouped_zones`, ed un successivo sortByKey() per ordinare i risultati in base all'ora. Successivamente con una `mapToPair()` si mantengono solamente i campi di interesse, ovvero l'ora, il metodo di pagamento preferito in quell'ora, media e deviazione standard dei tips, e lista contenente la distribuzione in percentuale dei viaggi rispetto ad ogni zona di partenza.

Analisi del DAG

I diversi stages prodotti all'interno del DAG (8.2) derivano dall'invocazione delle *wide transformations*, che richiedono lo shuffle dei dati, e quindi la comunicazione globale tra i diversi nodi di Spark: *stage1* → *stage2* tramite `reduceByKey()` per generare `aggregated` a partire da `trips`, *stage2* → *stage4* RDD `joined`, tramite `join()` di `trips` con `statistics`, *stage4* → *stage5* RDD `deviation` tramite `reduceByKey()` di `iterations`, *stage5* → *stage11* tramite `join()` con gli RDD `top_payments` e `grouped_zones` prodotti rispettivamente negli stages 8 e stage 11 (join di destra).

3.2.2. SparkSQL

Come per la precedente query, viene inizialmente formato un `Dataset` contenente solamente i dati di nostro interesse. Tramite una prima query otteniamo il numero di viaggi effettuati per ogni singola ora in ogni singola zona di NYC. Inoltre, tramite una successiva `JOIN` su un ulteriore `SELECT`, otteniamo il numero di viaggi che vengono effettuati in quella determinata ora nell'*intera* città di New York. Ciò ci permette dunque di calcolare la distribuzione dei viaggi per ogni singola ora, rispetto a tutte le 265 zone in cui NYC è suddivisa.

Una seconda query, utilizza le funzioni SQL `avg()` e `stddev_pop()` per calcolare la media e la deviazione standard della mancia su base oraria, indipendentemente quindi dalla zona di partenza della corsa.

Per quanto riguarda il calcolo del pagamento più utilizzato su base oraria in tutta la città, tramite lo statement `COUNT(*)` viene effettuato il conteggio delle corse raggruppate per ora e tipo di pagamento, ottenendo dunque il numero di pagamenti effettuati per ogni distinto tipo di pagamento, su base oraria. Infine, su tale `SELECT` viene ottenuto il tipo di pagamento più popolare utilizzando la funzione SQL `max()` sul numero di occorrenze di pagamenti.

Infine, al solo scopo di ottenere un output nel formato desiderato, è stata utilizzata la `collect_list()`, funzione offerta da SparkSQL che permette di aggregare tra loro valori in un array, sulle distribuzioni dei viaggi calcolate in precedenza.

Query 3

Per ogni giorno, identificare le 5 zone di destinazione più popolari (in ordine decrescente), indicando per ciascuna di esse il numero medio di passeggeri, la media e la deviazione standard della tariffa pagata (`fare_amount`).

3.2.3. SparkRDD

La query è calcolata su base giornaliera, è quindi necessario il campo tpep_dropoff_datetime che rappresenta il timestamp relativo all'istante di arrivo del taxi, e quindi la fine di una corsa. Inoltre, per il calcolo del numero medio di passeggeri, la media e la deviazione standard della tariffa pagata, sono necessari i campi passenger_count e fare_amount, oltre ovviamente al campo DOLocationID per le zone di destinazione.

Media tariffa e passeggeri per destinazione in ogni giorno

A partire dall'oggetto JavaRDD<Row> che mantiene il dataset si produce il nuovo RDD "days", mappando ogni oggetto Row

in una tupla tramite la funzione mapToPair(), trasformazione simile alla map ma ogni elemento viene mappato in una coppia chiave-valore: la chiave di ogni tupla è data da un oggetto KeyQ3, costituito dal giorno e dalla zona di destinazione, mentre il valore è un oggetto ValQ3, che contiene i valori di passenger_count, fare_amount e 1, utilizzato poi per il conteggio delle occorrenze.

Successivamente si esegue una reduceByKey() per aggregare i valori ricavati tramite la map: essendo la chiave la coppia giorno-destinazione, i valori vengono aggregati su base giornaliera e per zona di destinazione, per ogni elemento dell'RDD si aggregano quindi i vari ValQ3 sommando i diversi valori, ricavando l'RDD "reduced" dove per ogni giorno-zona di destinazione, è presente il totale di passenger_count, fare_amount e occorrenze giornaliere per la precisa zona.

