High-performance computing for big data in companies

Number of cores and executors used:

- Executors: 20
- Cores: 4

Startup

To start working with our dataset we first have to import the SQLContext for spark and use it to read, extract and save the table in a data frame.

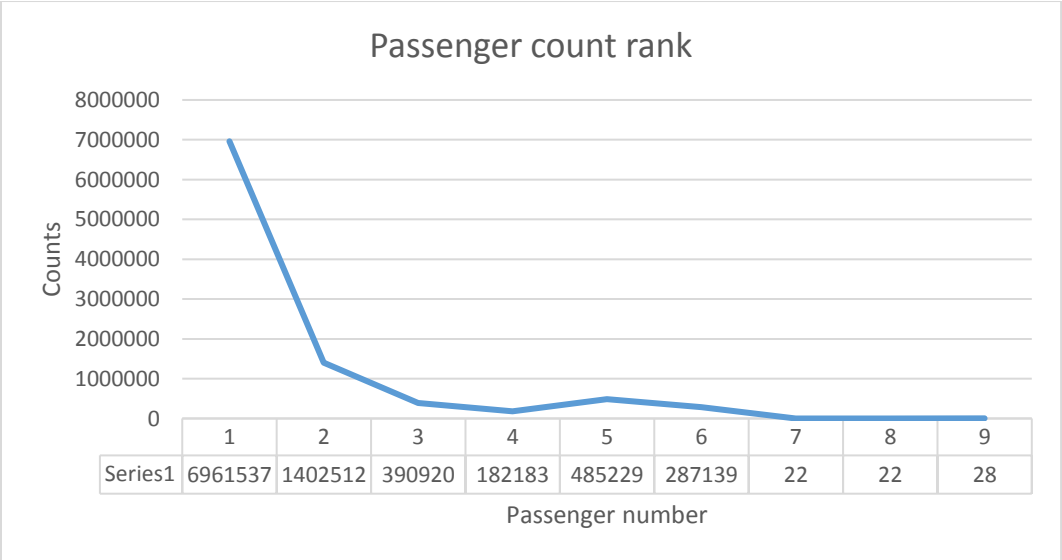
```
> import org.apache.spark.sql.SQLContext
> val df = sqlContext.read.format("com.databricks.spark.csv").option("header", "true").option("inferSchema", "true").load("hdfs:///YellowCab/2017/yellow_tripdata_2017-01.csv")
```

We then use the data frame to generate a temporary table in order to extract values from the table using the sql function from the SQLContext.

```
> df.registerTempTable("temp")
```

An ordered list showing on top positions the most common number of passengers for a trip and a value that represents the number of times that trip has been done.

```
> val extract = sqlContext.sql("SELECT Passenger_count FROM temp WHERE Passenger_count IS NOT NULL AND Passenger_count!=0")
> extract.map(x => x(0).toString).map(count => (count, 1L)).reduceByKey(_+_).sortBy(counts => (counts._1)).take(9).map(h => h._1 + ", " + h._2).foreach(println)
```



Francesca Sallicati y Borja Ruiz Amantegui

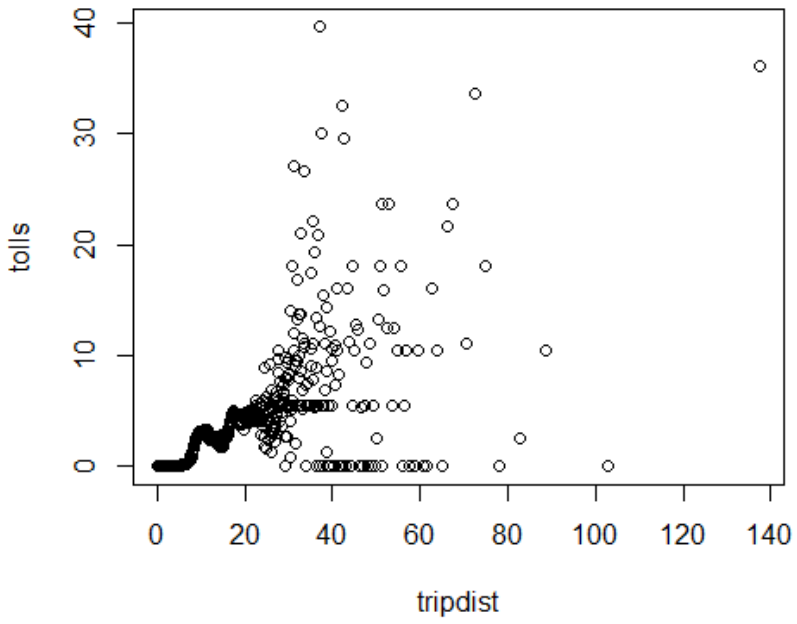
An Sql order to extract a set trip_distance, the number of times it has been realized and the sum of all tolls paid for that set distance. We map to obtain the mean amount paid for each trip_distance.

```
> val extract = sqlContext.sql("SELECT trip_distance, count('X'), sum(tolls_amount) FROM temp
  WHERE trip_distance!=0 GROUP BY (trip_distance) ORDER BY (trip_distance)")
> extract.collect().map(x => (x(0), x(2).toString.toDouble /
  x(1).toString.toDouble)).take(4420).map(h => h._1 + ", " + h._2).foreach(println)
```

