Danilo Dell'Orco
Jacopo Fabi
Michele Salvatori

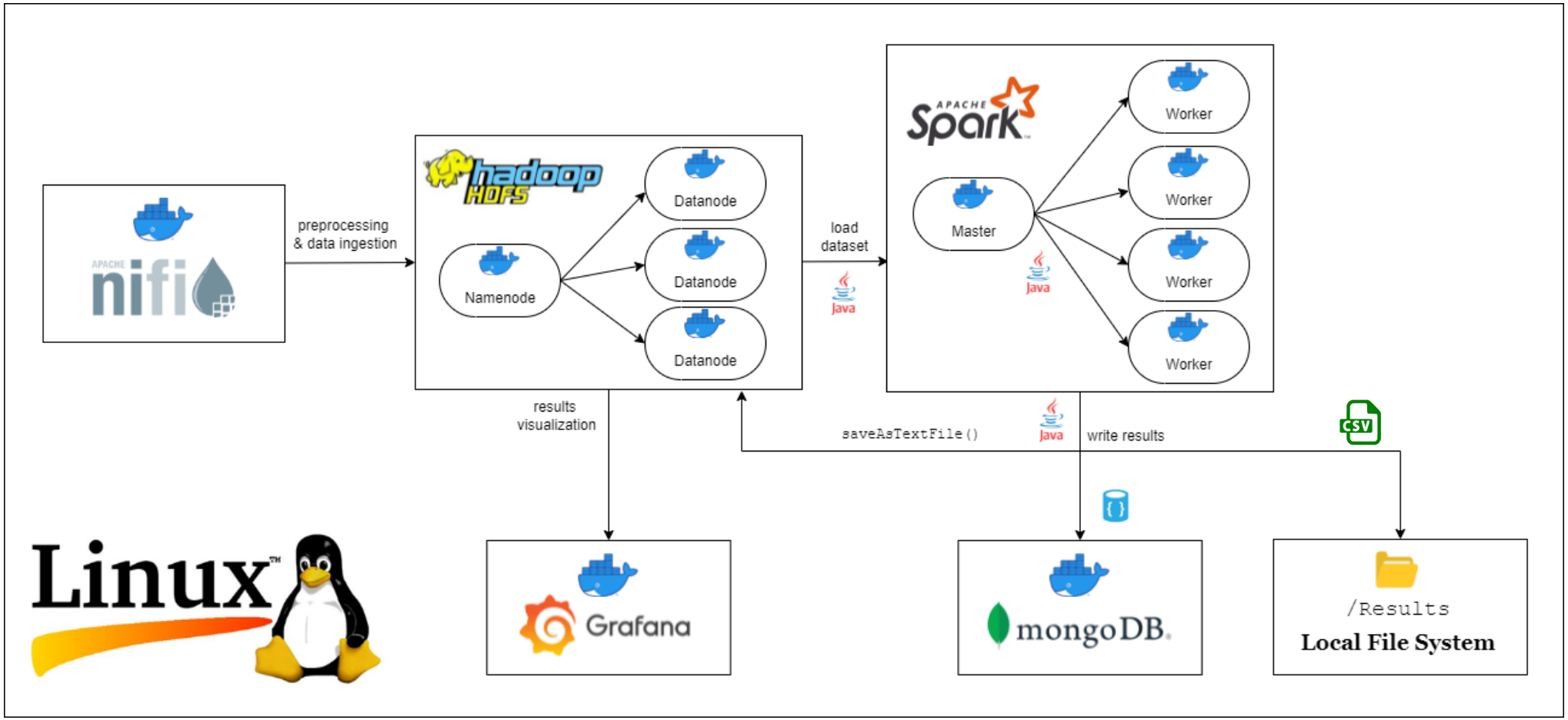
SABD – Progetto 1

ANALISI DEL DATASET DEI TAXI DI NYC

Introduzione

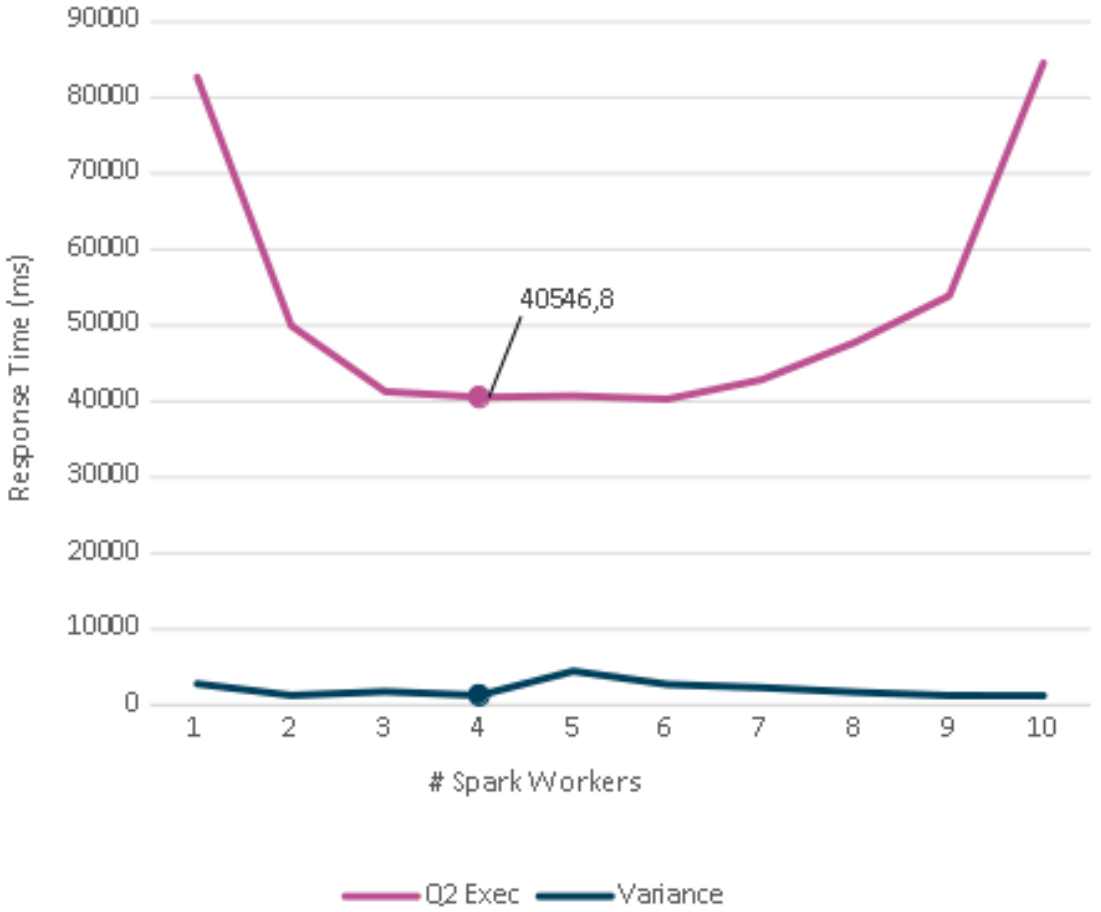
- L’obiettivo del progetto è quello di analizzare i dati forniti dal Taxi and Limousine Commission (TLC) riguardo le corse effettuate dai taxi, in particolare utilizzando il dataset *“TLC Trip Record Data”* fornito in formato parquet.
- Si applica tramite NiFi una fase di pre-processamento e caricamento del dataset su HDFS, utilizzato come layer di storage.
- L’analisi e l’esecuzione delle query avviene tramite Spark, sia sfruttando l'RDD API di più basso livello sia tramite l'astrazione offerta da SparkSQL.
- I risultati delle query vengono inseriti su MongoDB e salvati anche in formato CSV su HDFS.
- In aggiunta, i risultati ottenuti dall'esecuzione delle diverse query vengono resi visualizzabili tramite una dashboard offerta dal framework Grafana.

Architettura



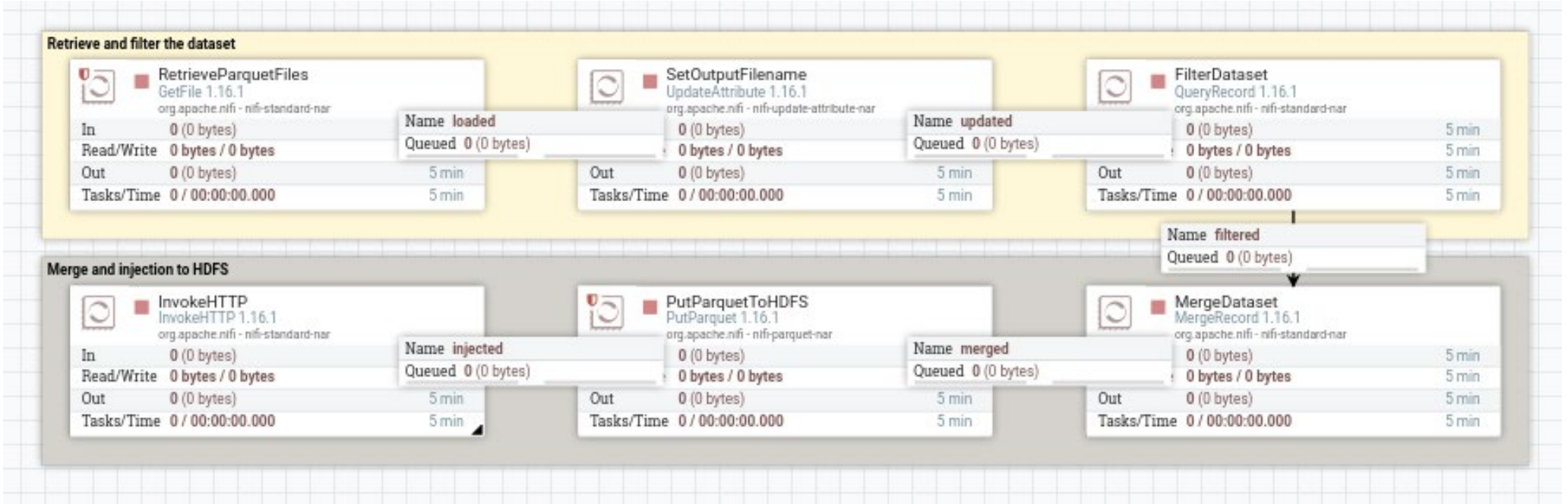
Architettura (2)

- Il numero di nodi workers Spark è stato scelto testando il tempo di risposta al variare del loro numero in esecuzione.
- La valutazione è basata sulla query 2 essendo la più complessa.
- Sono state effettuate cinque iterazioni per ogni configurazione del numero di workers.
- Consideriamo il valore della media e della varianza per identificare la configurazione ideale.



