Report Big Data project:
High Volume FHV TRip Records

Alberto Antonelli

June 9, 2023

Contents

1 Introduzione

1.1 Descrizione del dataset

Il dataset analizzato (https://www.kaggle.com/datasets/shuhengmo/uber-nyc-forhire-vehicles-trip-data-2021), contiene informazioni sui viaggi fatti da varie licenze "taxi" nella città di New York.
La grandezza totale del dataset è di 5 GB e contiene molte informazioni sui luoghi di partenza, arrivo, prezzo e varie tempistiche come l'orario e giorno di partenza e quelli d'arrivo.
Il dataset può essere utilizzato per analizzare il traffico di varie licenze Taxi nella città di New York, per comprendere la frequenza con cui i visitatori della città dendono ad utilizzarli, oltre a fare varie analisi sulla tipologia di licenza maggiormente utilizzata.
I dati del dataset sono nel formato Parquet. Quindi è stata necessaria una importazione e conversione in RDD. Solo successivamente sono state eseguite le query per le analisi.
I dati sono raggruppati mensilmente, per un totale di 12 file Parquet. Il dataset è composto dai seguenti attributi:

- licenseClass: tipo stringa che rappresenta la licenza de taxi.
  - HV0002: Juno
  - HV0003: Uber
  - HV0004: Via
  - HV0005: Lyft
- license: tipo Stringa che rappresenta il numero del taxi.
- request: tipo Timestamp che rappresenta la data/ora della richiesta del viaggio.
- pickup: tipo Timestamp che rappresenta la data/ora dell'inizio del viaggio.
- dropoff: tipo Timestamp che rappresenta la data/ora della fine del viaggio.
- distance: tipo Double che rappresenta la distanza totale in miglia percorsa dal taxi per questo viaggio.
- startloc: tipo Long che rappresenta la zona di partenza del viaggio.
- endloc: tipo Long che rappresenta la zona di arrivo del viaggio.
- fare: tipo Double che rappresenta il prezzo del viaggio.

Qui sopra ci sono elencati quelli che sono stati importati ma ulteriori dati sono all'interno del dataset di partenza.

1.2 Obbiettivi dell'analisi

L'obbiettivo del progetto è analizzare il dataset, al fine di rilevare informazioni utili sui dati raccolti. Di seguito vengono riportate le query più o meno complesse che ci siamo posti e a cui si è cercato di dare una risposta analizzando il dataset:

- Numero totale di record;
- Numero totale di zone di arrivo diverse;
- Stampa delle classi di licenza;
- Viaggi più lunghi in termine di KM, in relazione con la durata del viaggio;
- Numero di viaggi effettuati da ciascun conducente in base all'ora di inizio del viaggio;
- Viaggi più costosi in termine di KM;
- Distanza totale percorsa da ciascuna licenza di taxi in ciascuna zona di partenza;
- Tempo di viaggio medio, in base alla zona di partenza;
- In media quanti viaggi fa ciascuna licenza;
- Numero di viaggi effettuati per ogni combinazione di ora di inizio del viaggio e zona di arrivo;
- Classe di licenza che fa più viaggi, con la presenza anche della tariffa cliente;
- Licenza che fa più viaggi, con la presenza anche della tariffa cliente;
- Licenza di taxi con il prezzo medio del viaggio più alto per ciascuna classe di licenza;
- Distanza media percorsa dai taxi in un singolo viaggio, in relazione alla zona;
- Zona di arrivo più comune per ciascuna licenza di taxi;
- Analisi sul guadagno medio di ogni singolo viaggio per ogni giorno.

2 Setting per l'analisi

2.1 Configurazione del cluster

Prima di parlare nel dettaglio delle varie query, 'e necessario, anche se pur brevemente, parlare della configurazione del cluster. Nella mia analisi ho preferito una configurazione di default, utilizzata anche nella maggiorparte dei laboratori. Esso 'e composto da 1 master e 2 nodi slave, entrambi con 3 core e 6 GB di RAM. Credo di poter affermare che questo setting è risultato adeguato, perciò l'ho mantenuto per l'intero progetto.

```
%%configure -f
{"executorMemory":"6G", "numExecutors":2, "executorCores":3, "conf":
{"spark.dynamicAllocation.enabled": "false"}}
```

2.2 Prepocessing dei dati

Nell'immagine sotto possiamo vedere un metodo extract che converte un oggetto di tipo org.apache.spark.sql.Row in un'istanza della classe TaxiTrip, che sarà poi utilizzata durante il progetto.

