Analisi del dataset dei taxi di NYC con Apache Spark

Martina De Maio

0296447

Scopo del progetto

- Analizzare, mediante il framework di data processing **Apache Spark** ed il file system distribuito di Hadoop (**HDFS**), il dataset reale contenente dati riguardanti i viaggi in taxi nella città di New York (NYC), durante un periodo compreso tra Dicembre 2021 e Febbraio 2022, rispondendo a 3 diverse query.

Sommario

1) Architettura:

2) Queries

- Query1
- Query2
- Query3
- Query1SQL

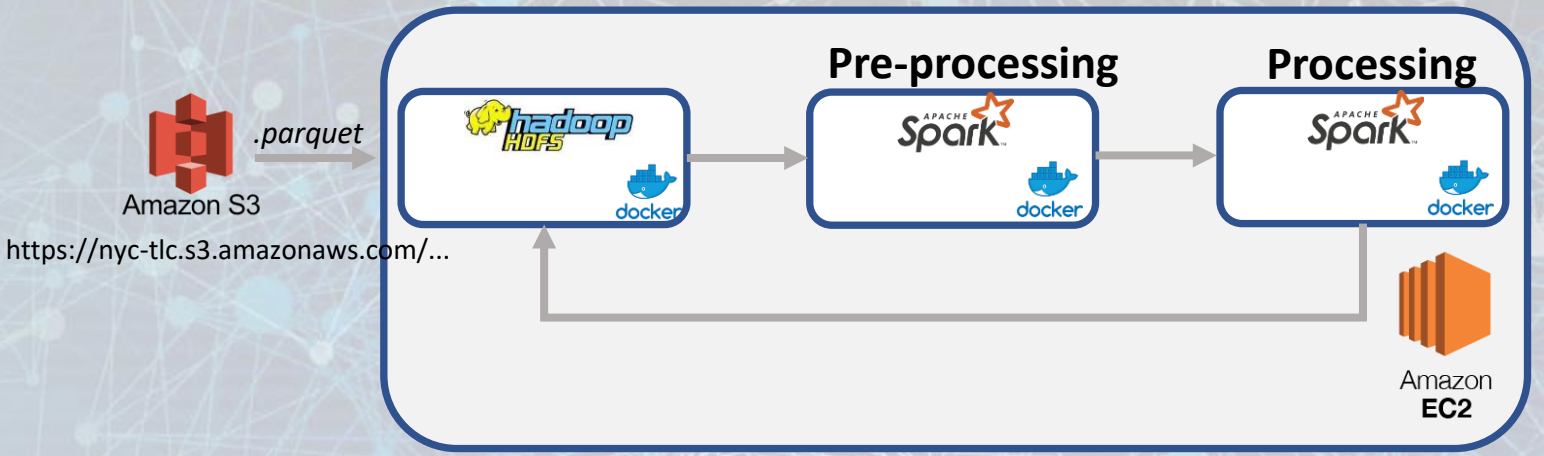
3) Analisi dei tempi di processamento

Architettura

- Per il deployment dell'applicazione è stata utilizzata un'istanza Linux di **Amazon EC2,** 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.

- Il tipo di istanza Linux utilizzata è una **t2.large**, avente le seguenti caratteristiche:
  - 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*)

- Per avviare i vari cluster è stato utilizzato **Docker Compose**, che consente di definire e condividere applicazioni Docker multi-container utilizzando il file `docker-compose.yml`, nel quale si specifica la configurazione dei servizi della propria applicazione.



Queries – Acquisizione dei dataset

- Per gli scopi del progetto sono stati utilizzati 3 diversi dataset, 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
- Tali file vengono inizialmente caricati all'interno di HDFS nella cartella /data, in modo da poterli leggere successivamente tramite il metodo `spark.read().parquet("path.parquet")`, il quale genera 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.