NiFi

- Il flusso di **NiFi** è stato progettato per rimuovere dal dataset eventuali dati corrotti e colonne inutili non necessarie nella fase di processamento tramite Spark.



NiFi (2)

Il dataset viene filtrato nel processore `QueryRecord`:

- Vengono eliminate le righe che presentano campi d'interesse con valori negativi.
- Vengono eliminate le righe che presentano come timestamp di fine corsa un valore che non rientra nei tre mesi in analisi.
- Vengono selezionate solamente le colonne necessarie per rispondere alle diverse query.

```
SELECT tpep_pickup_datetime, tpep_dropoff_datetime, PULocationID, DOLocationID,
payment_type, fare_amount, tip_amount, tolls_amount, total_amount, passenger_count
FROM FLOWFILE
WHERE (payment_type>0 AND tip_amount>=0 AND tolls_amount>=0 AND total_amount>=0
       AND fare_amount>=0 AND passenger_count>=0
       AND (tpep_dropoff_datetime BETWEEN '2021-12-01 00:00:01' AND '2022-02-28 23:59:59'))
```

NiFi (3)

Il processore `MergeRecord` viene utilizzato per unire più flowfiles in un singolo flowfile:

- L'obiettivo è scrivere su `HDFS` un singolo file parquet che rappresenta il dataset finale.
- Reso bloccante finché non sono stati uniti tutti i records in un singolo flowfile di output.
- Il numero massimo e minimo di records è impostato al numero totale di righe del dataset in output alla fase di filtraggio così da produrre una singola `bin`.

Property	Value	
Record Reader	**ParquetReader**	→
Record Writer	**ParquetRecordSetWriter**	→
Merge Strategy	**Bin-Packing Algorithm**	
Correlation Attribute Name	No value set	
Attribute Strategy	**Keep Only Common Attributes**	
Minimum Number of Records	**8336885**	
Maximum Number of Records	8336885	

NiFi (4)

Il processore `InvokeHTTP` effettua una `POST` all'indirizzo della macchina che ospita la rete dei vari containers:

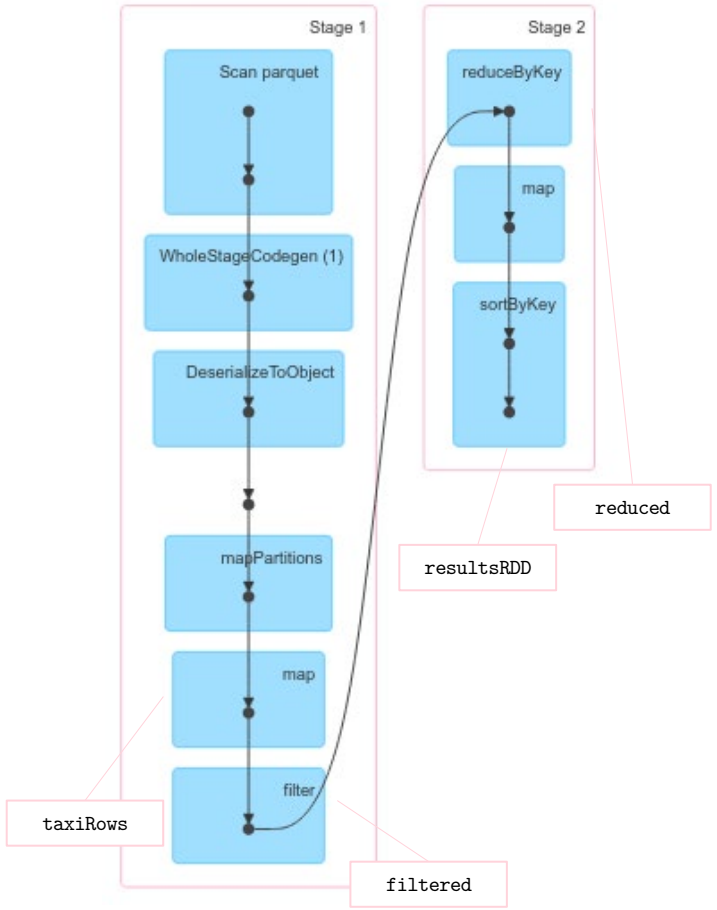
- Lo script seguente avvia `HDFS` e ne effettua il format, lancia il flusso di NiFi e si mette in ascolto sulla porta 5555.
- Al termine del flusso il server riceverà la `POST` dal processore.
- Si procedere ripristinando il flusso per una nuova esecuzione stoppandolo tramite le `REST API` offerte da Apache.

```
from http.server import BaseHTTPRequestHandler, HTTPServer
import subprocess
import urllib.request
import signal
import os

class handler(BaseHTTPRequestHandler):
    def do_POST(self):
        # stop nifi flow
        subprocess.call(['sh', './stop-nifi-flow.sh'])
        os.kill(os.getpid(), signal.SIGTERM)

with HTTPServer(('172.17.0.1', 5555), handler) as server:
    subprocess.call(['sh', './start-hdfs.sh'])

    # start nifi flow when the web UI is up and running
    message = 0
    while (message != 200):
        try:
            message = urllib.request.urlopen("http://localhost:8090/nifi")
                      .getcode()
            print("Nifi Web UI is up and running!")
            print(message)
        except:
            print("Nifi not running...")

    subprocess.call(['sh', './start-nifi-flow.sh'])
    server.serve_forever()
```

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.»

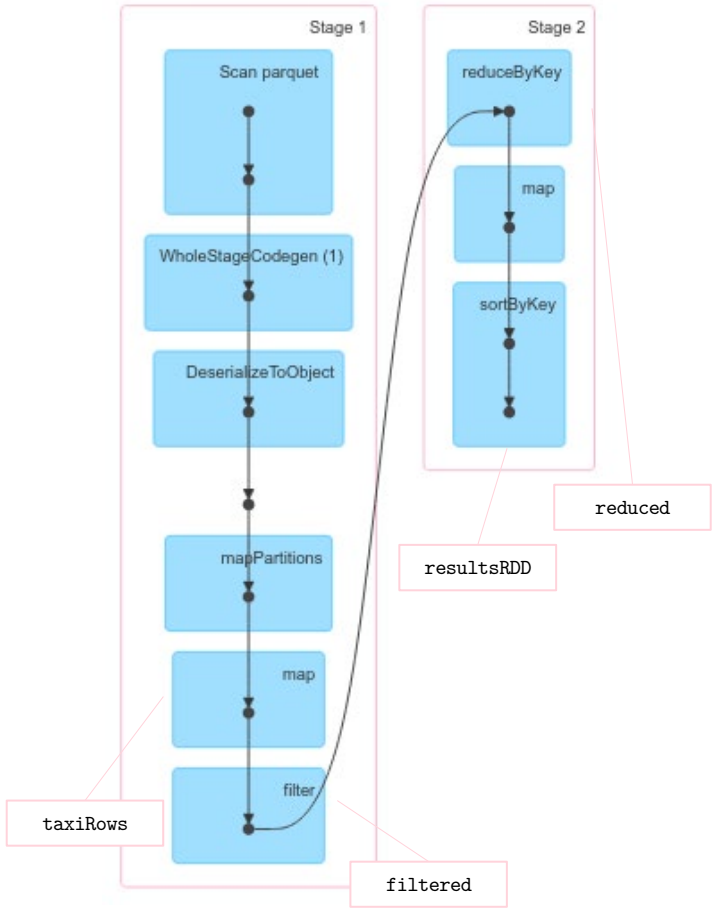
SparkRDD – Query1




```
public class ValQ1 implements Serializable {
   Double tip_amount;
   Double total_amount;
   Double tolls_amount;
   Long payment_type;
   Integer trips_number;
...
```

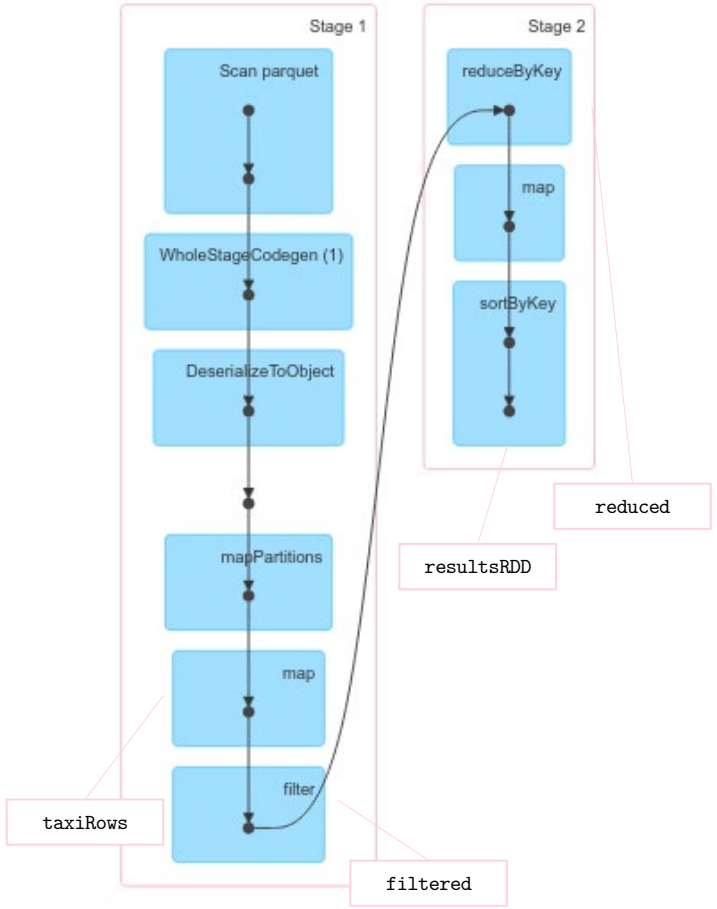
```
JavaPairRDD<String, ValQ1> taxiRows = dataset.mapToPair(
      r -> {
          String month = Tools.getMonth(r.getTimestamp(1));
          ValQ1 v1 = new ValQ1(r.getDouble(6), r.getDouble(8), r.getDouble(7), r.getLong(4), 1);
          return new Tuple2<>(month, v1);
      });

JavaPairRDD<String, ValQ1> filtered = taxiRows.filter(
            (Function<Tuple2<String, ValQ1>, Boolean>) r -> r._2().getPayment_type() == 1);
```

