Sistemi e Architetture per Big Data - AA 2021/2022
Progetto1: Analisi del dataset dei taxi di NYC

Martina De Maio
Corso di laurea magistrale in Ingegneria Informatica
Università degli studi di Roma "Tor Vergata"
martina.demaio@alumni.uniroma2.eu

***Abstract*—La seguente trattazione descrive l'implementazione di una soluzione per una serie di query su un dataset riguardante i viaggi in taxi nella città di New York, durante un periodo compreso tra Dicembre 2021 e Febbraio 2022. La soluzione proposta è stata sviluppata utilizzando il framework di data processing Apache Spark ed il file system distribuito di Hadoop (HDFS) per la parte di data ingestion.**
Per il deployment dell'applicazione è stata utilizzata un'istanza Linux di Amazon EC2 (Elastic Computing Cloud).

1. Introduzione

Lo scopo del progetto è rispondere a 3 query riguardanti il dataset dei dati della città di New York (NYC), utilizzando il framework di data processing Apache Spark.

Per gli scopi di questo progetto si utilizzano i seguenti file, forniti in formato *Parquet*, relativi, rispettivamente, ai viaggi dei taxi di colore giallo ed ai mesi di dicembre 2021, gennaio 2022 e febbraio 2022:

- *yellow tripdata 2021-12.parquet*,
- *yellow tripdata 2022-01.parquet*,
- *yellow tripdata 2022-02.parquet*,

disponibili agli URLs [1]-[3].

Le query a cui rispondere sono le seguenti:

1.1. Query1

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. Si chiede di indicare anche il numero totale di corse usate per calcolare il valore medio.

1.2. Query2

Per ogni ora, calcolare la distribuzione in percentuale del numero di corse rispetto alle zone di partenza (la zona di partenza è indicata da `PULocationID`), la mancia media e la sua deviazione standard, il metodo di pagamento più diffuso.

1.3. Query3

Per ogni giorno, identificare le 5 zone di destinazione (`DOLocationID`) 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`).

2. Architettura

Per il deployment dell'applicazione è stata utilizzata un'istanza Linux di **Amazon EC2** (Elastic Computing Cloud), sulla quale stati avviati un cluster di **Spark** per il processamento batch ed un cluster **HDFS** per leggere il dataset di input e memorizzare i risultati di output.

A tale scopo è stato utilizzato **Docker Compose**, che consente di definire e condividere applicazioni Docker multi-container utilizzando un file yaml denominato `docker-compose.yml`, nel quale si specifica la configurazione dei servizi della propria applicazione, e con un solo comando si creano e avviano tutti i servizi a partire dalla configurazione specificata. Si utilizza per creare container multipli all'interno di uno stesso host e coordinare l'esecuzione di più container in esecuzione su una singola macchina.

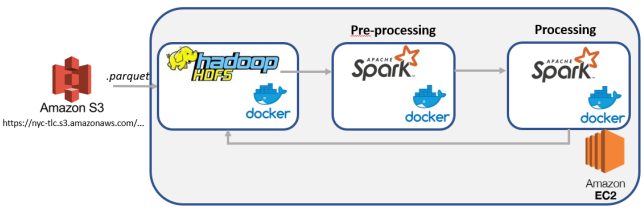


Figure 1. Architettura utilizzata.

2.1. Amazon Elastic Compute Cloud (Amazon EC2)

I files necessari al deployment dell'applicazione si trovano nella repository GitHub https://github.com/martina97/sabd_project1 nella cartella **"docker"**, che contiene il file `dockercompose. yml`e i vari scripts utili al deploy, che servono sia per il caricamento dei file parquet di input su HDFS, sia per l'esecuzione dell'applicazione sul cluster di Spark.