Since the precision of the trip_distance is pretty high and the values for tolls disperse, we extracted a 10^{th} portion of the elements to obtain a more representative graph.

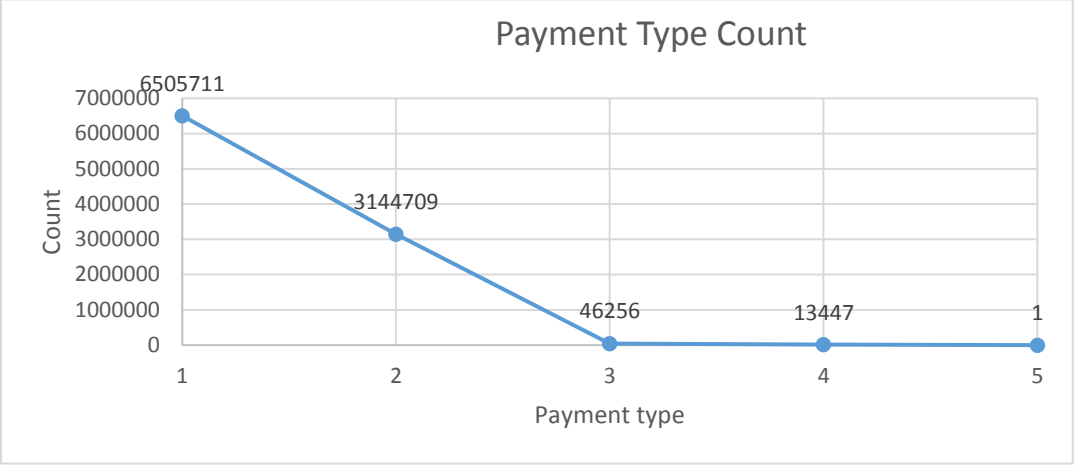
R code:

```
x<-problem2$Trip.distance
y<-problem2$Mean.tolls.paid.per.trip

tripdist<-NULL
tolls<-NULL

x1<-seq(1,4420,by=10)
z = 0;
for (i in x1){
    tolls[z] = y[i];
    tripdist[z]=x[i] z = z + 1;
}
plot(tripdist,tolls)
```



Payment type sorted by the most common.

```
> val extract = sqlContext.sql("SELECT Payment_type FROM temp")
> extract.map(x => (x.toString, 1L)).reduceByKey(_+_).sortBy(x => x._1)
```



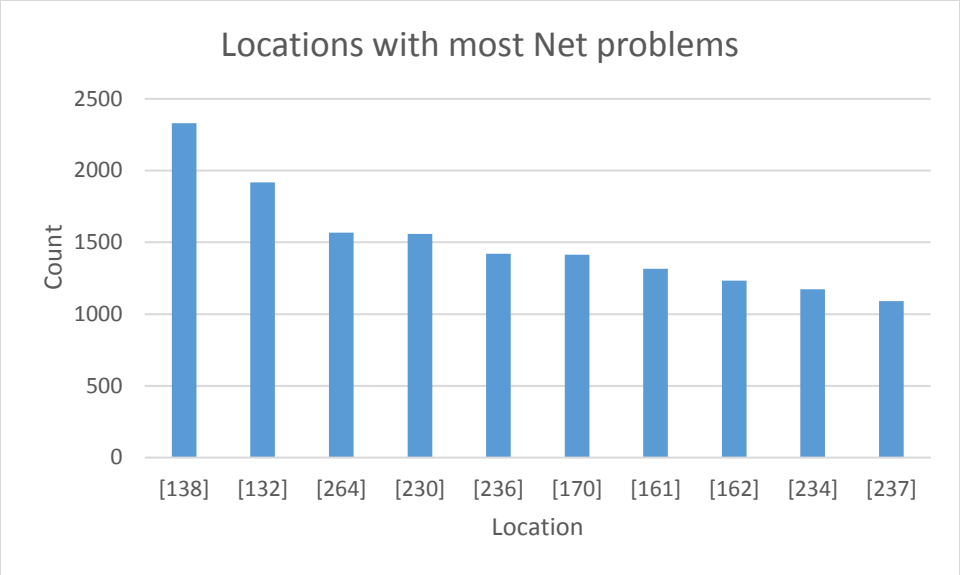
Extracted amount paid for the trip with the tip paid. Mapped to obtain the tips paid in relation to the total amount paid and obtained the mean from the whole dataset.

```
> val extract = sqlContext.sql("SELECT Total_amount, Tip_amount FROM temp WHERE
  Total_amount != 0 AND Tip_amount != 0")
> import org.apache.spark.AccumulatorParam
> val accum = sc.accumulator(0.0)
> val array = extract.map(x => x(1).toString.toDouble / x(0).toString.toDouble)
> array.foreach(accum += _)
> accum.value/array.count
```