Successivamente si esegue una nuova `mapToPair()` a partire dall'RDD `reduced`; si sfrutta il numero di occorrenze per mappare ogni giorno-zona di destinazione sui valori $\frac{passenger_{count}}{occurrences}$ e $\frac{fare_{amount}}{occurrences}$, producendo il nuovo RDD `mean` che presenta quindi le statistiche medie sul numero di passeggeri e sulla tariffa pagata, oltre al numero di occorrenze per ogni giorno-zona di destinazione.

Deviazione standard tariffa per destinazione ogni giorno

La deviazione standard è data dalla formula

$$fare_{stddev} = \sqrt{\frac{\sum(fare_{val} - fare_{mean})^2}{N}}$$

Per il calcolo della deviazione standard sono richiesti sia i singoli valori dei campi di interesse che i valori medi, motivo per cui si procede con la `join()` tra gli RDD `days` e `mean` in base alla chiave `KeyQ3`. Ogni elemento rappresenta quindi una certa tratta con associate le statistiche per un certo giorno-zona di destinazione, oltre alla media della tariffa, del numero di passeggeri ed il numero di occorrenze generali della singola coppia giorno-zona di destinazione, necessari per il calcolo della deviazione standard.

A questo punto, procediamo con una `mapToPair()` sul nuovo RDD, mantenendo solamente i valori di interesse e calcolando i singoli elementi della sommatoria.

$$fare_{dev} = (fare_{val} - fare_{mean})^2$$

Successivamente si esegue una `reduceByKey()` per calcolare la sommatoria, aggregando quindi tutti i valori $fare_{dev}$ calcolati in precedenza. Tramite una `mapToPair()` si genera l'RDD `deviation`, in cui viene calcolato il valore effettivo della deviazione standard, dividendo la sommatoria per il numero di occorrenze ed eseguendo la radice quadrata.

Ogni elemento di `deviation` rappresenta una zona di destinazione in un giorno specifico, a cui sono associati il numero di passeggeri medio, il numero di occorrenze, la media e la deviazione standard della tariffa pagata.

Risultato Finale

Per produrre l'output finale, si applica la trasformazione `groupBy()` sull'RDD `deviation`, così da raggruppare gli elementi dell'RDD in base al solo giorno: ogni elemento del nuovo RDD `grouped` ha quindi come chiave una stringa che rappresenta il giorno, mentre come valore una lista con le varie zone di destinazione e le statistiche ad esse legate.

Successivamente si applica all'RDD la funzione `mapToPair()` per selezionare su ogni giorno solamente le cinque zone di destinazione più popolari: si produce l'RDD `top_destinations` dove ogni elemento ha come chiave il giorno, e come valore una lista contenente cinque tuple, con i cinque DOLocationID più popolari, ed i relativi ValQ3 con le statistiche ad essi associati. Per ricavare le cinque destinazioni più popolari, si utilizza nella lambda function la funzione user-defined `getTopFiveDestinations()` che itera su tutta la lista di destinazioni associata al giorno per identificare le cinque zone con il maggior numero di occorrenze.

A questo punto, si applica la funzione `sortByKey()` all'RDD `top_destinations` per ordinare il risultato in maniera decrescente e fornire un output ordinato dal primo all'ultimo giorno.

Analisi del DAG

I diversi stages prodotti all'interno del DAG (7.3) derivano dall'invocazione di trasformazioni che richiedono lo shuffle dei dati, e quindi è necessaria la comunicazione globale tra i diversi nodi di Spark: *stage1* → *stage2* RDD reduced tramite `reduceByKey()` su `days()`, *stage2* → *stage4* tramite `join()` tra `days` e `mean`, *stage4* → *stage5* `stddev_aggr` tramite `reduceByKey()` su `iterations`, stage5 → stage6 tramite tramite `groupBy()` su `deviation`.

3.2.4. SparkSQL

Prima dell'esecuzione della query viene utilizzata la funzione `createSchemaFromRDD` per costruire tramite map l'RDD dei soli campi necessari a partire dal dataset, e creare da questo un dataframe che presenta i campi di interesse per la query: `tpep_dropoff_datetime`, `passenger_count`, `fare_amount` e `DOLocationID`.