Amazon EC2 fornisce capacità di calcolo scalabile in Amazon Web Services (AWS) Cloud. Consente, quindi, l'avviamento di server virtuali, configurandone le caratteristiche di sicurezza e scalabilità. EC2 permette l'implementazione di applicazioni fornendo un servizio web attraverso il quale un utente può avviare un'Amazon Machine Image (AMI) per creare una macchina virtuale ("istanza"), che conterrà il software desiderato. Un utente può creare, lanciare, e chiudere istanze, pagando i server a consumo, da cui l'aggettivo "elastica" nel nome del servizio. EC2 fornisce agli utenti il controllo sulla collocazione geografica delle istanze che permettono un'ottimizzazione della latenza e alti livelli di ridondanza.

Il tipo di istanza Linux utilizzata è una **t2.large**, tipo di istanza per uso generico e a basso costo che offre un livello base di prestazioni della CPU con la possibilità di espanderne la potenza in base alle proprie esigenze. Le istanze T2 sono ideali per un'ampia gamma di applicazioni per uso generico tra cui microservizi, applicazioni interattive a bassa latenza, database di piccole e medie dimensioni, desktop virtuali, sviluppo, ambienti di build e temporanei, repository di codici e prototipazione di prodotti.

Le caratteristiche di un'istanza t2.large sono le seguenti:

- 2 vCPU,
- 8 GiB di RAM,
- processore ad alta frequenza *Intel Xeon* scalabile fino a 3,0 GHz (*Haswell E5-2676 v3 o Broadwell E5-2686 v4*),

2.2. Hadoop Distributed File System (HDFS)

L'Hadoop Distributed File System è il componente principale dell'ecosistema Hadoop ed è un file system distribuito, portabile e scalabile utilizzato per l'archiviazione di dati di grandi dimensioni. Si è utilizzato HDFS per leggere i dataset di input, disponibili in formato *Parquet*, e su di esso sono stati scritti i file di output, in formato *CSV*. L'architettura dell'HDFS è un'architettura di tipo master-worker, in particolare in un cluster HDFS ci sono due tipi di nodi:

- un master, chiamato **NameNode**, su cui risiedono i metadati dei file,
- più workers, chiamati **DataNodes**, su cui risiedono, in blocchi di dimensione fissa, i file dell'HDFS

Partendo da un file di grandi dimensioni, HDFS lo suddivide in blocchi solitamente di 64 o 128 megabytes, e ogni blocco può essere memorizzato su nodi detti DataNodes, come mostrato nella seguente figura:

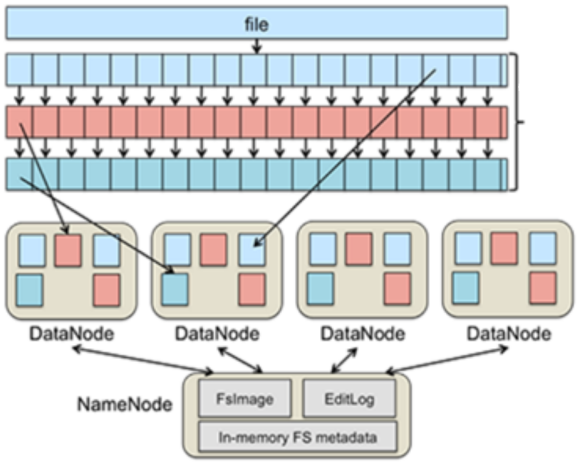


Figure 2. Funzionamento di HDFS.

Per avere un file system distribuito tollerante ai guasti, i blocchi in HDFS sono sparsi su diversi DataNodes e possono essere replicati su più macchine.

2.3. Apache Spark

Spark è un **framework di processamento batch**, un engine per Big Data veloce e general purpose che permette un'analisi unificata non soltanto di dati batch ma anche di dati stream, potenziando le funzionalità di Apache Spark tramite opportune librerie. Spark è completamente compatibile con tutto lo storage e le API usate da Hadoop, e la caratteristica saliente di Spark che gli consente di ottenere uno speed up fino a 10 volte rispetto alle prestazioni di Hadoop MapReduce per algoritmi di tipo iterativo è il fatto che lo storage non è più su disco ma in memory, all'interno della memoria RAM. L'architettura di Spark è un'architettura distribuita di tipo master-worker: c'è un master, detto **Spark Driver**, e molteplici worker detti **Spark Worker**. Come layer di memorizzazione viene usato **HDFS**.