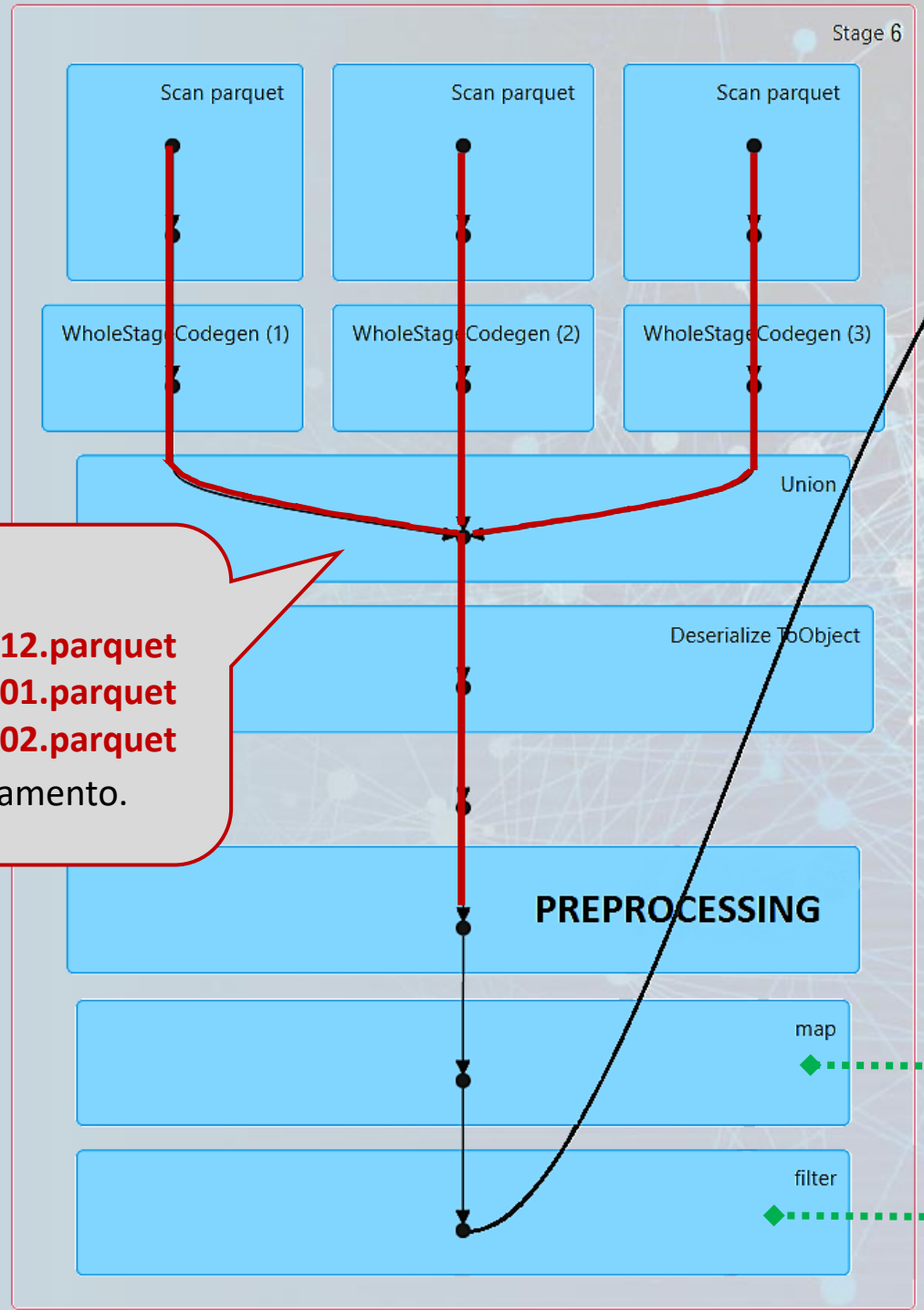
Query 1: descrizione

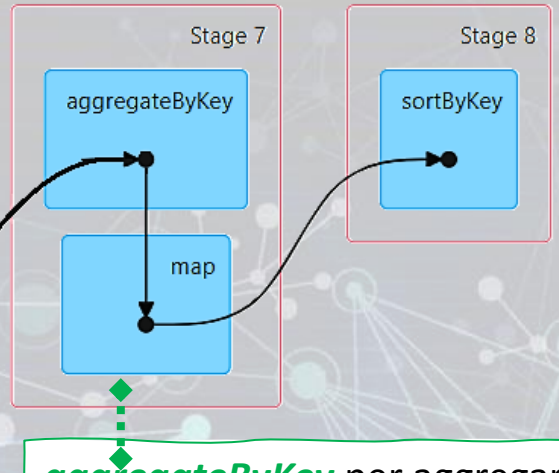
Per ogni mese solare, calcolare la percentuale media dell'importo della mancia rispetto al costo della corsa esclusi i pedaggi. Calcolare il costo della corsa come differenza tra l'importo totale (`Total_amount`) e l'importo dei pedaggi (`Tolls_amount`) ed includere soltanto i pagamenti effettuati con carta di credito. Si chiede di indicare anche il numero totale di corse usate per calcolare il valore medio.

- Nella fase iniziale di preprocessamento sono state selezionate le colonne di interesse:
  - `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
  - `Tolls_amount` (`double`): importo totale di tutti i pedaggi pagati durante il viaggio
  - `Total_amount` (`double`): importo totale addebitato ai passeggeri.

- 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, ed includendo solamente i pagamenti effettuati con carta di credito.

Query 1 DAG





Caricamento
yellow tripdata 2021-12.parquet
yellow tripdata 2022-01.parquet
yellow tripdata 2022-02.parquet
da HDFS e preprocessamento.

aggregateByKey per aggregare i valori di ogni chiave utilizzando la classe ***StatCounter*** della libreria `org.apache.spark.util`, che consente di tracciare le statistiche di un insieme di numeri (conteggio, media, varianza ...).

mapToPair per associare a ogni **mese** la percentuale media e il numero totale di corse usate per calcolare tale media (presi, rispettivamente, dai campi mean() e count() dello StatCounter).