A partire dal dataframe si effettua una select sfruttando la clausola `GROUP BY` su giorno e zona di destinazione, per recuperare le statistiche di passeggeri e tariffa e il conteggio giornaliero degli arrivi per ogni giorno-destinazione. Con una successive clausola `ORDER BY` si ordina in modo decrescente in base al nome della zona e al conteggio degli arrivi.

Avendo raggruppato ogni giorno in un insieme di righe che descrivono le diverse zone di destinazione, sfruttiamo l'ordinamento decrescente in base al numero di occorrenze

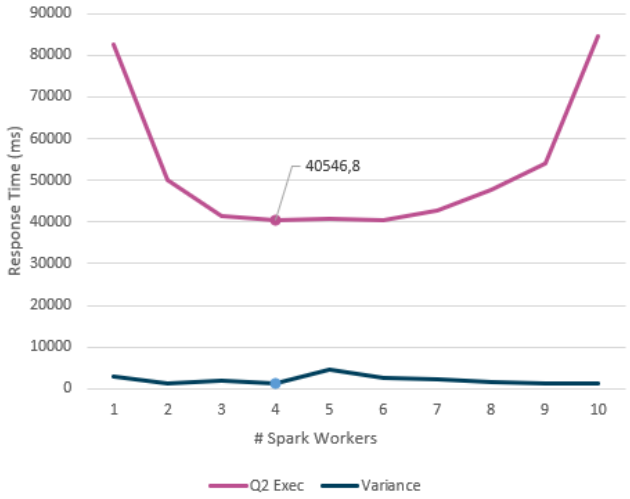


Figura 4 – Tempo di risposta query2 al variare del numero di workers

tramite l'operatore `ROW_NUMBER()`: indice di riga che agisce su una partizione, che definita sul giorno, consente di recuperare solamente le prime cinque righe di ognuno, che rappresentano le zone di destinazione più popolari richieste dalla query.

Per presentare il risultato come l'esempio di output suggerito sono state aggiunte due select. La prima sfrutta l'operatore `COLLECT_LIST()` su ogni gruppo giornaliero per unire le diverse righe, avremo quindi per ogni giorno una sola colonna per le destinazioni, medie dei passeggeri, medie e deviazioni standard della tariffa, ognuna caratterizzata da un array di cinque valori associati alle cinque diverse zone più popolari del giorno.

L'ultima select viene utilizzata per separare i valori concatenati in colonne singole tramite l'operatore `ELEMENT_AT()`, in particolare le colonne prodotte per ogni zona di destinazione giornaliera vengono convertite in stringa da intero per inserire il nome della zona: a questo proposito è stata sviluppata la funzione `UDF SetZones` che tramite l'ID scorre una JavaMap che mantiene il mapping tra ID e zone dei viaggi e al matching restituisce la stringa corrispondente per inserirla nel campo convertito.

4. TESTING E ANALISI DELLE PERFORMANCE

Per analizzare le prestazioni del sistema è stato prima individuato il numero ottimale di worker Spark, e successivamente sono state testati i tempi di risposta per ognuna delle query.

La macchina utilizzata per il testing ha le seguenti specifiche:

```
AMD Ryzen 5 5500U with Radeon Graphics, @6x2.10 GHz, RAM 8GB DDR4 @ 1593.9 MHz, Linux 5.15.23-76051523-generic amd64 (Pop!_OS by System76).
```

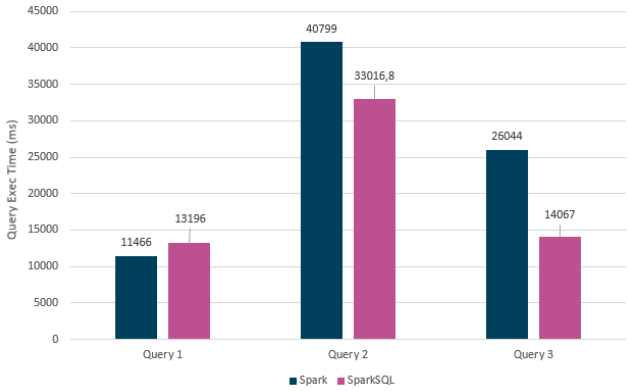


Figura 5 – Confronto tempo di risposta delle query RDD API vs SparkSQL

4.1. Numero di Workers

Per individuare il numero ideale di worker spark è stato utilizzato un approccio sperimentale, andando a testare il tempo di risposta al variare del numero di worker in esecuzione sull'ambiente locale. In particolare, è stato valutato il comportamento della query 2, essendo quella più complessa, ripetendo 5 iterazioni per ogni configurazione, e considerando per ognuna media e varianza.