Il programma principale (Driver Program), dialoga con il Cluster Manager, che alloca le risorse. All'interno dei nodi workers ci sono in esecuzione degli ambienti di esecuzione della computazione, detti Executor, e all'interno di ogni Executor verranno istanziati dei Task, che sono gli ambienti di esecuzione degli operatori. Gli executor sono processi che eseguono calcoli e memorizzano dati per l'applicazione. Ogni applicazione in Spark è costituita da un Driver Program ed Executors instanziati sul cluster.
Lo Spark core ci fornisce le funzionalità di base utilizzate da altri componenti, tra cui pianificazione dei task, gestione della memoria, ripristino dei faults, interazione con i sistemi di archiviazione, e fornisce un'astrazione dei dati denominata **Resilient Distributed Dataset** (**RDD**), una raccolta di elementi distribuiti su molti nodi di calcolo appartenenti al cluster di Spark che possono essere processati in parallelo. Gli RDD sono l'astrazione di dati fondamentale in Spark,

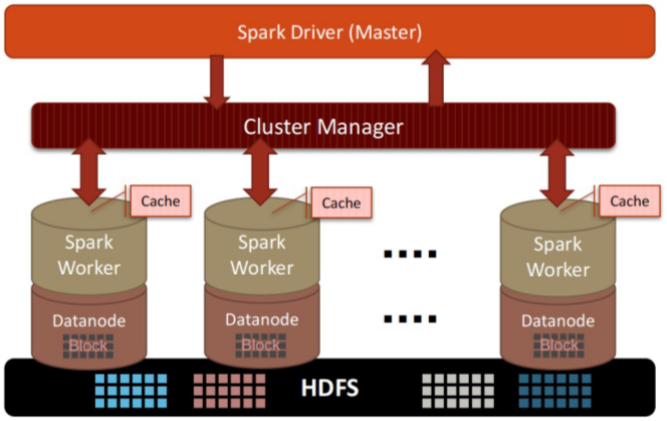
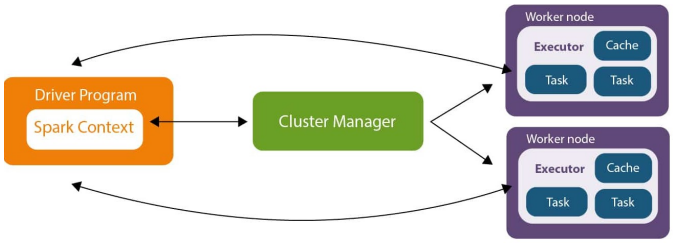


Figure 3. Architettura di Spark.



e un RDD si può immaginare come un array composto da tanti elementi in cui blocchi di elementi sono partizionati, e le diverse partizioni sono distribuite sui diversi nodi. Un RDD è una collezione di elementi distribuita, memorizzata in memoria RAM, immutabile, partizionata e tollerante ai guasti. Avendo partizionamento, si può lavorare in parallelo su partizioni diverse. Lo Spark Core fornisce API per la creazione e la manipolazione degli RDD.

3. Descrizione dell'implementazione

L'applicazione è stata scritta nel linguaggio di programmazione Java ed è stata gestita utilizzando Apache Maven, in modo da facilitare il download automatico delle librerie necessarie e la risoluzione di eventuali dipendenze. Il codice sorgente per rispondere alle 3 query, disponibile nella repository GitHub https://github.com/martina97/sabd_project1, si trova nella cartella *"src/main/java"* ed è suddiviso in tre packages: ***"queries"***, contenente le classi *Query1.java*, *Query2.java*, *Query3.java* e *StartQueries.java*, ***"SQLQueries"***, che contiene la classe *SqlQuery1.java*, ***"utils"***, contentente classi di supporto per l'implementazione del codice.