Per associare ad ogni **mese** il rapporto tip_amount/(total_amount - tolls_amount).

Per rimuovere i valori NaN generati dal rapporto tip_amount/(total_amount - tolls_amount).

Martina De Maio, 0296447

Query 2: descrizione

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.

- Nella fase iniziale di preprocessamento sono state selezionate le colonne di interesse:
  - `tpep_pickup_datetime`
  - `PULocationID`: zona di partenza in cui il tassametro è stato attivato
  - `payment_type`
  - `tip_amount`

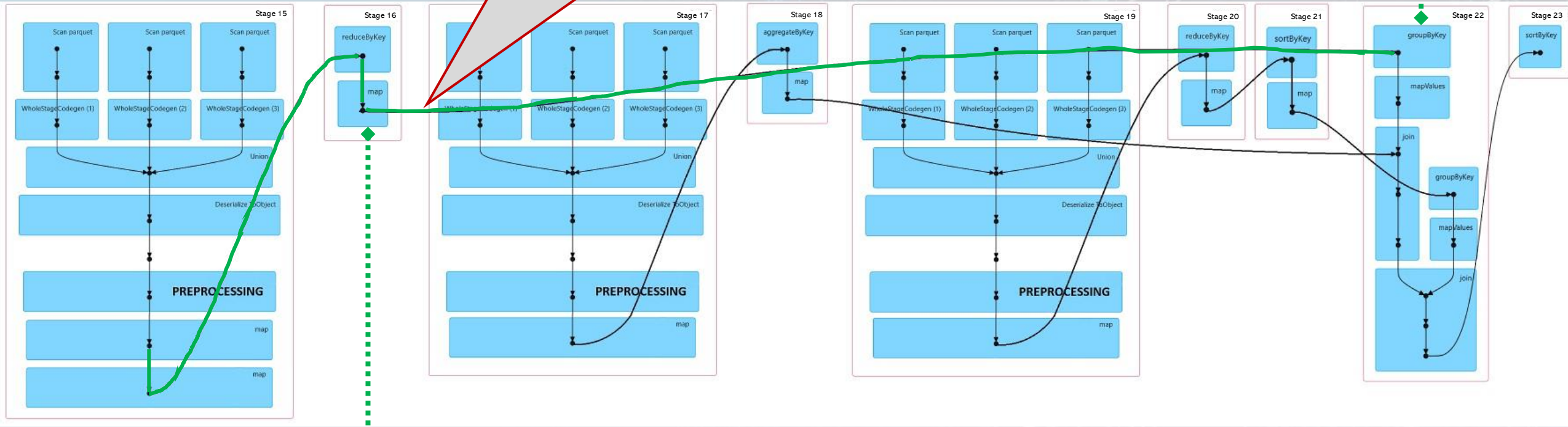
- Anche in questo caso, tramite una ***filter*** sono stati scartati i valori non relativi alle date di interesse ed eventuali valori nulli (NaN) contenuti nel dataset di partenza.

Martina De Maio, 0296447

Query 2 DAG

Calcolo, per ogni ora, la distribuzione in percentuale del numero di corse rispetto alle zone di partenza.

Infine si effettua una ***groupByKey per raggruppare i valori per chiave.***



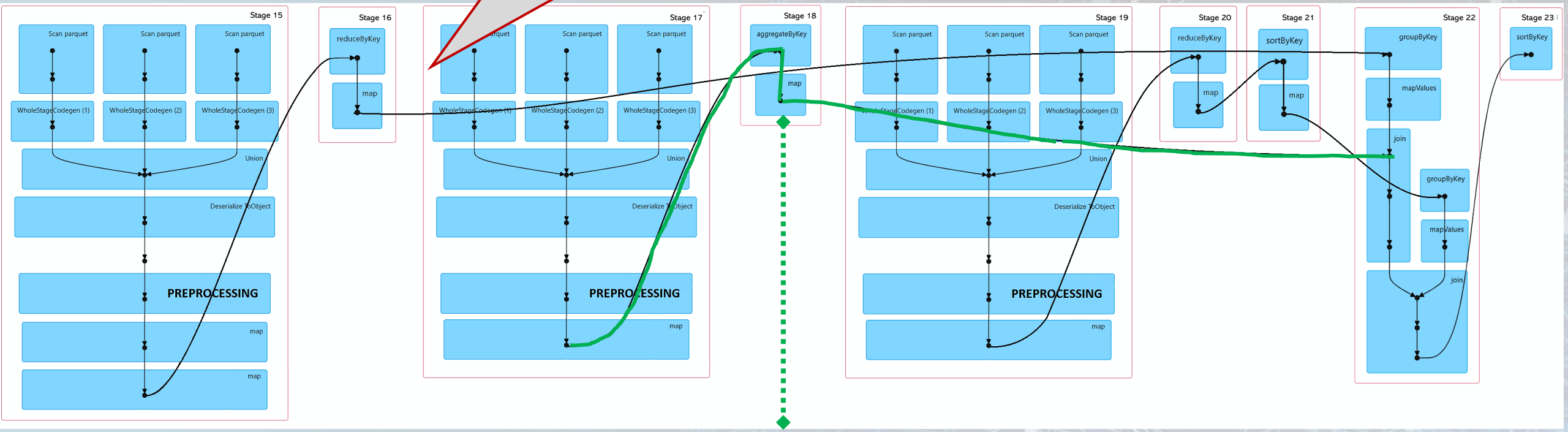
Tramite una ***mapToPair*** si associa ad ogni coppia {**ora, zona di partenza**} il valore 1, in modo da poi effettuare una ***reduceByKey*** e, in modo simile al caso del WordCount, sommare il numero di corse rispetto a quella determinata zona in quell'ora.