SparkRDD – Query1




```
// RDD:=[month,values_aggr]
JavaPairRDD<String, ValQ1> reduced = filtered.reduceByKey((Function2<ValQ1, ValQ1, ValQ1>)
      (v1, v2) -> {
            Double tips = v1.getTip_amount() + v2.getTip_amount();
            Double total = v1.getTotal_amount() + v2.getTotal_amount();
            Double tolls = v1.getTolls_amount() + v2.getTolls_amount();
            Integer trips = v1.getTrips_number() + v2.getTrips_number();

            ValQ1 v = new ValQ1();
            v.setTip_amount(tips);
            v.setTotal_amount(total);
            v.setTolls_amount(tolls);
            v.setTrips_number(trips);
            return v;
        });

// RDD:=[month,tip_percentage,trips_number]
JavaPairRDD<String, Tuple2<Double, Integer>> resultsRDD = reduced.mapToPair(
        r -> {
            Double tips = r._2().getTip_amount();
            Double tolls = r._2().getTolls_amount();
            Double total = r._2().getTotal_amount();
            Double mean = tips / (total - tolls);
            Integer trips = r._2().getTrips_number();
            return new Tuple2<>(r._1(), new Tuple2<>(mean, trips));
            })
        .sortByKey();
```

SparkRDD – Query1: Risultato Finale



YYYY-MM	tip percentage	trips number
2021-12	0.15973426958603487	2361627
2022-01	0.15624681545577337	1874779
2022-02	0.15672618046552844	2296156

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»

SparkRDD – Query 2: Tuple

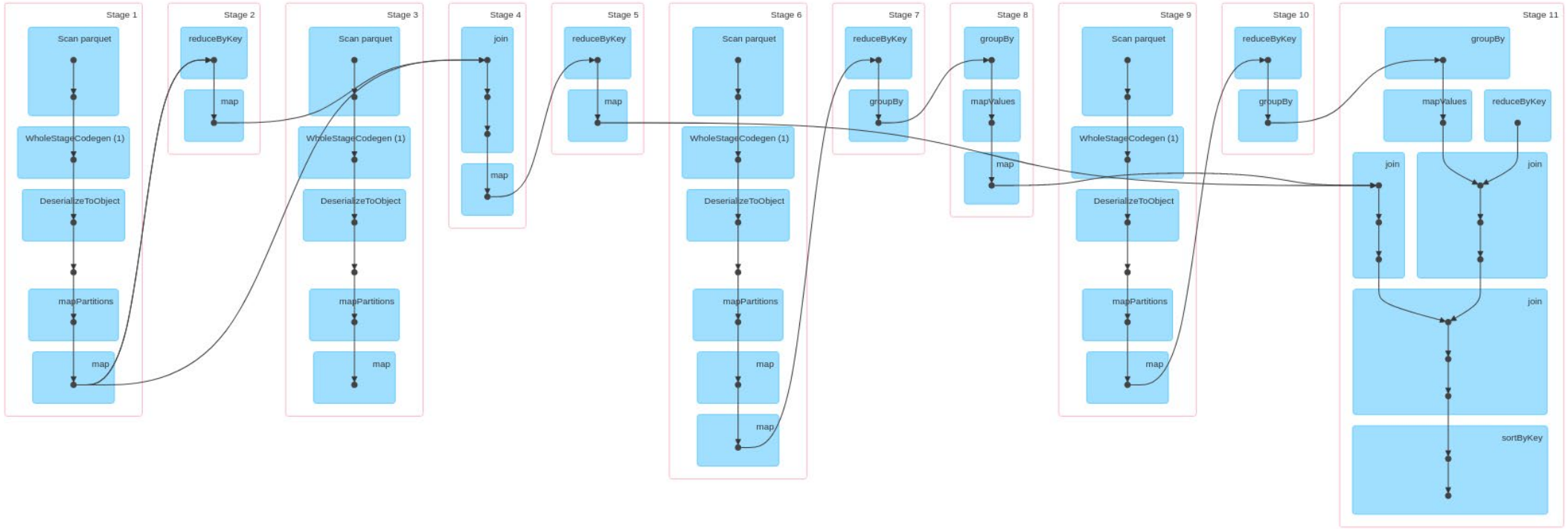
- Aggregazione su base oraria:
  - `tpep_pickup_datetime`
- Distribuzione dei viaggi per ogni ora
  - `PULocationID`
- Media e deviazione standard delle mance
  - `tip_amount`
- Metodo di pagamento preferito
  - `payment_type`

```
public class ValQ2 implements Serializable {
    Double tips;
    Integer num_payments;
    Integer num_trips;
    Long payment_type;
    Double tips_stddev;
}
```

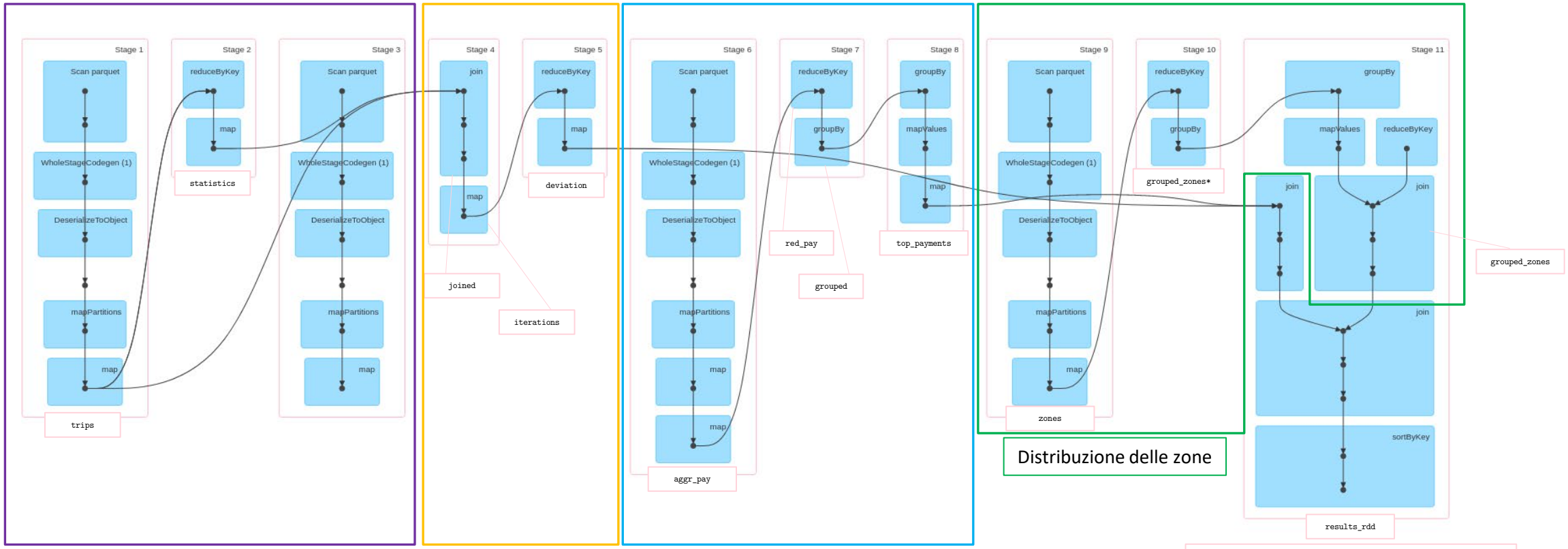
```
public class KeyQ2PU implements Serializable {
    String hour;
    Long source;
}
```

```
public class KeyQ2Pay implements Serializable {
    String hour;
    Long payment;
}
```