In [53]:
```
import java.sql.Timestamp
case class TaxiTrip(
  licenseClass:String,
  license:String,
  request:Timestamp,
  pickup:Timestamp,
  dropoff:Timestamp,
  distance:Double,
  startloc:Long,
  endloc:Long,
  //time:Long,
  fare:Double,
)

object TaxiTrip {
    def extract(row:org.apache.spark.sql.Row) = {
        val licenseClass = row.getString(0)
        val license = row.getString(1)
        val request = row.getTimestamp(4)
        val pickup = row.getTimestamp(5)
        val dropoff = row.getTimestamp(6)
        val distance = row.getDouble(9)
        val startloc = row.getLong(7)
        val endloc = row.getLong(8)
        //val time = row.getLong(10)
        val fare = row.getDouble(11)

        new TaxiTrip(licenseClass,license,request,pickup,dropoff,distance,startloc,endloc,fare)


        //new TaxiTrip(license,request,pickup,dropoff,startloc,endloc,distance,time,fare)
    }
}
```

3 Query

3.1 Numero totale di record

```
]: rddTaxiTrip.count()
```

▸ Spark Job Progress

```
res6: Long = 11908468
```

3.2 Numero totale di zone di arrivo diverse

```
rddTaxiTrip.map(x => x.endloc).distinct().count()
```

▸ Spark Job Progress

```
res7: Long = 262
```

3.3 Stampa delle classi di licenza

```
rddTaxiTrip.map(x => x.licenseClass).distinct().collect()
```

▸ Spark Job Progress

```
res12: Array[String] = Array(HV0004, HV0005, HV0003)
```

3.4 Viaggi più lunghi in termine di KM, in relazione con la durata del viaggio

Viene creato un RDD applicando una trasformazione map all'RDD di partenza. Nella trasformazione map viene creata una tupla con il campo distance come chiave e la differenza tra la data/ora di dropoff e la data/ora di pickup divisa per 60.000 (per ottenere il tempo di viaggio in minuti) come valore. Quindi, la tupla risultante viene restituita come risultato della trasformazione.
La stessa cosa viene fatta per l'altra query
Dopo aver fatto la union tra i due RDD, vengono ordinati per la chiave.

```
In [18]: val query1 = rddTaxiTrip.map(x => (x.distance, (x.dropoff.getTime() - x.pickup.getTime())/(60*1000)))
         val query1802 = rddTaxiTrip2.map(x => (x.distance, (x.dropoff.getTime() - x.pickup.getTime())/(60*1000)))

         query1: org.apache.spark.rdd.RDD[(Double, Long)] = MapPartitionsRDD[81] at map at <console>:35
         query1802: org.apache.spark.rdd.RDD[(Double, Long)] = MapPartitionsRDD[82] at map at <console>:35

In [19]: val query1union = query1.union(query1802).cache()

         query1union: org.apache.spark.rdd.RDD[(Double, Long)] = UnionRDD[83] at union at <console>:36

In [20]: query1union.sortByKey(false).take(10)
```

▸ Spark Job Progress

```
res13: Array[(Double, Long)] = Array((738.95,666), (527.11,575), (512.5,541), (480.73,512), (454.49,452), (432.359,379), (417.7
7,469), (408.19,440), (389.65,370), (381.95,444))
```

3.5 Numero di viaggi effettuati da ciascun conducente in base all'ora di inizio del viaggio

```
val result = rddTaxiTrip.map(x => ((getDayTime(x.pickup.getTime()).get(Calendar.HOUR)), 1)).reduceByKey(_ + _).collect()
```