Successivamente 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 generata in precedenza tramite una ***countByKey***.

Martina De Maio, 0296447

Query 2 DAG

Calcolo, per ogni ora, la mancia media e la sua deviazione standard.

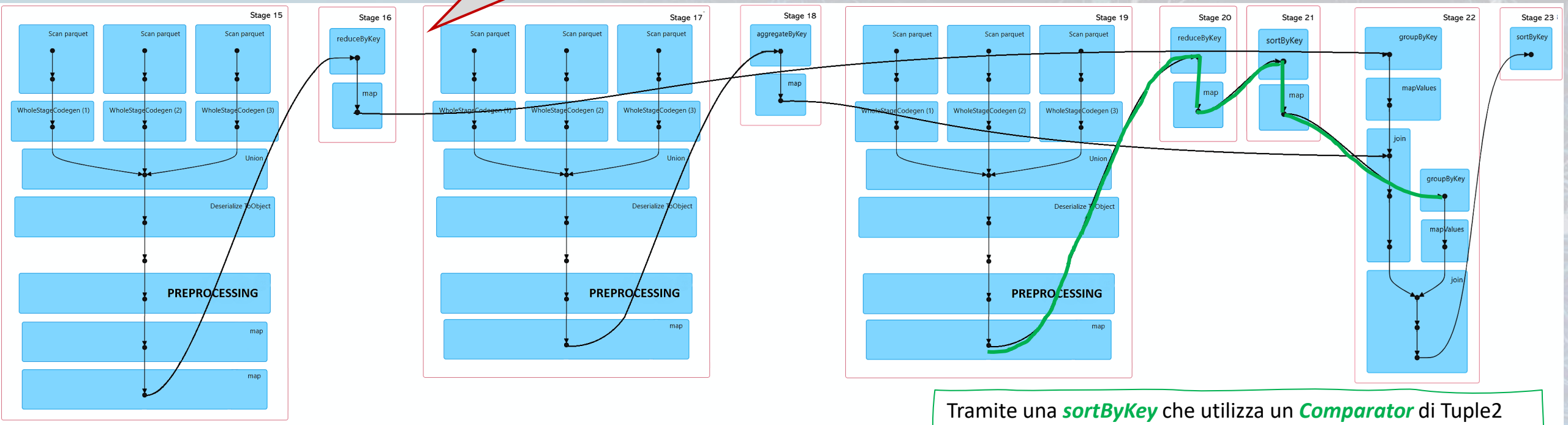


Tramite una ***mapToPair*** si associa ad **ora** il valore del `fare_amount` corrispondente, e tramite una ***aggregateByKey*** insieme allo ***StatCounter*** si è calcolata la media e deviazione standard del `fare_amount` per ogni chiave.

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.

Query 2 DAG

Calcolo, per ogni ora, il metodo di pagamento più diffuso.