accum = 0.16174787149595726

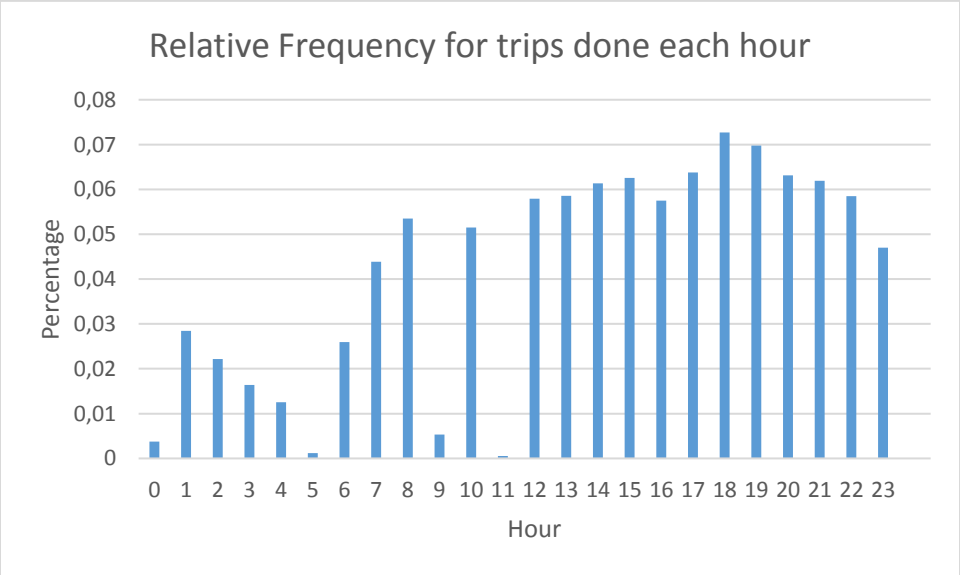
Extracted location where the payment wasn't done immediately after the drop off and reduced counting the appearances in order to obtain the locations with worst net connection.

```
> val extract = sqlContext.sql("SELECT DOLocationID FROM temp WHERE Store_and_fwd_flag LIKE
  'Y'")
> extract.map(x => x.toString).map(count => (count, 1L)).reduceByKey(_+_).sortBy(x => -
  x._2).take(10).map(h => h._1 + ", " + h._2).foreach(println)
```



We group by hours in order to obtain the total amount of trips. We then sort by hour and obtain the relative frequencies.

```
> val extract =sqlContext.sql("SELECT hour(tpep_pickup_datetime), count('X') from temp group by hour(tpep_pickup_datetime)")
> extract.map(x => (x(0).toString, x(1).toString)).sortBy(x => x._1).take(24).foreach(println)
```



We obtained the total minutes of trips per day by substracting drop off and pick up time.

```
> val extract = sqlContext.sql("SELECT dayofyear(tpep_pickup_datetime),
  hour(tpep_pickup_datetime), minute(tpep_pickup_datetime), hour(tpep_dropoff_datetime),
  minute(tpep_dropoff_datetime) FROM temp WHERE dayofyear(tpep_pickup_datetime) ==
  dayofyear(tpep_dropoff_datetime)")
> extract.map(x => (x(0).toString.toInt,  x(3).toString.toDouble*60.toString.toDouble +
  x(4).toString.toDouble - x(1).toString.toDouble*60 + x(2).toString.toDouble
  )).reduceByKey(_+_).sortBy(x => x._1).collect().map(h => h._1 + ", " + h._2).foreach(println)
```