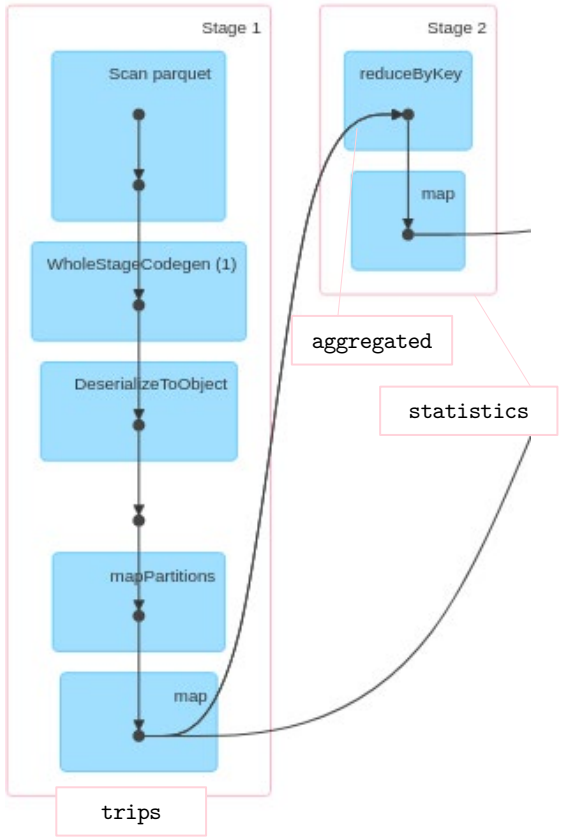
SparkRDD – Query2



SparkRDD – Query2

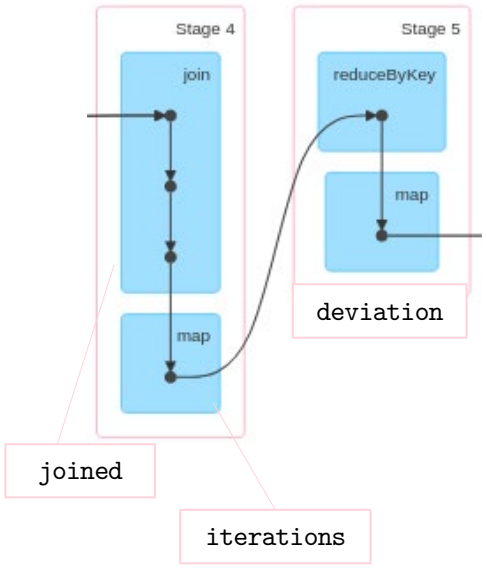


SparkRDD – Query2: Media Mance



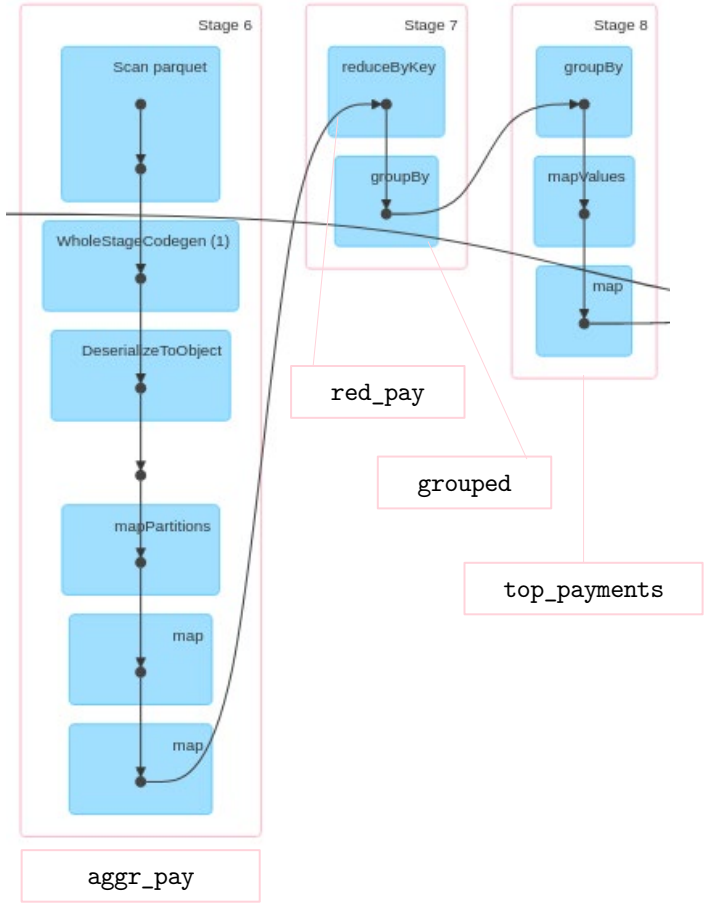

```
// RDD:=[hour,statistics]
JavaPairRDD<String, ValQ2> trips = dataset.mapToPair(r ->
      new Tuple2<>(Tools.getHour(r.getTimestamp(0)),
                  new ValQ2(r.getDouble(6), r.getLong(4), 1, 1))); //tips,payment_type


JavaPairRDD<String, ValQ2> aggregated = trips.reduceByKey((Function2<ValQ2, ValQ2, ValQ2>) (v1, v2) -> {
           ValQ2 v = new ValQ2();
           Integer aggr = v1.getNum_trips() + v2.getNum_trips();
           Integer occ = v1.getNum_payments() + v2.getNum_payments();
           Double tips = v1.getTips() + v2.getTips();
           v.setNum_trips(aggr);
           v.setNum_payments(occ);
           v.setTips(tips);
           return v;
        });

JavaPairRDD<String, ValQ2> statistics = aggregated.mapToPair(
      r -> {
            Integer num_occurrences = r._2().getNum_payments();
            Double tips_mean = r._2().getTips() / num_occurrences;

            return new Tuple2<>(r._1(),
                    new ValQ2(tips_mean, num_occurrences));
      });
```

SparkRDD – Query2: Deviazione Standard



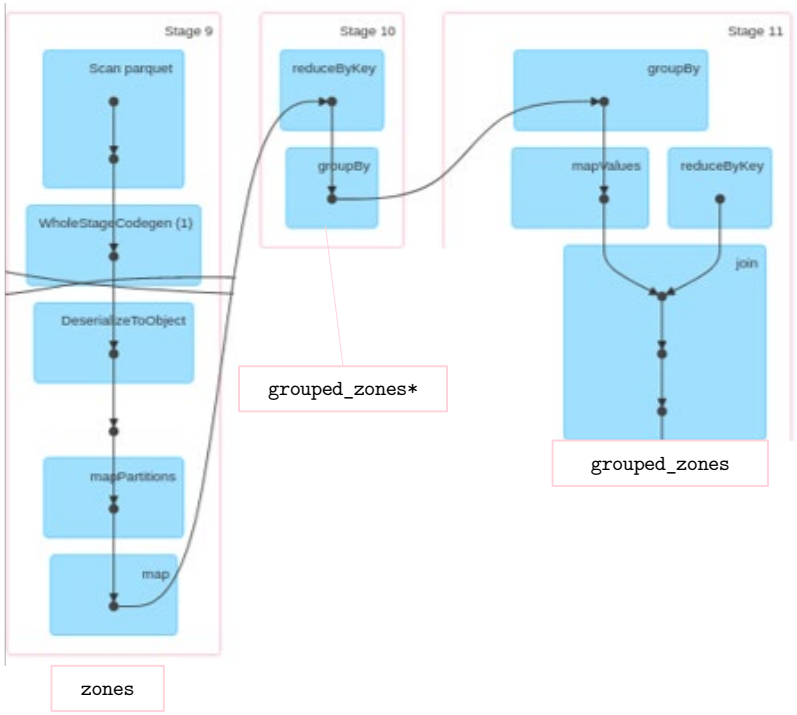

```
// RDD:=[hour,statistics_deviation_it]
JavaPairRDD<String, ValQ2> iterations = joined.mapToPair( //joined = trips.join(statistics)
        r -> {
            Double tips_mean = r._2()._2().getTips();
            Double tip_val = r._2()._1().getTips();
            Double tips_dev = Math.pow((tip_val - tips_mean), 2);
            r._2()._2().setTips_stddev(tips_dev);

            return new Tuple2<>(r._1(), r._2()._2());
        });

// RDD:=[hour,statistics_deviation_sum]
JavaPairRDD<String, ValQ2> stddev_aggr = iterations.reduceByKey((Function2<ValQ2, ValQ2, ValQ2>) (v1, v2) -> {
    Double tips_total_stddev = v1.getTips_stddev() + v2.getTips_stddev();
    ValQ2 v = new ValQ2(v1.getTips(), v1.getNum_payments(), v1.getPayment_type(), tips_total_stddev);
    return v;
});

// RDD:=[hour,statistics_deviation]
JavaPairRDD<String, ValQ2> deviation = stddev_aggr.mapToPair(
        r -> {
            Double tips_mean = r._2().getTips();
            Integer n = r._2().getNum_payments();
            Double tips_dev = Math.sqrt(r._2().getTips_stddev() / n);
            ValQ2 v = new ValQ2(tips_mean, n, tips_dev);
            return new Tuple2<>(r._1(), v);
        });
```

SparkRDD – Query2: Metodo di Pagamento



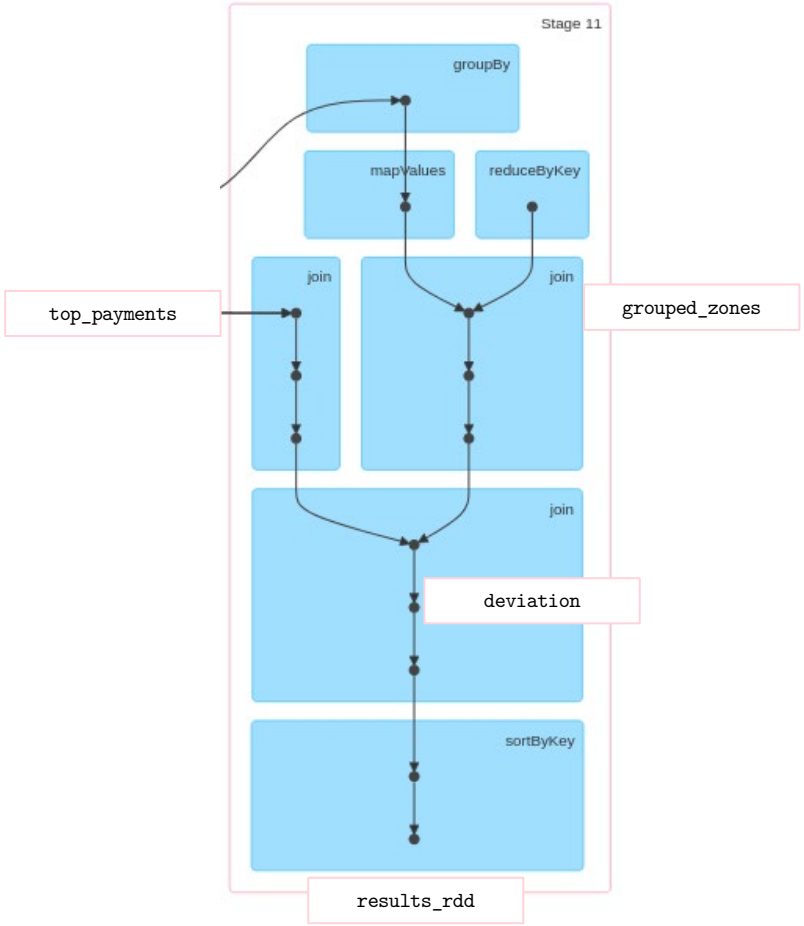

```
        // RDD:=[(hour,payment),1]
        JavaPairRDD<KeyQ2Pay, Integer> aggr_pay = trips.mapToPair(r ->
                new Tuple2<>(
                        new KeyQ2Pay(r._1(),r._2().getPayment_type())
                        ,1));

        // RDD:=[(hour,payment),occurrences]
        JavaPairRDD<KeyQ2Pay, Integer> red_pay = aggr_pay.reduceByKey((Function2<Integer, Integer, Integer>)
(v1, v2) -> {
            Integer occ = v1 + v2;
            return occ;
        });

        // RDD:=[hour,{((hour,payment),occurrences]
        JavaPairRDD<String, Iterable<Tuple2<KeyQ2Pay, Integer>>> grouped =
red_pay.groupBy((Function<Tuple2<KeyQ2Pay, Integer>, String>) r -> r._1().getHour());

        // RDD:=[hour,(top_payment,occurrences)]
        JavaPairRDD<String, Long> top_payments = grouped.mapToPair(r ->
                new Tuple2<>(
                        r._1(),
                        getMostFrequentPayment(r._2())
                ));
```