Analizzando i risultati forniti dal grafico (*figura4*) è possibile vedere come con 1 o 2 worker non si sfrutta il parallelismo di spark, mentre da 3 a 6 worker si ha circa sempre lo stesso tempo di risposta (40s). Da 7 a 10 worker invece il tempo di risposta torna ad aumentare, in quanto si va a saturare la macchina locale su cui spark è in esecuzione.

Analizzando la varianza vediamo che il risultato migliore si ottiene utilizzando 4 worker, tramite cui la query 2 impiega sempre 40 secondi. Invece con 5 e 6 worker aumenta la varianza, quindi anche se con lo stesso tempo medio di risposta, si hanno risultati più variabili. Per questo è stata preferita la configurazione con 4 nodi worker, sia perché risulta essere più stabile nei tempi di esecuzione, sia perché grava meno sulle risorse della macchina locale.

4.2. Testing delle Query

Per testare le prestazioni delle query, vengono eseguite diverse ripetizioni di ogni query, resettando l'ambiente di esecuzione al termine di ogni ripetizione. In questo modo sono state valutate le prestazioni a parità di condizioni in ognuno dei test, caricando ogni volta il dataset nel sistema. Si eseguono 5 ripetizioni di ogni test e per ognuno si valutano media e varianza dei tempi di risposta nell'esecuzione di ogni query.

Come possiamo vedere dal grafico (*figura5*), complessivamente sia nel caso di SQL che degli RDD, la query 2 è quella che ha il tempo di risposta maggiore. Questo è un comportamento atteso, in quanto aggrega dati su una finestra temporale più fine (delle ore), oltre a richiedere più computazioni rispetto alla query 3. Infatti la query 3 aggrega ogni destinazione per ogni giorno, mentre la query 2 aggrega sia ogni `POLocationID` in ogni ora che ogni metodo di pagamento per ogni ora.

Come possiamo vedere dal grafico, per la query 1 otteniamo tempi di risposta simili sia utilizzando gli RDD che utilizzando le Dataset API. Questo è dovuto principalmente alla semplicità della query, che richiede solo poche trasformazioni. Utilizzando gli RDD si ha un tempo di risposta di 11 secondi, contro i 13 di SparkSQL. Questo perché i dataset sono comunque ottimizzati per le operazioni di aggregazione, e non essendo presenti operatori di aggregazione (*es.* `groupBy()`), si beneficia nelle prestazioni dell'utilizzo di API di più basso livello.

Per le restanti due queries si hanno prestazioni migliori nel caso di SparkSQL. Questo è un comportamento atteso, in quanto le Dataset API risultano maggiormente ottimizzate nelle operazioni di grouping e aggregazione rispetto alle RDD API[6]. Infatti nelle query 2 e 3 sono presenti molte operazioni di groupBy() e `reduceBykey()`, che risultano più efficienti nelle query SQL che utilizzando i Dataset vanno a sfruttare a pieno i vantaggi del catalyst optimizer [7].

In particolare per la query 2 vediamo un miglioramento di 7 secondi nel caso di SparkSQL, mentre per la query 3 un miglioramento ancora più netto, di circa 12 secondi. Questo è probabilmente dovuto al fatto che nella soluzione proposta della query 2 in sql sono presenti due `join()`, mentre nessuna `join()` viene utilizzata per la query 3.

5. CONCLUSIONI

5.1. Punti di Forza

Le query impiegano complessivamente un tempo ragionvevole per l'esecuzione, anche considerando la dimensione importante del dataset.

Inoltre, a livello di processamento si è risparmiato tempo avendo eseguito il preprocessamento su NiFi, e quindi lavorando con una versione ridotta e filtrata del dataset.