3.1. Acquisizione dei dati e Preprocessamento

I 3 dataset sono disponibili in formato Parquet, e hanno tutti una dimensione tra i 38 e i 50 MB. Poichè convertendo i file da foramto Parquet a CSV la dimensione dei file aumenta però di circa 7x, si è deciso di lavorare direttamente in formato Parquet. Una volta avviati i vari container, tramite lo script `start_hdfs.sh` , sono stati caricati all'interno di HDFS, in particolare nella cartella */data*, i 3 file Parquet, in modo da poterli leggere successivamente.

Tramite il metodo ***importParquet()*** all'interno della classe *QueriesPreprocessing.java* si prelevano i 3 dataset da HDFS, tramite `spark.read().parquet("path.parquet")`, preservando lo schema dei file in quanto i file Parquet sono autodescrittivi, e il risultato di tale operazione è un Dataset<Row>. Successivamente si trasforma tale Dataset in un JavaRDD tramite il metodo ***toJavaRDD()***.

Per la parte di preprocessamento si è utilizzato Spark, andando a filtrare, per ogni query, le colonne di interesse e rimuovendo eventuali valori nulli.

3.2. Processamento

3.2.1. Query1. Per la query1 inizialmente si è effettuata una fase di preprocessamento, attraverso la quale sono state selezionate le colonne di interesse, ossia tpep_pickup_datetime, payment_type, tip_amount, tolls_amount, tot_amount, dove:

- `tpep_pickup_datetime`(timestamp): data e ora in cui il tassametro è stato attivato,
- `Payment_type`(long): codice numerico che indica come il passeggero ha pagato il viaggio,
- `Tip_amount`(double): importo della mancia; viene compilato automaticamente per le mance pagate con carta di credito, mentre le mance in contanti non sono incluse,
- `Tolls_amount`(double): importo totale di tutti i pedaggi pagati durante il viaggio
- `Total_amount`(double): importo totale addebitato ai passeggeri; non include le mance in contanti.

Inoltre, tramite la trasformazione ***filter*** sono stati filtrati solamente i dati di interesse ai fini dell'analisi, ossia relativi a Dicembre 2021, Gennaio e Febbraio 2022, scartando i dati non relativi a tali mesi, e includendo solamente i pagamenti effettuati con carta di credito.

Successivamente, nella parte di processamento vera e propria, inizialmente è stata effettuata una ***mapToPair***, in modo da ottenere un `JavaPairRDD` formato da coppie chiave-valore, in cui la chiave è il mese e il valore è il rapporto tra la mancia e il costo della corsa esclusi i pedaggi, ossia il rapporto tip_amount/(total_amount – tolls_amount). Poiché tale rapporto può avere come risultato dei valori NaN, ad esempio se si effettua la divisione 0.0/0.0, si è effettuata l'operazione di ***filter*** andando a escludere eventuali NaN. In seguito si è effettuata una ***aggregateByKey***, che aggrega i valori di ciascuna chiave utilizzando determinate funzioni di combinazione. In particolare, per combinare i valori associati a una determinata chiave si è utilizzata la classe ***StatCounter*** della libreria `org.apache.spark.util`,

la quale consente di tracciare le statistiche di un insieme di numeri (conteggio, media, varianza ...) in modo numericamente robusto. In questo modo, è stato possibile calcolare la media dei valori associati ad ogni chiave, ossia la media dell'importo della mancia rispetto al costo della corsa e il numero totale di corse usate per calcolare il valore medio (rispettivamente i campi *mean()* e *count()* dello StatCounter, il quale contiene le statistiche dei valori per ogni chiave). Successivamente è stata effettuata una ***mapToPair***, che crea un JavaPairRDD avente come chiave il mese e come valore una Tupla2 formata dalla percentuale della media appena calcolata e dal numero totale di corse usate per calcolare il valore medio, e infine una ***sortByKey*** per ordinare l'RDD per chiave in ordine crescente.