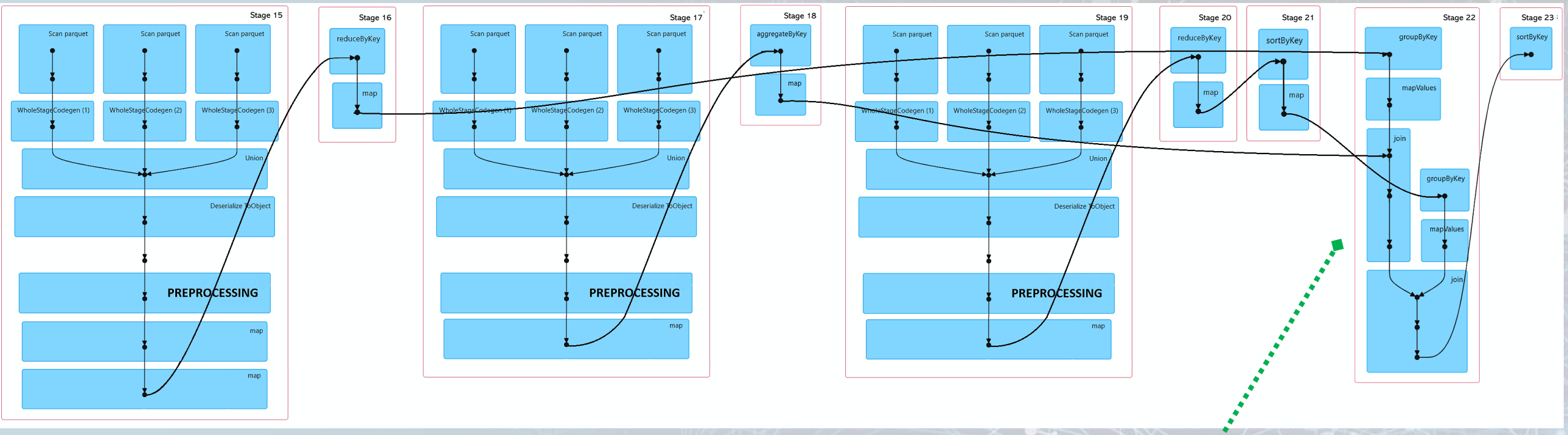


Si usa una ***mapToPair*** per generare un JavaPairRDD avente come **chiave** una tupla2 contenente **l'ora** e il **metodo di pagamento**, e 1 come valore. In modo analogo al calcolo del wordCount si è effettuata una ***reduceByKey*** per sommare le occorrenze dei singoli pagamenti per ogni ora, e successivamente 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, cosi 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 pagamento, 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)`.

Query 2 DAG

Join dei 3 RDD risultanti.



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.

Martina De Maio, 0296447

Query 3: descrizione

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).

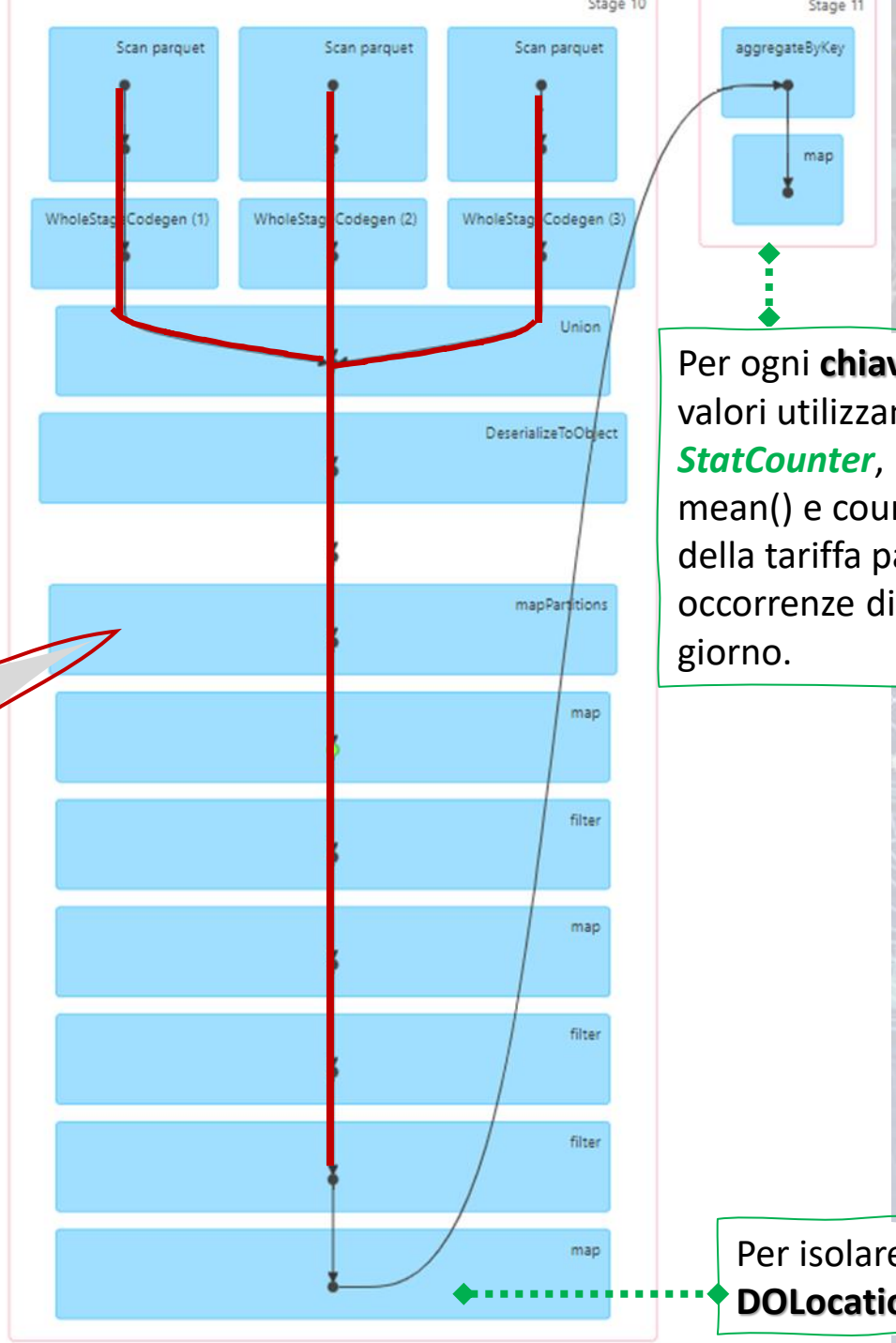
- Nella fase iniziale di preprocessamento sono state selezionate le colonne di interesse:
  - `tpep_pickup_datetime`
  - `passenger_count`: Numero dei passeggeri nel veicolo
  - `DOLocationID`: zona di destinazione in cui il tassametro è stato disattivato
  - `Fare_amount`: tariffa pagata