▸ Spark Job Progress

```
result: Array[(Int, Int)] = Array((4,844296), (0,934327), (8,1182479), (1,886481), (9,1076083), (5,953156), (6,1082789), (10,1061788), (2,870515), (11,987859), (3,848995), (7,1179700))
```

3.6 Viaggi piu costosi in termine di KM

```
val query2 = rddTaxiTrip.filter(_.distance > 20).map(x => (x.fare, ((x.dropoff.getTime() - x.pickup.getTime())/(60*1000))))
val query2802 = rddTaxiTrip2.filter(_.distance > 20).map(x => (x.fare, ((x.dropoff.getTime() - x.pickup.getTime())/(60*1000))))
```

▸ Spark Job Progress

```
query2: org.apache.spark.rdd.RDD[(Double, Long)] = MapPartitionsRDD[23] at map at <console>:28
query2802: org.apache.spark.rdd.RDD[(Double, Long)] = MapPartitionsRDD[25] at map at <console>:28
```

```
val query2union = query2.union(query2802).cache()
```

▸ Spark Job Progress

```
query2union: org.apache.spark.rdd.RDD[(Double, Long)] = UnionRDD[26] at union at <console>:29
```

```
query2union.sortByKey(false).take(10)
```

▸ Spark Job Progress

3.7 Distanza totale percorsa da ciascuna licenza di taxi in ciascuna zona di partenza

Viene applicata una trasformazione map all'RDD. Per ogni elemento viene creato una tupla con la chiave composta da license e startloc e il valore distance. Quindi, la tupla risultante viene restituita come risultato della trasformazione. Viene applicata una trasformazione reduceByKey. La trasformazione reduceByKey raggruppa gli elementi con la stessa chiave e applica una funzione di riduzione per combinare i valori corrispondenti. In questo caso, la funzione di riduzione somma i valori distance per ogni chiave.

```
In [46]: val totalDistanceByLicenseAndStartLoc = rddTaxiTrip.map(x =>
                                    ((x.license, x.startloc), x.distance)).reduceByKey((x, y) => x + y).take(10)
```

▸ Spark Job Progress

```
totalDistanceByLicenseAndStartLoc: Array[((String, Long), Double)] = Array(((B02888,11),1187.4000000000012), ((B02876,177),6032.130000000006), ((B02879,39),6840.850000000006), ((B02510,203),43056.412999999986), ((B02869,119),7910.020000000002), ((B02682,200),3115.9299999999985), ((B02395,14),3045.6600000000026), ((B02764,233),21495.780000000035), ((B02880,170),4044.1900000000002), ((B02882,76),15702.400000000032))
```

3.8 Tempo di viaggio medio, in base alla zona di partenza

Il codice esegue le seguenti operazioni:

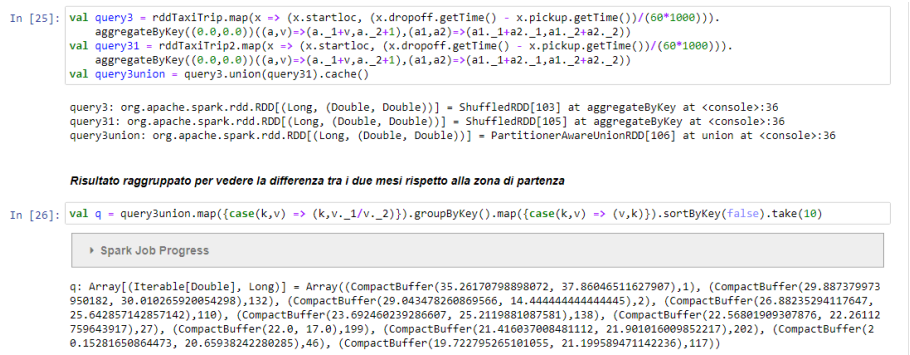
- Viene eseguita una map per estrarre la zona di partenza e il tempo di viaggio;

- Successivamente viene fatta un aggregazione in base alla chiave per andare a calcolare la somma dei tempi e il conteggio dei viaggi;

Queste azioni sono state svolte per entrambi gli RDD dei mesi.