5.2. Limitazioni

Grafana non dispone di un plugin per leggere i dati direttamente da hdfs, ma soltanto per monitorare le risorse del cluster. Pertanto non è stato possibile automatizzare il ciclo di visualizzazione dei risultati, motivo per cui simuliamo tale scenario caricando i csv nel container grafana.

Una possibile alternativa sarebbe stata quella di utilizzare il plugin di mongo disponibile su grafana, che ci avrebbe permesso di leggere i documenti salvati su tale storage in modo automatico e visualizzarli su grafana al termine del processamento. Tuttavia tale soluzione richiede un account Grafana Cloud Pro, oppure una licenza attiva di Grafana Enterprise, per cui si è optato per la prima soluzione.

Avendo utilizzato una soluzione basata su containerizzazione non sono stati sfruttati a pieno i vantaggi del processamento parallelo, avendo come collo di bottiglia le risorse computazionali della macchina utilizzata. Utilizzando i servizi cloud di AWS si sarebbe simulato uno scenario più vicino a quello reale, evitando il collo di bottiglia descritto e potendo scalare idealmente ad un numero infinito di nodi.

6. RIFERIMENTI

[1] Nifi Docker Image: https://hub.docker.com/r/apache/nifi

[2] HDFS Docker Image: https://hub.docker.com/r/matnar/hadoop

[3] Spark Docker Image: https://hub.docker.com/r/apache/spark

[4] MongoDB Docker Image: https://hub.docker.com/_/mongo

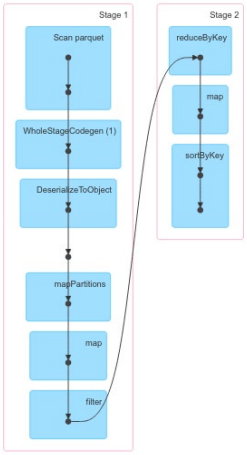
[5] Grafana Docker Image: https://hub.docker.com/r/grafana/grafana

[6] https://www.analyticsvidhya.com/blog/2020/11/what-is-the-difference-between-rdds-dataframes-and-datasets

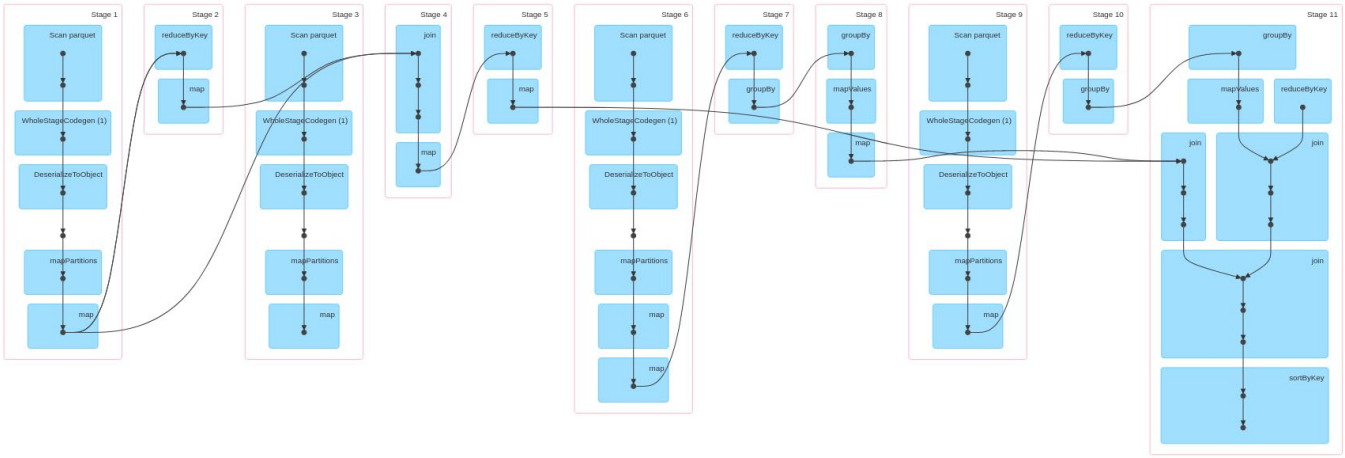
[7] https://blog.bi-geek.com/en/spark-sql-optimizador-catalyst/

7. APPENDICE: DAGs

7.1. Query 1



7.2. Query 2



7.3. Query 3