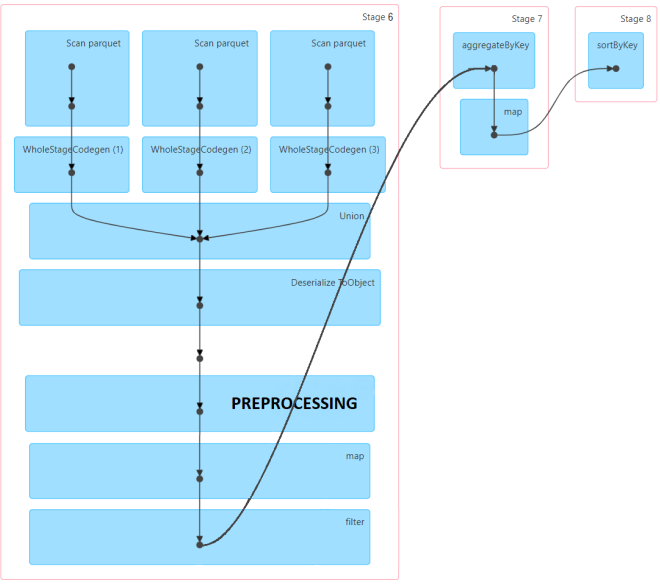


Figure 4. DAG Query1.

3.2.2. Query2. Per quanto riguarda la query2, nella fase di preprocessamento le colonne considerate sono tpep_pickup_datetime, PULocationID, payment_type, tip_amount. `PULocationID` indica la zona di partenza in cui il tassametro è stato attivato, mentre per la descrizione delle altre colonne si rimanda al paragrafo relativo alla query1. Anche in questo caso, tramite una ***filter*** sono stati scartati i valori non relativi alle date di interesse ed eventuali valori nulli (NaN) contenuti nel dataset di partenza. Per rispondere a tale query il flusso di lavoro è stato diviso in 3 parti:

1) Tramite il metodo ***CalculateDistribution()***, si calcola per ogni ora la distribuzione in percentuale del numero di corse rispetto alle zone di partenza
2) Tramite il metodo ***CalculateAvgStDevTip()*** si calcola per ogni ora la mancia media e la sua deviazione standard
3) Tramite il metodo ***CalculateTopPayment()*** si calcola per ogni ora il metodo di pagamento più diffuso.

Nel metodo *CalculateDistribution()* si effettua inizialmente una ***mapToPair*** sull'RDD generato dal preprocessamento, tramite la quale si crea un JavaPairRDD avente come chiave la specifica ora, e come valore una Tupla2 contenente la zona di partenza e il numero 1. A tale JavaPairRDD si è poi applicata una ***countByKey*** per salvare in una mappa il numero di viaggi che partono in ogni ora, che poi verrà utilizzato per poter calcolare la distribuzione in percentuale. Successivamente, tramite una ***mapToPair*** si è trasformato il JavaPairRDD precedente in un altro JavaPairRDD avente stavolta come chiave la Tupla2 composta dall'ora e dalla zona, e 1 come valore, in modo così da poter effettuare una ***reduceByKey***, per contare, in modo simile al caso del WordCount, il numero di corse rispetto a quella determianta zona in quell'ora. A questo punto si effettua un'ulteriore ***mapToPair*** per generare un JavaPairRDD avente come chiave l'ora e come valore una Tupla2 composta dalla zona di partenza e dalla distribuzione in percentuale del numero di corse rispetto a quella zona, calcolata utilizzando la mappa contenente il numero di corse effettuate per ogni ora. Infine si effettua una groupByKey.

Per calcolare invece la mancia media e la sua deviazione standard, all'interno del metodo ***CalculateAvgStDevTip()*** si effettua inizialmente una ***mapToPair*** in modo da creare un JavaPairRDD avente come chiave l'ora e come valore il campo tip_amount. Successivamente viene applicata a tale RDD una ***aggregateByKey*** utilizzando uno ***StatCounter***, per calcolare la media e la deviazione standard dei valori associati ad ogni chiave, ossia, in questo caso, la mancia media e la sua deviazione stantard per ogni ora, in modo del tutto analogo al calcolo effettuato nella query1. Successivamente tramite una ***mapToPair*** si genera un JavaPairRDD avente come chiave la singola ora e come valore una Tupla2 contenente la mancia media e la sua deviazione standard calcolate tramite lo StatCounter.