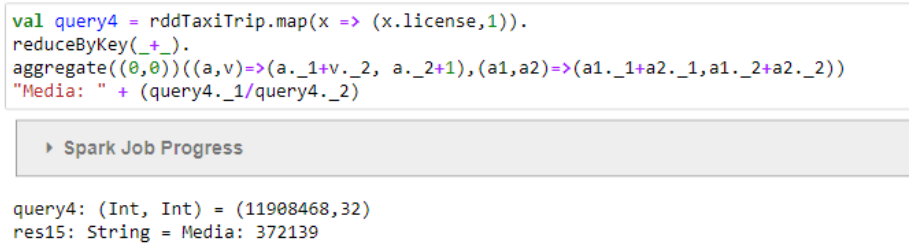
- Viene eseguita una union per raggruppare i due valori;

Il risultato finale che si sta cercando è quello di vedere per ogni zona quanto è il tempo medio di viaggio. Da notare che il risultato è raggruppato per vedere la differenza tra i due mesi rispetto alla zona di partenza.



3.9 In media quanti viaggi fa ciascuna licenza

Viene calcolato il numero medio di viaggi per licenza attraverso l'utilizzo di una serie di operazioni sui dati contenuti nel dataset. In particolare, vengono mappati gli identificativi dei viaggi e per ognuno di essi, viene creato un valore pari a 1. Questi valori vengono poi sommati insieme utilizzando la funzione "reduceByKey". Con la funzione "aggregate" si ottiene una struttara del (numero viaggi, numero licenze). Infine basta dividere il numero viaggi per numero licenze ed ottenere cos'ı la media dei viaggi per licenza.



3.10 Numero di viaggi effettuati per ogni combinazione di ora di inizio del viaggio e zona di arrivo

```
val NTrip = rddTaxiTrip.map(x => ((getDayTime(x.pickup.getTime()).get(Calendar.HOUR), x.endloc), 1)).
reduceByKey(_ + _).
map { case ((hour, endloc), count) => (count, (hour, endloc)) }.
sortByKey(false).
collect()
```

▸ Spark Job Progress

```
NTrip: Array[(Int, (Int, Long))] = Array((37314,(7,265)), (36103,(8,265)), (34085,(10,265)), (33517,(6,265)), (33365,(9,265)),
(31528,(11,265)), (29066,(5,265)), (29061,(0,265)), (26894,(1,265)), (26685,(2,265)), (25943,(4,265)), (25941,(3,265)), (20991,
(7,61)), (20621,(8,61)), (20286,(10,61)), (19643,(9,61)), (19287,(6,61)), (19081,(11,61)), (18057,(0,61)), (17079,(5,61)), (170
69,(8,76)), (16668,(7,76)), (16421,(1,61)), (15959,(2,61)), (15618,(4,61)), (15611,(3,61)), (15520,(11,76)), (15490,(9,76)), (1
5273,(6,76)), (14947,(10,76)), (14935,(2,76)), (14852,(3,76)), (14849,(5,76)), (14367,(7,37)), (14302,(7,42)), (14289,(8,42)),
(14129,(0,76)), (14023,(10,37)), (13843,(5,132)), (13819,(8,37)), (13782,(4,76)), (13693,(1,76)), (13310,(8,244)), (13193,(9,4
2)), (13146,(9,37)), (13115,(6,244))...
```

3.11 Classe di licenza che fa piu viaggi, con la presenza anche della tariffa cliente

Questo codice stampa i risultati delle classi di licenze che fa più viaggi in relazione anche al guadagno. In particolare, il codice mappa ogni elemento in una tupla in cui la chiave è la licenza, contenente il guadagno e il numero di viaggi di ogni licenza. Poi, i dati vengono aggregati tramite la funzione reduceByKey che somma i valori associati alla stessa chiave. Successivamente, il risultato viene mappato in una tupla contenente il nome della licenza, il guadagno totale e il numero di viaggi totali e viene ordinato in modo decrescente in base al guadagno. Infine, vengono stampati gli elementi della lista di risultati ottenuti dalla funzione collect().

```
val query5 = rddTaxiTrip.map(x => (x.licenseClass,(x.fare,1))).
reduceByKey((t1,t2) => (t1._1+t2._1, t1._2+t2._2)).
map(v => (v._1,v._2._1, v._2._2)).
sortBy(_._2,false).
collect().foreach(println(_))
```