- 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.

Query 3: DAG

Per ogni giorno, si indica per ogni zona di destinazione (DOLocationID) la media e la deviazione standard della tariffa pagata .

Caricamento
yellow tripdata 2021-12.parquet
yellow tripdata 2022-01.parquet
yellow tripdata 2022-02.parquet
da HDFS e preprocessamento.

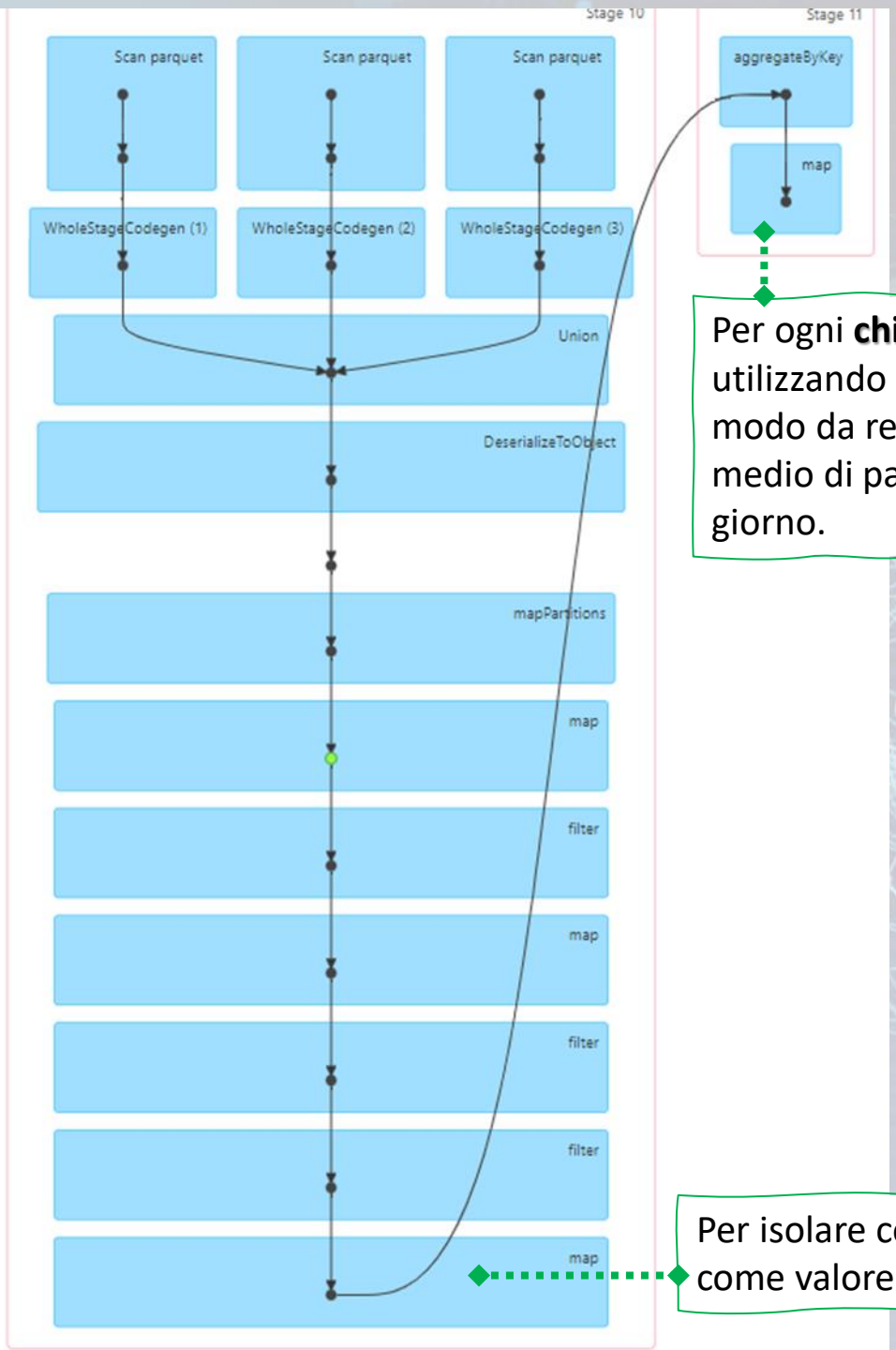


Per ogni **chiave** (**giorno, DOLocationID**) si aggregano i valori utilizzando una ***aggregateByKey*** con uno ***StatCounter***, in modo da recuperare, tramite i campi mean() e count(), la media e la deviazione standard della tariffa pagata (`fare_amount`) e il numero di occorrenze di ogni zona di destinazione per ogni giorno.