Per il calcolo del metodo di pagamento più diffuso, invece, all'interno del metodo *CalculateTopPayment()* si è effettuata una mapToPair per generare un JavaPairRDD avente come chiave una tupla2 contenente l'ora e il metodo di pagamento, e 1 come valore. In modo analogo al calcolo del wordCount si è effettuata una ***reduceByKey*** per sommare le occorrenze dei singoli pagamenti per ogni ora, e successivamente tramite una ***mapToPair*** si è generato un JavaPairRDD avente come chiave l'ora e il numero di occorrenze del singolo pagamento, e come valore il tipo di pagamento. Tramite una ***sortByKey*** che utilizza un ***Comparator*** di Tuple2 viene ordinato l'RDD in base al numero di occorrenze dei pagamenti in ordine decrescente, così da effettuare poi una ***mapToPair*** che genera un JavaPairRDD avente come chiave l'ora e come valore una Tupla2 composta dal pagamento e il numero di occorrenze di tale pagameto, e poi,

tramite una ***groupByKey***, si raggruppano tutti i valori relativi a quella chiave in un ***Iterable*** che contiene già i valori ordinati in modo decrescente. In questo modo, il pagamento più diffuso per ogni ora è il primo elemento dell'Iterable, che si può recuperare tramite il metodo `Iterables.get(iterable, 0)`.

Una volta generati i 3 JavaPairRDD, aventi tutti come chiave l'ora, e come valore, rispettivamente:

- un Iterable di Tuple2 contenenti la coppia (PULocationID, distribuzione in percentuale del numero di corse rispetto a tale zona),
- una Tupla2 contenente la coppia (media, standard deviation) della mancia
- un Double che indica il metodo di pagamento più diffuso per tale fascia oraria

si effettua il join dei 3 RDD, in modo da eseguire un inner-join sulla chiave degli RDD.

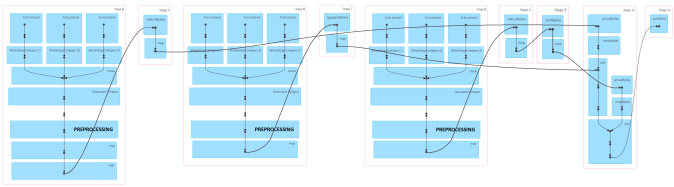


Figure 5. DAG Query2.

3.2.3. Query3. Per la parte di preprocessamento della query 3 le colonne di interesse che sono state considerate sono tpep_pickup_datetime, passenger_count, ossia il numero dei passeggeri nel veicolo, DOLocationID (long), ossia la zona di destinazione, fare_amount, ossia la tariffa pagata. Poiché la parte di preprocessamento, per ogni riga del dataset divide i diversi campi tramite uno split, durante la trasformazione in double della stringa relativa al campo passenger_count, ci si è accorti eseguendo il codice che spesso tale stringa era una stringa "null" in quanto si generava una EXCEPTION, quindi le righe contenenti tali stringhe null sono state scartate. Anche in questo caso sono state scartate le date non relative ai 3 mesi di interesse.

Per rispondere alla query3 inizialmente è stata effettuata una ***mapToPair***, generando un PairRDD avente come chiave la Tupla2 formata dalla coppia giorno, DOLocationID e come valore il campo `fare_amount`. Tramite una ***aggregateByKey*** sono stati aggregati tutti i valori corrispondenti alla stessa chiave utilizzando nuovamente l'oggetto ***StatCounter***, in modo da ottenere, per ogni chiave, l'insieme delle statistiche dei valori associati a tale chiave. Tramite una ***mapToPair*** si è poi creato un PairRDD avente come chiave la chiave precedente giorno, DOLocationID, e come valore una Tupla3 contenente (occorrenze della zona di destinazione per quel giorno, media della tariffa pagata, deviazione standard della tariffa pagata). Chiamiamo tale PairdRDD ***meanStdevFareAmountRDD***.