▸ Spark Job Progress

```
(HV0003,1.4682788602879338E8,8704128)
(HV0005,5.5097683459859E7,3094325)
(HV0004,2816007.860000914,110015)
query5: Unit = ()
```

3.12 Licenza che fa piu viaggi, con la presenza anche della tariffa cliente

```
val query6 = rddTaxiTrip.map(x => (x.license,(x.fare,1))).
reduceByKey((t1,t2) => (t1._1+t2._1, t1._2+t2._2)).
map(v => (v._1,v._2._1, v._2._2)).
sortBy(_._2,false).
collect().foreach(println(_))
```

▸ Spark Job Progress

```
(B02510,5.503748608986006E7,3091000)
(B02764,1.709378720999917E7,1009388)
(B02872,1.5509604710001156E7,924960)
(B02875,1.2014741050004447E7,735450)
(B02765,1.0189792220006326E7,591242)
(B02869,7744772.440006703,452098)
(B02887,5608155.9400040135,333768)
(B02871,5581241.000004057,330085)
(B02682,5456731.730003582,321599)
(B02866,5284853.590003659,309274)
(B02864,5203100.810003353,316395)
(B02878,5125333.480003258,312013)
(B02617,4884453.930003084,281432)
(B02883,4582633.870002626,268391)
(B02884,4354322.950002186,257674)
(B02882,4097472.360001952,241988)
```

3.13 Licenza di taxi con il prezzo medio del viaggio più alto per ciascuna classe di licenza

In [30]:
```
val avgFareByLicenseClass = rddTaxiTrip.map(x => (x.licenseClass, x.fare)).groupByKey()
.mapValues(fares => fares.sum / fares.size).reduceByKey((fare1, fare2) => if (fare1 > fare2) fare1 else fare2)
```

```
avgFareByLicenseClass: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[134] at reduceByKey at <console>:35
```

In [31]:
```
avgFareByLicenseClass.collect()
```

▸ Spark Job Progress

```
res16: Array[(String, Double)] = Array((HV0004,25.59658101168853), (HV0005,17.806042823510747), (HV0003,16.86876457110848))
```

3.14 Distanza media percorsa dai taxi in un singolo viaggio, in relazione alla zona

```
In [32]: val query8 = rddTaxiTrip.filter(_.distance < 100).map(x => (x.startloc,x.distance)).
             aggregateByKey((0.0,0.0))((a,v)=>(a._1+v,a._2+1),(a1,a2)=>(a1._1+a2._1,a1._2+a2._2))
         val query81 = rddTaxiTrip2.filter(_.distance < 100).map(x => (x.startloc,x.distance)).
             aggregateByKey((0.0,0.0))((a,v)=>(a._1+v,a._2+1),(a1,a2)=>(a1._1+a2._1,a1._2+a2._2))

         query8: org.apache.spark.rdd.RDD[(Long, (Double, Double))] = ShuffledRDD[137] at aggregateByKey at <console>:36
         query81: org.apache.spark.rdd.RDD[(Long, (Double, Double))] = ShuffledRDD[140] at aggregateByKey at <console>:36

In [33]: val query8union = query8.union(query81).cache()

         query8union: org.apache.spark.rdd.RDD[(Long, (Double, Double))] = PartitionerAwareUnionRDD[141] at union at <console>:36

In [34]: val q8 = query8union.map({case(k,v) => (k,v._1/v._2)}).map({case(k,v) => (v,k)}).sortByKey(false).take(10)

         ▸ Spark Job Progress

         q8: Array[(Double, Long)] = Array((20.136333333333326,1), (19.978372093023246,1), (15.583862644195618,132), (14.86465207208320
         6,132), (11.889173913043477,2), (10.915017273061434,138), (10.765875894988065,27), (10.74933302682418,138), (10.515,199), (10.5
         07823529411766,110))
```