SparkRDD – Query2: Distribuzione Zone




```java
// RDD:=[(hour,PU),1]
JavaPairRDD<KeyQ2PU, Integer> zones = dataset.mapToPair(r ->
      new Tuple2<>(new KeyQ2PU(Tools.getHour(r.getTimestamp(0)), r.getLong(2)), 1));

// RDD:=[(hour,PU),occurrences]
JavaPairRDD<KeyQ2PU, Integer> red_zones = zones.reduceByKey((Function2<Integer, Integer, Integer>)
(v1, v2) -> {
      Integer occ = v1 + v2;
      return occ;
});

// RDD:=[hour,((hour,PU),occurrences)]
JavaPairRDD<String, Tuple2<Iterable<Tuple2<KeyQ2PU, Integer>>, ValQ2>> grouped_zones = red_zones
.groupBy((Function<Tuple2<KeyQ2PU, Integer>, String>) r -> r._1().getHour())
.join(aggregated);

…

Iterable<Tuple2<KeyQ2PU, Integer>> occList = r._2()._2()._1();
List<Tuple2<Double, Long>> percentages = calcPercentagesList(occList, totalTrips);
```

SparkRDD – Query2: Risultato Finale




```java
// RDD:=[hour,payment_stats,trips_stats]
JavaPairRDD<String, Tuple2<Tuple2<ValQ2, Long>, Tuple2<Iterable<Tuple2<KeyQ2PU, Integer>>, ValQ2>>>
final_joined = deviation
        .join(top_payments)
        .join(grouped_zones)
        .sortByKey(new DateComparator());

// RDD:= [hour, List<percentages>, avgTip, devTip, topPayment]
JavaPairRDD<String, Tuple4<List<Tuple2<Double, Long>>, Double, Double, Long>> results_rdd = final_joined
        .mapToPair(
        r -> {
                String hour = r._1();
                Long topPayment = r._2()._1()._2;
                double avgTip = r._2()._1()._1().getTips();
                double devTip = r._2()._1()._1().getTips_stddev();
                Integer totalTrips = r._2()._2()._2().getNum_trips();
                Iterable<Tuple2<KeyQ2PU, Integer>> occList = r._2()._2()._1();
                List<Tuple2<Double, Long>> percentages = calcPercentagesList(occList, totalTrips);

                return new Tuple2<>(hour, new Tuple4<>(percentages, avgTip, devTip, topPayment));
        });
```

SparkRDD – Query2: Risultato Finale

YYYY-MM-DD HH	...	perc_PU10	perc_PU11	perc_PU12	perc_PU13	perc_PU14	...	avg_tip	stddev_tip	pref_payment
2021-12-01-06	...	9,57854E+11	0	4,78927E+11	4,78927E+15	0	...	2,755987	3,542697	1
2021-12-01-07	...	7,02083E+11	0	2,34028E+11	6,08472E+15	0	...	2,493251	2,703577	1
2021-12-01-08	...	1,81061E+11	0	3,62122E+11	7,78562E+15	0	...	2,43526	2,445842	1
2021-12-01-09	...	1,67196E+12	0	1,67196E+12	4,84869E+15	0	...	2,495586	2,594521	1

Query 3

«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). Nell'output della query, indicare il nome della zona (TLC Taxi Destination Zone) anziché il codice numerico.»

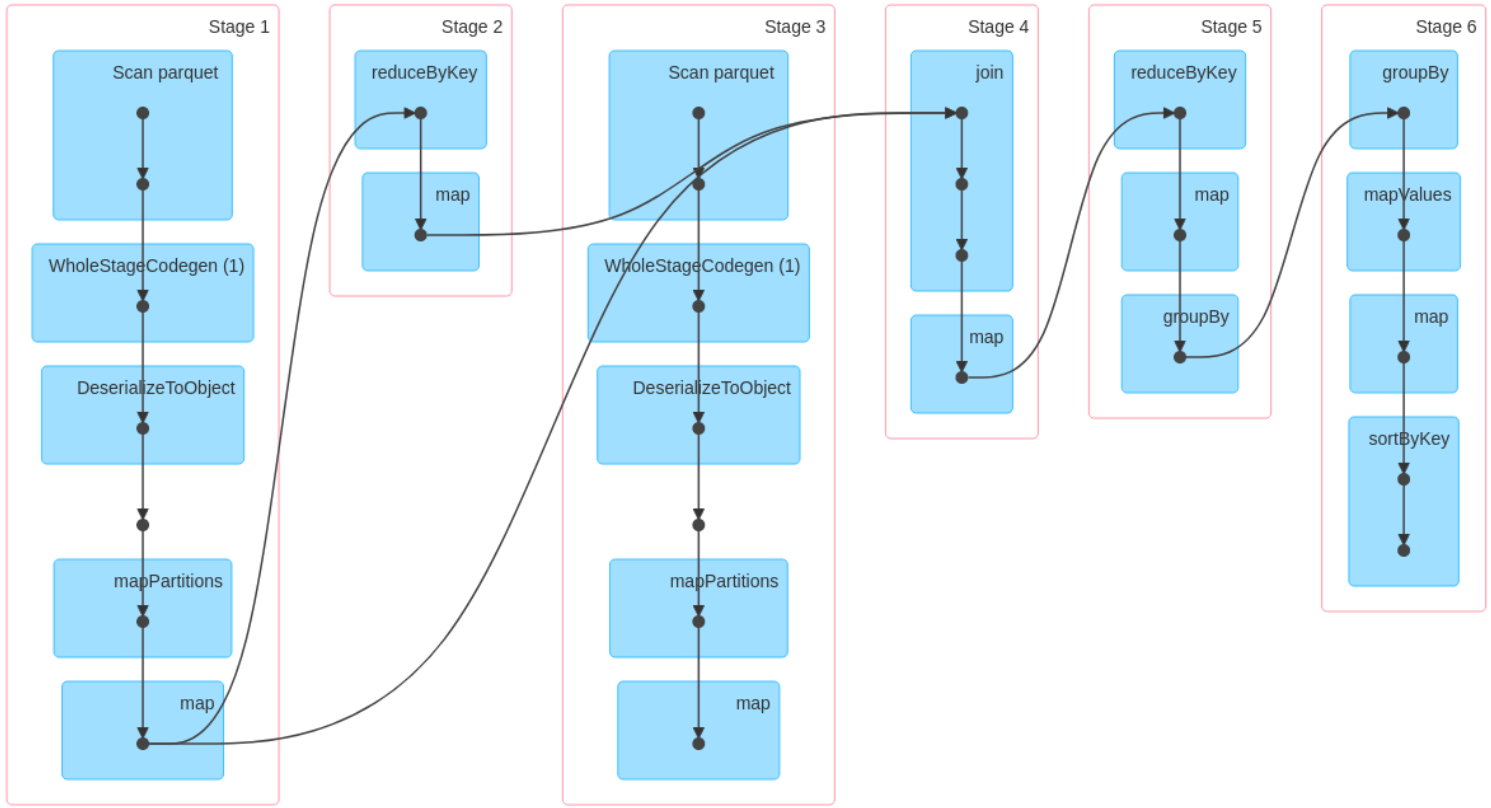
SparkRDD – Query 3: Tuple

- Identificazione delle cinque zone di destinazione più popolari
  - `DOLocationID`
- Aggregazione su base giornaliera
  - `tpep_dropoff_datetime`
- Media e deviazione standard della tariffa pagata
  - `fare_amount`
- Media del numero di passeggeri
  - `passenger_count`

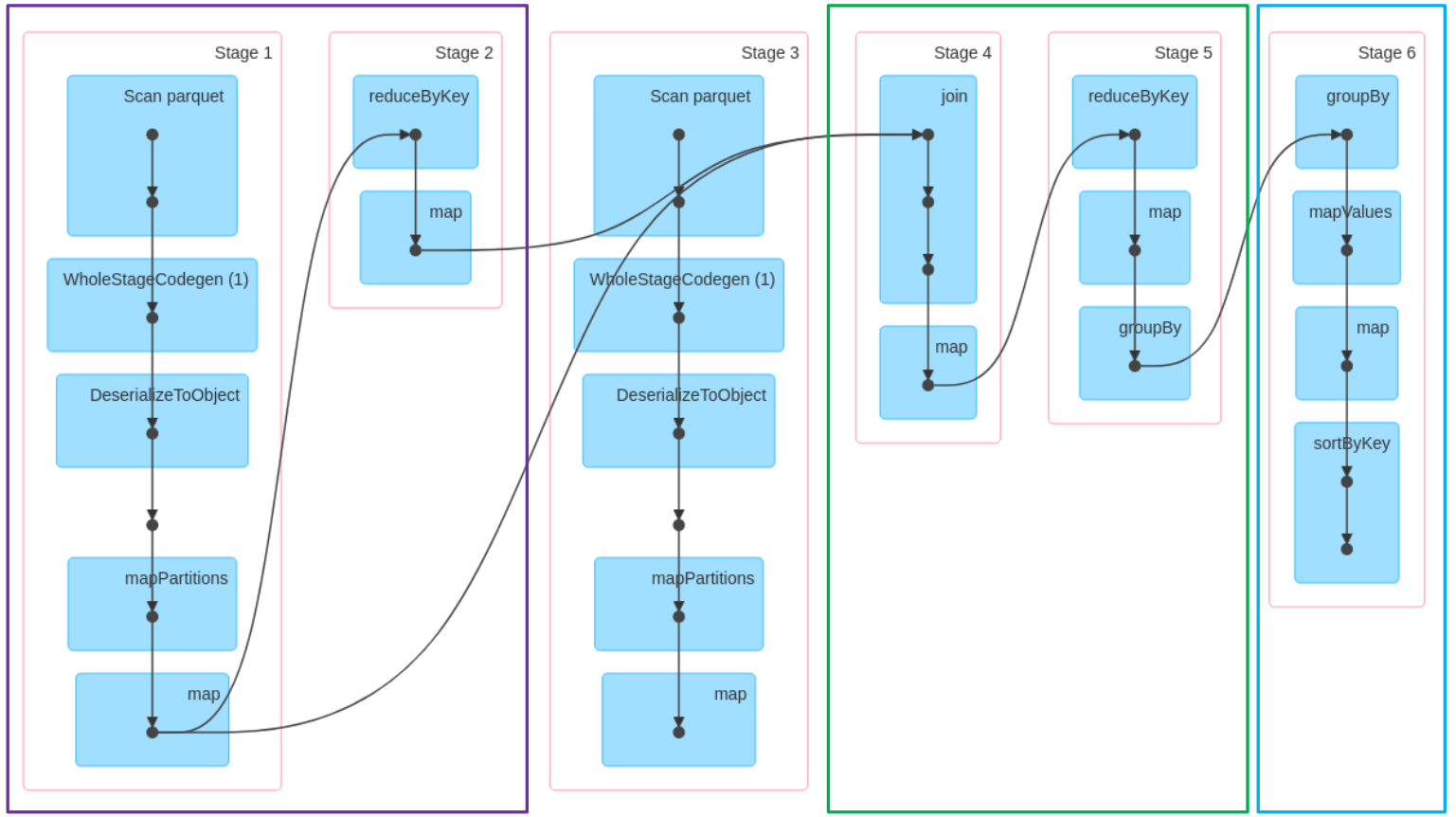
```
public class ValQ3 implements Serializable {
    Double passengers;
    Double fare;
    Integer occurrences;
    Double fare_stddev;
}
```

```
public class KeyQ3 implements Serializable {
    String day;
    Long dest;
}
```