Successivamente, tramite una ***mapToPair*** si è creato un ulteriore PairRDD avente come chiave sempre la coppia giorno, DOLocationID, e come valore il campo `passenger_count`. Tramite una ***aggregateByKey***, combinata con lo ***StatCounter***, e con una successiva ***mapToPair***, a ogni chiave giorno, DOLocationID è stata associata la media di passenger_count, così da avere, per ogni giorno e per ogni zona di destinazione, il numero medio di passeggeri. Chiamiamo tale RDD risultatnte ***avgPassengerRDD***.

A questo punto si è effettuata una ***join*** di *meanStdevFareAmountRDD* e avgPassengerRDD, in modo da generare un PairRDD composto da coppie chiave-valore in cui la chiave è composta dalla tupla2 giorno, DOLocationID, e i valori sono occorrenze DOLocationID,media fare_amount,std fare_amount,num medio passeggeri. Tale RDD è poi stato ordinato per chiave utilizzando un Comparator di Tuple2, in modo da ordinare in ordine decrescente per il numero di occorrenze di DOLocationID. In questo modo, con una successiva ***mapToPair*** si crea un RDD avente come chiave il giorno e come valori una Tupla4 composta da DOLocationID,media fare_amount,std fare_amount,num medio passeggeri, ordinato sempre per occorrenze decrescenti di DOLocationID, e tramite una ***groupByKey*** si raggruppano i valori per ogni chiave, generando un Iterable di Tuple4 ordinato in modo decrescente. Per prendere i primi 5 valori di tale iterable, che corrispondono alle 5 zone di destinazione più popolari (in ordine decrescente), tramite il metodo `Iterables.limit(iterable, 5)` si genera, per ogni chiave, un nuovo Iterable composto dalle prime 5 Tuple4. In questo modo, per ogni giorno si associa una lista composta da 5 Tuple4, composte a loro volta, in ordine decresente, dai valori DOLocationID,media fare_amount,std fare_amount,num medio passeggeri.

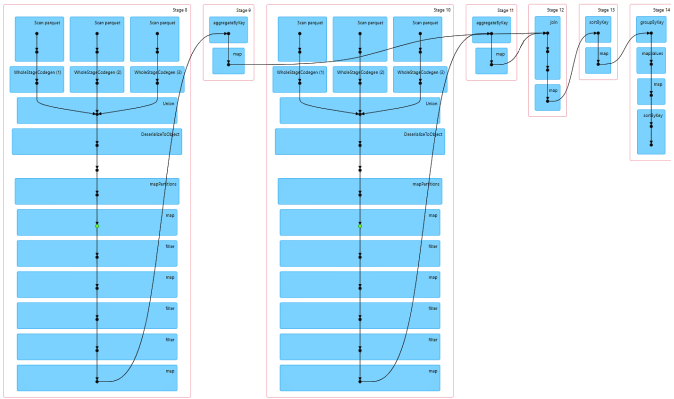


Figure 6. DAG Query3.

3.2.4. Query1 SQL. Per rispondere alla Query 1 utilizzando SparkSQL, una volta generato il JavaRDD dai file Parquet, si crea un Dataset<Row> a partire da tali dati, tramite il

metodo createSchemaFromPreprocessedData, selezionando come colonne di interesse le stesse colonne utilizzate per rispondere alla query1. Si da a tale tabella il nome "query1". Di seguito il dettaglio del codice utilizzato per rispondere a tale query:

```
"SELECT tpep_pickup_datetime, AVG(tip_amount/(total_amount-tolls_amount)) AS tip_percentage,"+
 "count(*) as trips_number FROM query1 " +
"GROUP BY tpep_pickup_datetime ORDER BY tpep_pickup_datetime"
```

Figure 7. Codice Query1 SQL.

Si è effettuata la media di tutti i rapporti (tip_amount/total_amount-toll_amount) per ogni chiave tpep_pickup_datetime, ossia il mese, e calcolato il numero di occorrenze di tale chiave. Infine si è raggruppato ed ordinato per mese.