Per isolare come **chiave** la coppia (**giorno, DOLocationID**) e come valore `fare_amount`.

Query 3: DAG

Per ogni giorno, si indica per ogni zona di destinazione (DOLocationID) il numero medio di passeggeri.



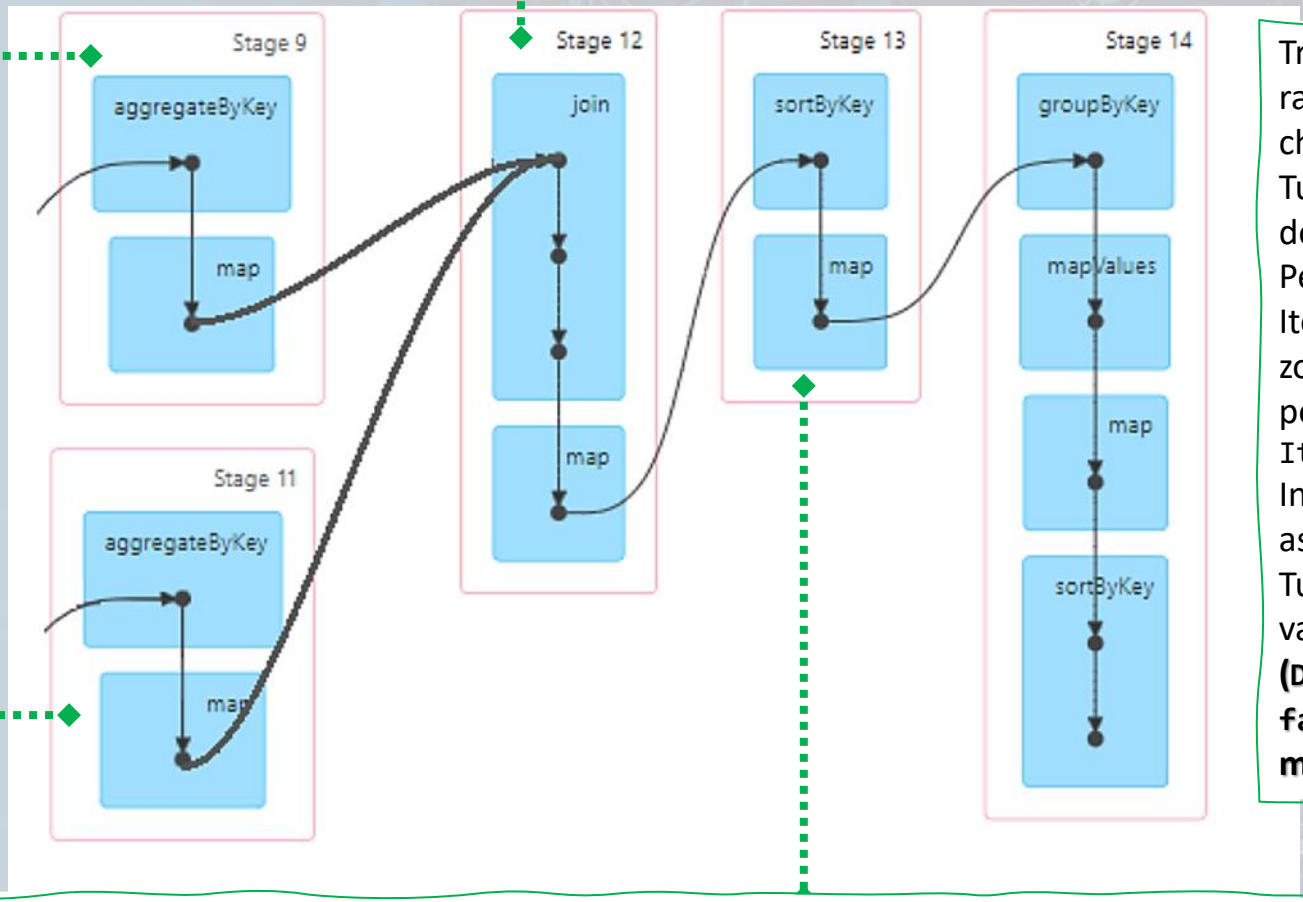
Per ogni **chiave (giorno, DOLocationID)** si aggregano i valori utilizzando una ***aggregateByKey*** con uno ***StatCounter***, in modo da recuperare, tramite il campo mean(), il numero medio di passeggeri per ogni zona di destinazione per ogni giorno.