3.15 Zona di arrivo più comune per ciascuna licenza di taxi

- Viene applicata una trasformazione map all'RDD. Per ogni elemento viene creato una tupla con la chiave composta da license e endloc e il valore 1. Questo attribuisce un valore di 1 a ogni combinazione unica di license e endloc.
- La trasformazione reduceByKey raggruppa gli elementi con la stessa chiave e applica una funzione di riduzione per combinare i valori corrispondenti.
- Viene applicata una trasformazione map successiva all'RDD risultante dalla trasformazione reduceByKey.
- Successivamente la trasformazione map estrae la chiave license e crea una nuova tupla con la chiave license e il valore composto da una tupla contenente endloc e count. Viene applicata una trasformazione groupByKey all'RDD risultante dalla trasformazione precedente.
- La trasformazione groupByKey raggruppa gli elementi con la stessa chiave e restituisce un RDD di coppie chiave-valore, dove il valore è un iterabile dei valori corrispondenti alla chiave. Viene applicata una trasformazione mapValues all'RDD risultante dalla trasformazione groupByKey.
- La trasformazione mapValues applica una funzione a tutti i valori di ogni coppia chiave-valore. In questo caso, la funzione seleziona il valore massimo di count per ogni chiave, restituendo la tupla (endloc, count) corrispondente a quel valore massimo.

In sintesi, la query calcola la combinazione di license e endloc più comune (con il numero massimo di occorrenze) per ogni license. Restituisce un RDD con la chiave license e il valore endloc corrispondente alla combinazione più comune.

```
[37]: val mostCommonEndLocByLicense = rddTaxiTrip.map(x => ((x.license, x.endloc), 1)).reduceByKey(_ + _)
      .map { case ((license, endloc), count) => (license, (endloc, count)) }.groupByKey().mapValues { locs => locs.maxBy(_._2)._1 }

      mostCommonEndLocByLicense: org.apache.spark.rdd.RDD[(String, Long)] = MapPartitionsRDD[153] at mapValues at <console>:37

[38]: mostCommonEndLocByLicense.collect()

      ▸ Spark Job Progress

      res30: Array[(String, Long)] = Array((B02682,265), (B02871,61), (B02835,265), (B02877,265), (B02880,265), (B02844,265), (B0239
      5,265), (B02872,265), (B02836,265), (B02887,265), (B03136,140), (B02764,265), (B02869,265), (B02878,265), (B02879,265), (B0286
      4,265), (B02765,265), (B02888,265), (B02510,265), (B02882,265), (B02889,265), (B02883,265), (B02865,265), (B02884,265), (B0251
      2,265), (B02617,265), (B02800,166), (B02866,265), (B02875,265), (B02867,265), (B02870,265), (B02876,265))

[39]: import org.apache.spark.sql.SaveMode
      val mostCommonEndLocDF = mostCommonEndLocByLicense.toDF("license", "endloc")
      mostCommonEndLocDF.write.format("csv").mode(SaveMode.Overwrite).save("s3a://"+bucketname+"/datasets/project/output/MaxLoc")
```

3.16 Analisi sul guadagno medio di ogni singolo viaggio per ogni giorno

Il codice ha lo scopo di calcolare il guadagno medio di ogni classe di licenza in base al giorno in cui si è svolto il viaggio. Il risultato è stato poi salvato in un file CSV. Sono state eseguite le seguenti operazioni:

- Viene eseguito creato un RDD (avgFareByDay) che associa a ciascuna coppia (classe di licenza, Giorno) una tupla contenente la somma del guadagno e il numero di viaggi durante il giorno.
- Viene quindi eseguita una riduzione per chiave (reduceByKey) per aggregare i dati raggruppati per classe di licenza e giorno.
- Successivamente, viene eseguita una mappatura (mapValues) per calcolare il guadagno medio di ogni viaggio per ogni giorno.
- In seguito, viene eseguita una mappatura (map) per riorganizzare i dati e raggrupparli per classe di licenza.
- Vengono poi eseguite ulteriori operazioni di trasformazione (flatMap e coalesce) per preparare i dati per il salvataggio su file CSV.
- Infine, viene convertito l'RDD in un DataFrame e salvato in un file CSV.

```
: val avgFareByDay = rddTaxiTrip.map(x => ((x.licenseClass, getDayTime(x.pickup.getTime()).get(Calendar.DAY_OF_WEEK)), (x.fare, 1))
    reduceByKey((accum, value) => (accum._1 + value._1, accum._2 + value._2)).
    mapValues(sumCount => sumCount._1 / sumCount._2).
    map(item => (item._1._1, item._2, item._1._2))
    //collect() //(licenza,avg,giorno)

  avgFareByDay: org.apache.spark.rdd.RDD[(String, Double, Int)] = MapPartitionsRDD[159] at map at <console>:42

: // Raggruppo per licenza
  val avgFareByLicense = avgFareByDay.groupBy(_._1).
    mapValues(_.map(item => (item._2, item._3)))//.collect()

  avgFareByLicense: org.apache.spark.rdd.RDD[(String, Iterable[(Double, Int)])] = MapPartitionsRDD[162] at mapValues at <console
  >:40
```

```
//flatmap con chiave prodotto e anno
val avgFareByLicenseAndDay = avgFareByLicense.flatMap {
    case (license, dayFare) => dayFare.map {
        case (avgFare, day) => ((license, day),avgFare)
  }
}.coalesce(1)
```

```
avgFareByLicenseAndDay: org.apache.spark.rdd.RDD[((String, Int), Double)] = CoalescedRDD[164] at coalesce at <console>:43
```

```
import org.apache.spark.sql.SaveMode
// Conversione in DataFrame con stringa
val df = avgFareByLicenseAndDay.map{case ((license, day), avgFare) => (license,day,avgFare)}.toDF()

// Salvataggio su file CSV
df.write.format("csv").mode(SaveMode.Overwrite).save("s3a://"+bucketname+"/datasets/project/output/avgFare")
```

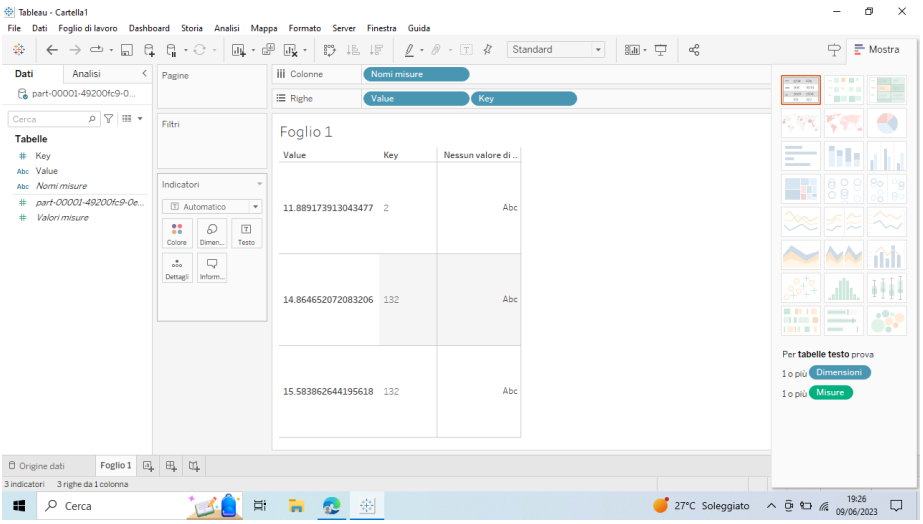
▸ Spark Job Progress

```
import org.apache.spark.sql.SaveMode
df: org.apache.spark.sql.DataFrame = [_1: string, _2: int ... 1 more field]
```

4 Risultati ottenuti e Tableau

4.1

Il primo risultato è della query "Distanza media percorsa dai taxi in un singolo viaggio, in relazione alla zona" ossia la 3.14.



4.2

Il secondo risultato è della query "Analisi sul guadagno medio di ogni singolo viaggio per ogni giorno" ossia la 3.16.