4. ANALISI DEI TEMPI

I test sono stati effettuati su un'istanza Linux di EC2, avente a disposizione 8 GiB di RAM e 2 vCPU, e processore Intel Xeon.

Sono stati effettuati 3 cicli di test, con un numero di nodi worker per HDFS e Spark pari a 2,3,4. In tabella si riportano i tempi relativi a ogni Query misurati in millisecondi:

# workers	Query1 (ms)	Query2 (ms)	Query3 (ms)	Query1 SQL (ms)
2	61060	217709	103477	65230
3	72549	222678	104332	77678
4	75775	235225	106031	87565

Figure 8. Tempi di esecuzione delle query al variare del numero di nodi.

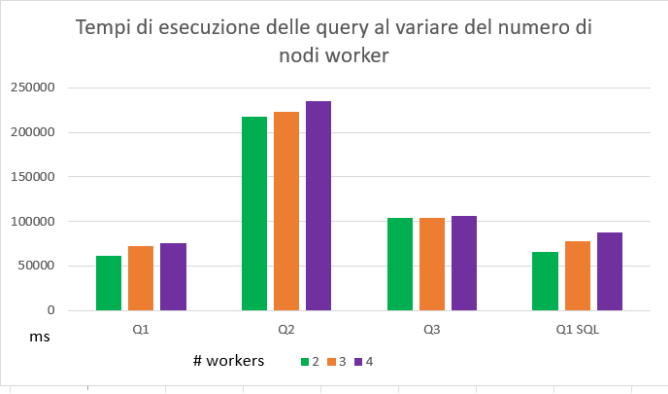


Figure 9. Confronto tempi di esecuzione query al variare del numero di nodi.

Sono state effettuate anche prove con un numero maggiore di 4 di nodi senza comunque avere miglioramenti, motivo per cui non sono state riportate in tabella.

Dalla tabella e dai grafici si può osservare che, al variare del numero dei nodi, non si ha un notevole cambiamento nelle prestazioni, questo dovuto molto probabilmente al fatto che i dataset utilizzati non sono di grandi dimensioni (tra i 38 e i 50 MB), e quindi l'overhead per allocare le risorse è nettamente maggiore rispetto al tempo di computazione vero e proprio dell'applicazione.

Gli alti tempi di processamento si possono attribuire in parte al tipo di macchina utilizzata per effettuare i test, in quanto dà a disposizione solamente 2 vCPU.

Inoltre, l'utilizzo della classe StatCounter per calcolare le varie medie, deviazioni standard e conteggi potrebbe causare un rallentamento dei tempi di processamento in quanto, utilizzato insieme alla aggregateByKey, calcola, per ogni chiave, le statistiche dei suoi valori, quindi il numero di occorrenze, il valor medio, i valori massimi e minimi, la deviazione standard, la somma.

Dal grafico precedente si può osservare che la Query2 è quella che presenta tempi di esecuzione più alti. Questo si può attribuire al fatto che, anche visualizzandone il DAG, si effettuano molte più trasformazioni rispetto alle altre 2 query, ed inoltre si chiede di analizzare i dati per ogni ora, e non mensilmente o giornalmente.

Inoltre, si può osservare che i tempi di processamento della Query 1 non cambiano sensibilmente se si utilizza Spark SQL, quindi usando Dataset invece che RDD.

5. Sviluppi futuri

Per ridurre i tempi di processamento, una prima opzione è quella di provare ad utilizzare un'altra istanze di EC2, ad esempio una t2.xlarge o t2.2xlarge, che mettono a disposizione, rispettivamente, 4 vCPU e 8 vCPU.

Un altro miglioramento è filtrare e ridurre il dataset, in fase di preprocessamento, utilizzando il framework Apache Nifi.

References

[1] https://nyc-tlc.s3.amazonaws.com/trip+data/yellow_tripdata_2021-12.parquet

[2] https://nyc-tlc.s3.amazonaws.com/trip+data/yellow_tripdata_2022-01.parquet

[3] https://nyc-tlc.s3.amazonaws.com/trip+data/yellow_tripdata_2022-02.parquet