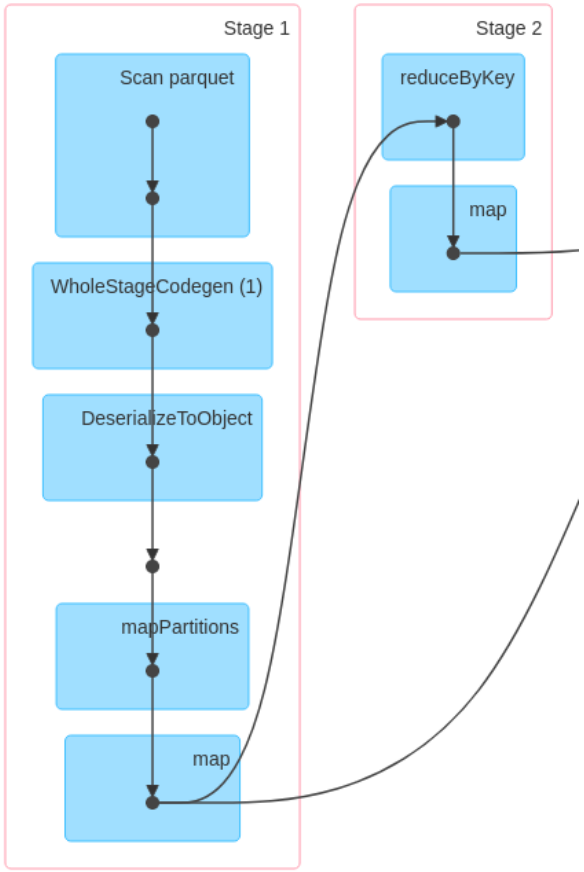
SparkRDD – Query3



SparkRDD – Query3

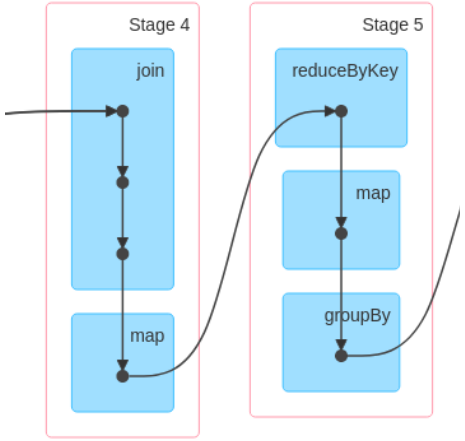


SparkRDD – Query3: Medie




```
        // RDD:=[(day,DO),statistics]
        JavaPairRDD<KeyQ3, ValQ3> days = dataset.mapToPair(
                r -> new Tuple2<>(new KeyQ3(Tools.getDay(r.getTimestamp(1)), r.getLong(3)),
                        new ValQ3(r.getDouble(9), r.getDouble(5), 1)));

        // RDD:=[(day,DO),statistics_aggregated]
        JavaPairRDD<KeyQ3, ValQ3> reduced = days.reduceByKey((Function2<ValQ3, ValQ3,
ValQ3>)  (v1, v2) -> {
            Double pass = v1.getPassengers() + v2.getPassengers();
            Double fare = v1.getFare() + v2.getFare();
            Integer occ = v1.getOccurrences() + v2.getOccurrences();
            ValQ3 v = new ValQ3(pass, fare, occ);
            return v;
        });

        // RDD:=[(day,DO),statistics_mean]
        JavaPairRDD<KeyQ3, ValQ3> mean = reduced.mapToPair(
                r -> {
                    Integer num_occurrences = r._2().getOccurrences();
                    Double pass_mean = r._2().getPassengers() / num_occurrences;
                    Double fare_mean = r._2().getFare() / num_occurrences;

                    return new Tuple2<>(r._1(),
                            new ValQ3(pass_mean, fare_mean, num_occurrences));
                });
```

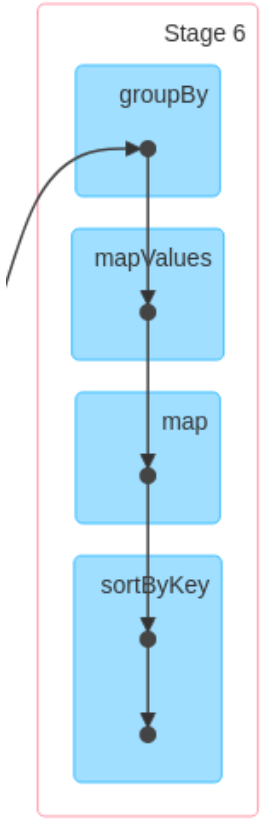
SparkRDD – Query3: Deviazione Standard




```
        // RDD:=[(day,DO),(statistics,statistics_mean)]
        JavaPairRDD<KeyQ3, Tuple2<ValQ3, ValQ3>> joined = days.join(mean);

        // RDD:=[(day,DO),statistics_stddev_iteration]
        JavaPairRDD<KeyQ3, ValQ3> iterations = joined.mapToPair(
                r -> {
                    Double fare_mean = r._2()._2().getFare();
                    Double fare_val = r._2()._1().getFare();
                    Double fare_dev = Math.pow((fare_val - fare_mean), 2);
                    r._2()._2().setFare_stddev(fare_dev);

                    return new Tuple2<>(r._1(), r._2()._2());
                });
```

```
      // RDD:=[(day,DO)),statistics_stddev_aggregated]
      JavaPairRDD<KeyQ3, ValQ3> stddev_aggr = iterations.reduceByKey((Function2<ValQ3, ValQ3, ValQ3>) (v1, v2) -> {
           Double fare_total_stddev = v1.getFare_stddev() + v2.getFare_stddev();
           ValQ3 v = new ValQ3(v1.getPassengers(), v1.getFare(), v1.getOccurrences(), fare_total_stddev);
          return v; });
    JavaPairRDD<KeyQ3, ValQ3> deviation = stddev_aggr.mapToPair(
              r -> {
                  Double fare_mean = r._2().getFare();
                  Integer n = r._2().getOccurrences();
                  Double fare_dev = Math.sqrt(r._2().getFare_stddev() / n);
                  Double pass_mean = r._2().getPassengers();
                  ValQ3 v = new ValQ3(pass_mean, fare_mean, n, fare_dev);
                  return new Tuple2<>(r._1(), v); });
```

SparkRDD – Query3: Grouping Statistiche




```
// RDD:=[day,List<DO_with_statistics>]
JavaPairRDD<String, Iterable<Tuple2<KeyQ3,ValQ3>>> grouped =deviation.
      groupBy((Function<Tuple2<KeyQ3,ValQ3>, String>) r -> r._1().getDay());

// RDD:=[day,List<top_5_DO_with_statistics>]
JavaPairRDD<String, List<Tuple2<Long,ValQ3>>> top_destinations = grouped.mapToPair(r ->
      new Tuple2<>(
            r._1(),
            getTopFiveDestinations(r._2())
      ))
      .sortByKey(new DateComparator());
```

SparkRDD – Query3: Risultato Finale

YYYY-MM-DD	DO1	...	avg pax DO1	...	avg fare DO1	...	stddev fare DO1	...
01/12/2021	Manhattan,Upper East Side North,Yellow Zone	...	1,36722E+16	...	1,00933E+16	...	6,36052E+15	...
02/12/2021	Manhattan,Upper East Side North,Yellow Zone	...	1,38101E+16	...	1,01709E+16	...	6,04909E+14	...
03/12/2021	Manhattan,Upper East Side North,Yellow Zone	...	1,39016E+16	...	9,90999E+15	...	6,18061E+15	...
04/12/2021	Manhattan,Upper East Side South,Yellow Zone	...	1,45702E+16	...	9,50914E+15	...	6,63641E+15	...

SparkSQL – Dataset Creation

```
public Dataset<Row> createSchemaFromRDD(SparkSession spark, JavaRDD<Row> dataset) {
    List<StructField> fields = new ArrayList<>();

    fields.add(DataTypes.createStructField("tpep_dropoff_datatime", DataTypes.TimestampType, true));
    fields.add(DataTypes.createStructField("tip_amount", DataTypes.DoubleType, true));
    fields.add(DataTypes.createStructField("tolls_amount", DataTypes.DoubleType, true));
    fields.add(DataTypes.createStructField("total_amount", DataTypes.DoubleType, true));
    fields.add(DataTypes.createStructField("payment_type", DataTypes.LongType, true));
    StructType schema = DataTypes.createStructType(fields);

    Calendar cal = Calendar.getInstance();
    cal.setTimeZone(TimeZone.getTimeZone("UTC"));
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");
    sdf.setTimeZone(TimeZone.getTimeZone("UTC"));

    JavaRDD<Row> rowRDD = dataset.map((Function<Row, Row>)
            v1 -> {
                Timestamp ts = v1.getTimestamp(1);
                cal.setTime(ts);

                Timestamp ts_zone = Timestamp.valueOf(sdf.format(cal.getTime()));
                return RowFactory.create(ts_zone, v1.getDouble(6), v1.getDouble(7), v1.getDouble(8), v1.getLong(4));
            });

    return spark.createDataFrame(rowRDD, schema);
}
```

SparkSQL – Query1

```
Dataset<Row> data = createSchemaFromRDD(spark, dataset);
data.createOrReplaceTempView("taxi_row");

Dataset<Row> values = spark.sql( "SELECT date_format(tpep_dropoff_datatime, 'y/MM') AS date," +
                                 "sum(tip_amount) AS tips, sum(tolls_amount) AS tolls, sum(total_amount) AS total," +
                                 "count(*) AS trips_number" +
                                 "FROM taxi_row WHERE payment_type = 1 GROUP BY date_format(tpep_dropoff_datatime, 'y/MM')"
                                );


values.createOrReplaceTempView("taxi_values");

results = spark.sql("SELECT date," +
                    "(tips/(total-tolls)) AS tips_percentage, trips_number FROM taxi_values ORDER BY date ASC");
```