Per isolare come **chiave** la coppia (**giorno, DOLocationID**) e come valore passenger_count.

Query 3: DAG

mapToPair per isolare come **chiave** la coppia **(giorno, occorrenze DOLocationID),** e come valore una Tupla4 composta da (DOLocationID, media fare_amount,std fare_amount, num medio passeggeri)

JavaPairRDD<**Tuple2**<giorno, DoLocationID >, **Tuple3**<occorrenze DoLocationID, media fare amount, stdev fare amount>>



JavaPairRDD<**Tuple2**<giorno, DOLocationID>, numero medio passeggeri>

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, per ogni chiave si utilizza il metodo `Iterables.limit(iterable, 5)`.
In questo modo, per ogni giorno si associa Iterable composto da 5 Tuple4, composte a loro volta dai valori **(`DOLocationID`,media `fare_amount`,std `fare_amount`,num medio passeggeri).**

sortByKey: per ordinare l'RDD per chiave utilizzando un Comparator di Tuple2, in modo da ordinare in ordine decrescente per il numero di occorrenze di DOLocationID.
mapToPair: per isolare come ***chiave*** il ***giorno***, e come valori una Tupla4 composta da (DOLocationID,media fare_amount,std fare_amount,num medio passeggeri). Tale RDD è ordinato sempre per occorrenze decrescenti di DOLocationID.

Martina De Maio, 0296447

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"
```

- 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 tramite count(*) si calcola il numero totale dei viaggi.
- Infine si è raggruppato ed ordinato per mese.

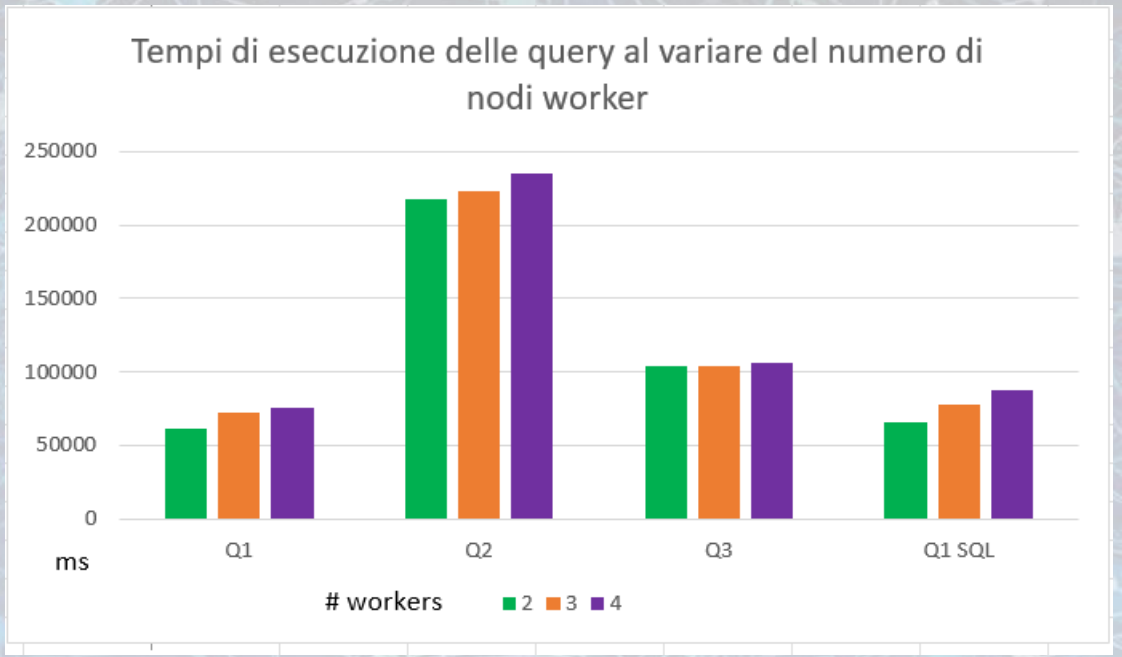
Analisi tempi di processamento

- 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.
- Di seguito 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

Analisi tempi di processamento

- 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 ha a disposizione solamente **2 vCPU**.
- Con analisi più approfondite dei tempi di processamento ci si è accorti che la stragrande maggioranza del tempo di processamento ricade nella ***sortByKey,*** che viene utilizzata per tutte e 3 le query per poter scrivere il CSV ordinato. Prendendo i tempi di processamento escludendo la sortByKey finale, con 2 nodi worker i tempi di processamento delle 3 queries si dimezzano.
- La Query2 è quella che presenta tempi di esecuzione più alti. Questo si può attribuire al fatto che 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.
- I tempi di processamento della Query 1 non cambiano sensibilmente se si utilizza Spark SQL, quindi usando Dataset invece che RDD.



Sviluppi futuri

- Per ridurre i tempi di processamento, una prima opzione è quella di provare ad utilizzare un'altra istanza 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.