SparkSQL – Query2: Distribuzioni viaggi

```
Dataset<Row> data = createSchemaFromRDD(spark, dataset);
data.createOrReplaceTempView("trip_infos");

// {timestamp, zone}, trips, total_trip_per_hour, zone_perc
Dataset<Row> scheduledTrips = spark.sql("SELECT timestamp, zone, trips, total_trip_hour, float(trips/total_trip_hour) as zone_perc FROM " +
                "(SELECT date_format(tpep_pickup_datatime, 'y-MM-dd HH') as timestamp, pu_location_id as zone, COUNT(*) as trips" +
                "FROM trip_infos " +
                "GROUP BY timestamp, pu_location_id)" +
                "JOIN " +
                "(SELECT date_format(tpep_pickup_datatime, 'y-MM-dd HH') as timestamp_2, COUNT(*) AS total_trip_hour" +
                " FROM trip_infos GROUP BY timestamp_2)" +
                "ON timestamp = timestamp_2 ORDER BY timestamp ASC");

scheduledTrips.createOrReplaceTempView("scheduled_trips");

// {timestamp}, list(zone_id:zone_perc)
 Dataset<Row> groupedTrips = spark.sql("SELECT timestamp, collect_list(concat_ws(':', zone, zone_perc)) as zone_percs" +
               "FROM scheduled_trips GROUP BY timestamp");
 groupedTrips.createOrReplaceTempView("grouped_trips");
```

SparkSQL – Query2: Statistiche

```
// {timestamp}, trips, avg(tip), stddev(tip)
Dataset<Row> hourly_values = spark.sql("SELECT date_format(tpep_pickup_datatime, 'y-MM-dd HH') as timestamp, COUNT(*) as trips," +
               "avg(tip) AS avg_tip, stddev_pop(tip) AS stddev_tip " +
               "FROM trip_infos " +
               "GROUP BY timestamp " +
               "ORDER BY timestamp ASC");
hourly_values.createOrReplaceTempView("hourly_values");

// {timestamp, payment_type}, occurrences
Dataset<Row> paymentOccurrences = spark.sql("SELECT date_format(tpep_pickup_datatime, 'y-MM-dd HH') AS timestamp," +
               "payment_type, COUNT(*) AS counted " +
               "FROM trip_infos GROUP BY timestamp, payment_type " +
               "ORDER BY timestamp ASC");
paymentOccurrences.createOrReplaceTempView("payment_occurrences");

// {timestamp}, most_popular_payment, payment_occurrences
Dataset<Row> mostPopularPaymentType = spark.sql("SELECT timestamp, payment_type as most_popular_payment, counted AS payment_occurrences
               "FROM  payment_occurrences table_1 WHERE counted =" +
               "(SELECT MAX(counted) FROM payment_occurrences WHERE timestamp = table_1.timestamp)
               "ORDER BY timestamp ASC");
mostPopularPaymentType.createOrReplaceTempView("most_popular_payment");
```

SparkSQL – Query2: Statistiche

```
// {timestamp}, avg_tip, stddev_tip, most_popular_payment, zone_percs
results = spark.sql(SELECT table_1.timestamp AS timestamp, avg_tip, stddev_tip, most_popular_payment, string(zone_percs) AS
                " trips_distribution, zone_percs AS percs_array FROM " +
                "(SELECT most_popular_payment.timestamp AS timestamp, avg_tip, stddev_tip, most_popular_payment FROM " +
                "hourly_values JOIN most_popular_payment ON hourly_values.timestamp = most_popular_payment.timestamp) table_1 " +
                "JOIN grouped_trips ON table_1.timestamp = grouped_trips.timestamp " +
                "ORDER BY timestamp ASC");
```

SparkSQL – Query3: Top-5

```
Dataset<Row> data = createSchemaFromRDD(spark, dataset);
data.createOrReplaceTempView("trip_infos");

Dataset<Row> values = spark.sql(" SELECT DATE(tpep_dropoff_datetime) AS day, do_location_id AS destination, AVG(passenger_count) AS
        passenger_avg, AVG(fare_amount) as fare_avg, STDDEV_POP(fare_amount) AS fare_stddev, COUNT(*) AS dest_for_day
        FROM trip_infos GROUP BY day, destination ORDER BY day, dest_for_day DESC");
values.createOrReplaceTempView("values");

Dataset<Row> top5_per_day = spark.sql(" SELECT day, destination, dest_for_day, passenger_avg, fare_avg, fare_stddev
        FROM (SELECT *, ROW_NUMBER() OVER (PARTITION BY day ORDER BY dest_for_day DESC) AS top FROM values) WHERE top <= 5");
top5_per_day.createOrReplaceTempView("top5_per_day");
```

SparkSQL – Query3: Output

```
Dataset<Row> merged_days = spark.sql(" SELECT day, COLLECT_LIST(destination) AS dest, COLLECT_LIST(passenger_avg) AS pass,
        COLLECT_LIST(fare_avg) AS fare, COLLECT_LIST(fare_stddev) AS stddev
        FROM top5_per_day GROUP BY day");
        merged_days.createOrReplaceTempView("merged_days");

spark.udf().register("setZones", (UDF1<String, String>) id -> Zone.zoneMap.get(Integer.parseInt(id)), DataTypes.StringType);
results = spark.sql(" SELECT day, setZones(CAST(ELEMENT_AT(dest, 1) AS String)) AS D01, setZones(CAST(ELEMENT_AT(dest, 2) AS String)) AS
        D02,setZones(CAST(ELEMENT_AT(dest, 3) AS String)) AS D03, setZones(CAST(ELEMENT_AT(dest, 4) AS String)) AS D04,
        setZones(CAST(ELEMENT_AT(dest, 5) AS String)) AS D05, ELEMENT_AT(pass, 1) AS avg_pax_D01, ELEMENT_AT(pass, 2) AS avg_pax_D02,
        ELEMENT_AT(pass, 3) AS avg_pax_D03, ELEMENT_AT(pass, 4) AS avg_pax_D04, ELEMENT_AT(pass, 5) AS avg_pax_D05, ELEMENT_AT(fare, 1) AS
        avg_fare_D01, ELEMENT_AT(fare, 2) AS avg_fare_D02, ELEMENT_AT(fare, 3) AS avg_fare_D03, ELEMENT_AT(fare, 4) AS avg_fare_D04,
        ELEMENT_AT(fare, 5) AS avg_fare_D05, ELEMENT_AT(stddev, 1) AS avg_stddev_D01, ELEMENT_AT(stddev, 2) AS avg_stddev_D02,
        ELEMENT_AT(stddev, 3) AS avg_stddev_D03, ELEMENT_AT(stddev, 4) AS avg_stddev_D04, ELEMENT_AT(stddev, 5) AS avg_stddev_D05
        FROM merged_days");
```

Analisi performance

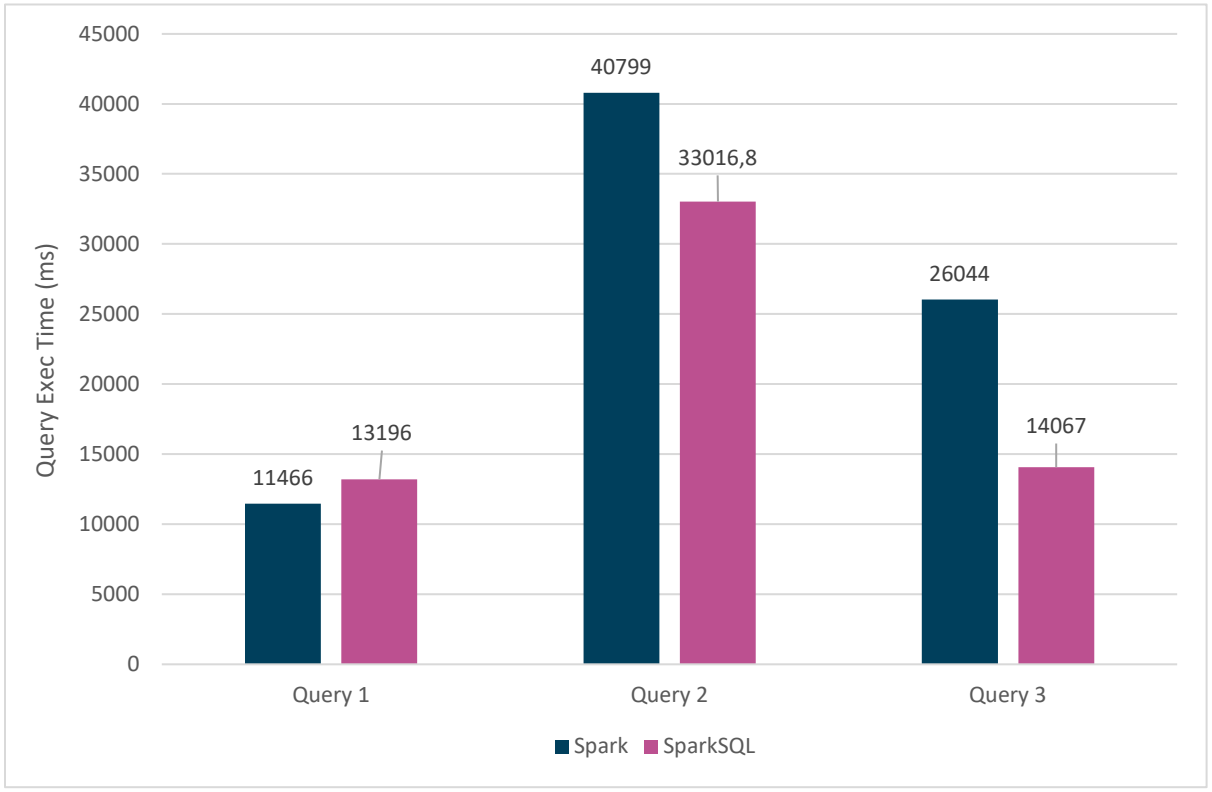
- Per l’analisi delle prestazioni sono state eseguite diverse ripetizioni delle Query
- Si valuta la media dei tempi di risposta di ogni ripetizione
- Si carica il dataset ogni volta che si esegue una query
- Specifiche macchina locale:

```
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).
```

Analisi performance

- Nel grafico vediamo la valutazione delle query effettuate tramite RDD API rispetto a quelle eseguite dal modulo SparkSQL.

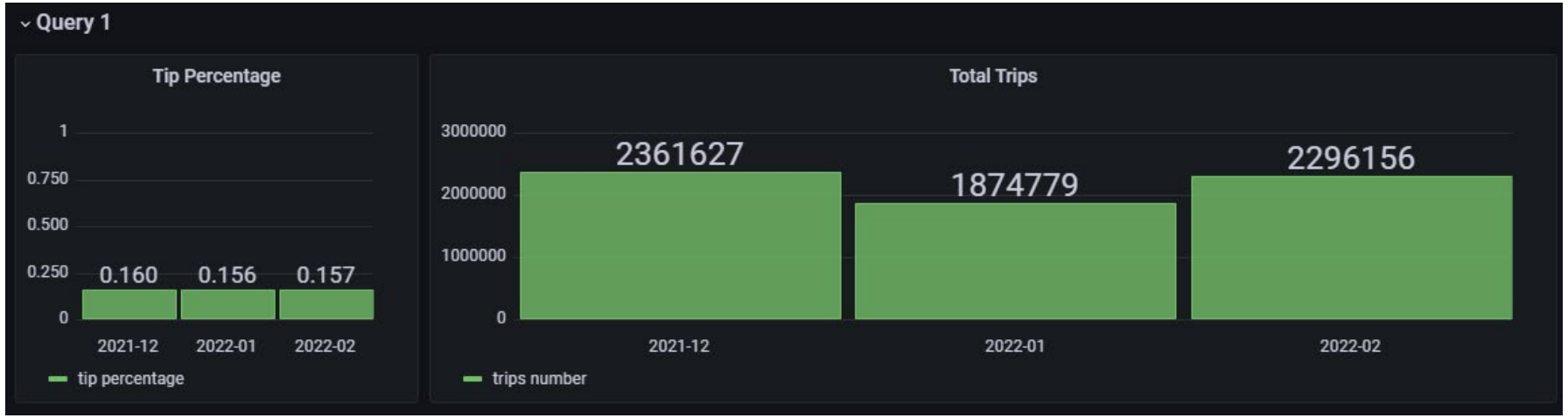
Q1	Q2	Q3	Q1SQL	Q2SQL	Q3SQL
11753	40248	25982	13612	33334	15308
11176	40915	26303	12299	33750	14053
11369	40195	26615	13157	32279	13524
11412	41048	25321	13470	34323	14088
11620	41589	25999	13442	31398	13362



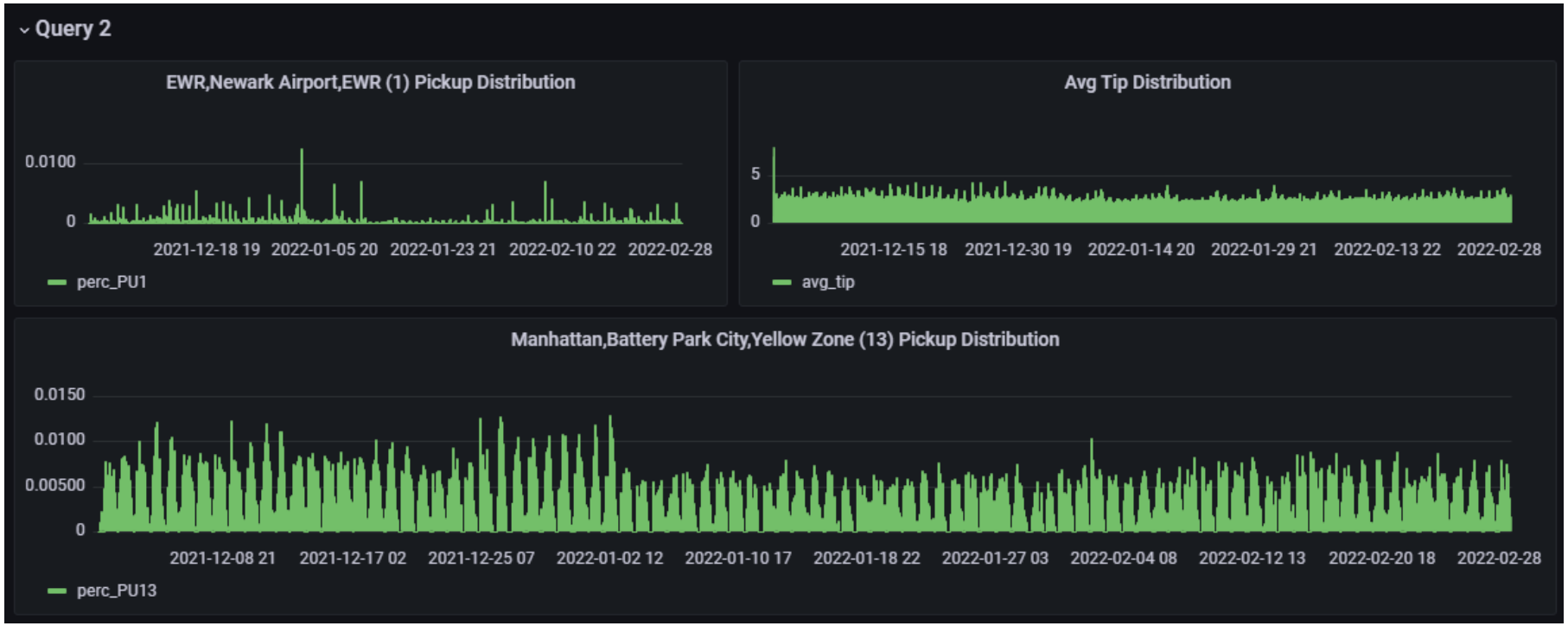
Grafana Dashboard

- La dashboard è stata creata andando ad effettuare query sui file csv, risultati della computazione di Spark.
- Tramite un immagine docker di Grafana, viene offerta una visualizzazione dei dati tramite una semplice Dashboard.
- Link Dashboard : http://localhost:3001/d/QVfEthCnz/sabd-1?orgId=1

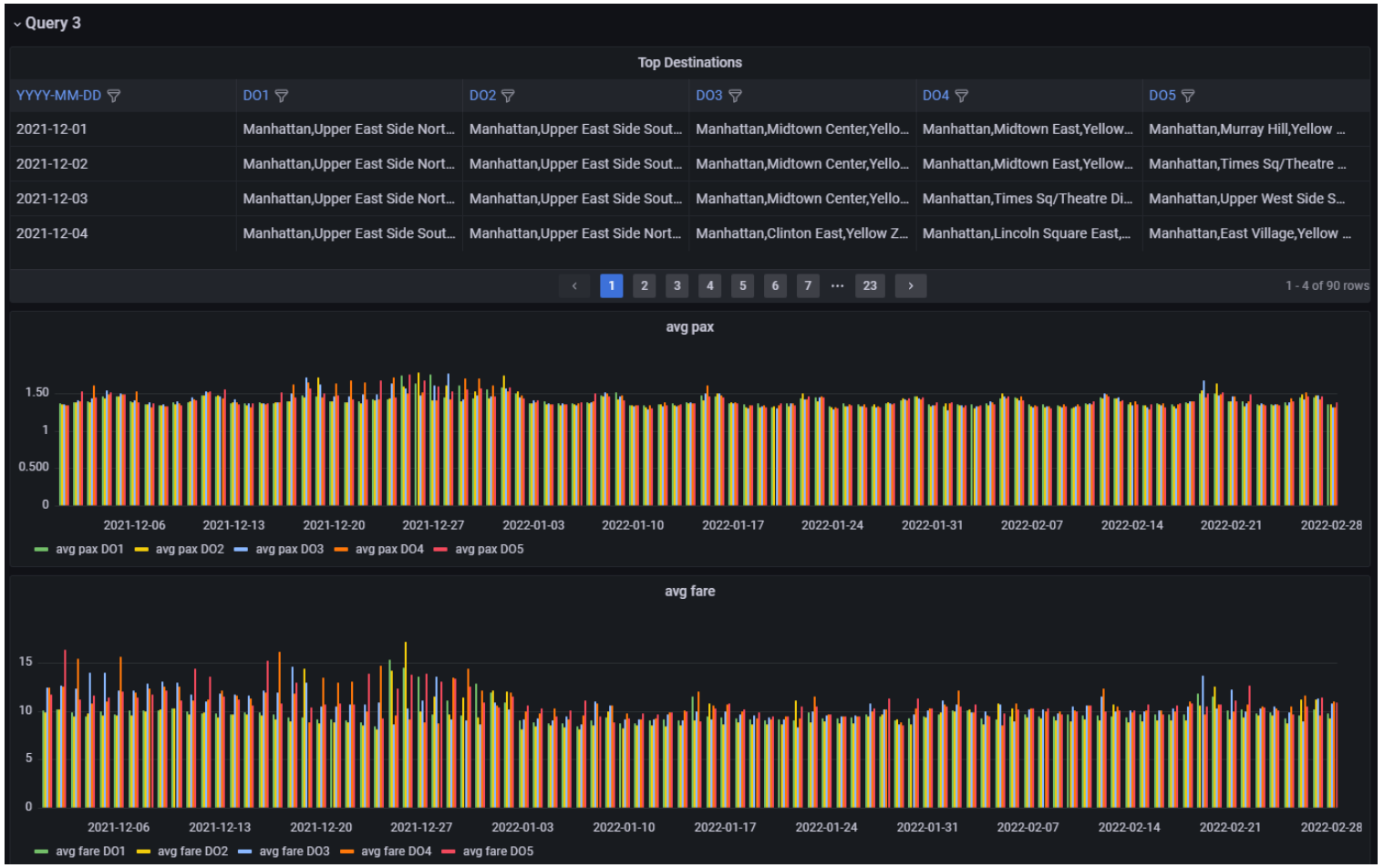
Grafana Dashboard – Query 1



Grafana Dashboard – Query 2



Grafana Dashboard – Query 3



Conclusioni

- Punti di forza:
  - Ragionevole velocità di esecuzione
  - Pre-processamento tramite framework esterno
- Limitazioni:
  - Supporto limitato di Grafana verso HDFS e MongoDB
  - Scalabilità limitata a causa delle ridotte risorse computazionali della singola macchina host

Grazie per l’attenzione

DANILO DELL’ORCO

JACOPO FABI

MICHELE SALVATORI