Big Data in NYC for Taxi rides management

with Apache Hadoop

Gabriele Favia matr. 579166

November 2019

Contents

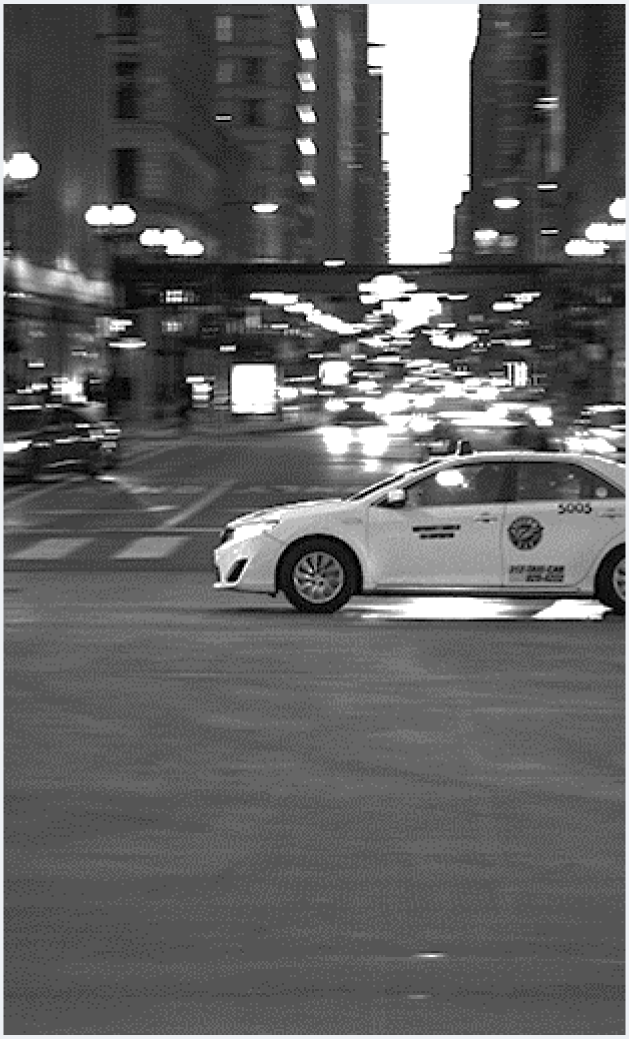
The NYC Taxi and Limousine Commission

The New York City Taxi and Limousine Commission (NYC TLC) is an agency of the New York City government that licenses and regulates the medallion taxis and for-hire vehicle industries, including app-based companies. The TLC's regulatory landscape includes medallion (yellow) taxicabs, green or Boro taxicabs, black cars (including both traditional and app-based services), community-based livery cars, commuter vans, paratransit vehicles (ambulettes), and some luxury limousines [1].

But what is the difference between yellow and green taxi (cab in American English)?

The famous NYC yellow taxis provide transportation exclusively through street-hails.
The number of taxicabs is limited by a finite number of medallions issued by the TLC.
It's possible to access this mode of transportation by standing in the street and hailing an available taxi with your hand. The pickups are not pre-arranged.

The green taxis know as well as "Boro" cab or Street Hail Livery (SHL) are permitted to accept street-hails above 110th Street in Manhattan and in the outer-boroughs of New York City. The SHL program allows livery vehicle owners to license and outfit their vehicles with green borough taxi branding, meters, credit card machines, and ultimately the right to accept street hails in addition to pre-arranged rides [2].



Requests

Designing and implement in Hadoop MapReduce:

1. A job returning the total number of passengers in yellow and green taxi in 2018.

2. A job returning, for each rate code, the total amount charged to passengers of yellow and green taxi in 2018.

3. A job returning, for each PULocationID, the list of related DOLocationID for yellow taxi in 2018, ordered by increasing average trip distance.

From now on, the requests will be redefined as “Point 1”, “Point 2” and “Point 3”, respectively.

Data dictionary

The two kind of file of interest, yellow and green have a different header, which means that the software has the need to distinguish and operate the two cases differently.

Moreover, the green and yellow taxis dataset produced from January 2016 to June 2016, follow a different data dictionary that requires some variation to the parsing procedure for the points 1 and 2, while for the point 3, the dataset cannot be use because of the absence of these fields.

The fields of interest to be extracted from both the yellow and green cabs files are:

Field	Description
lpep_pickup_datetime or rpep_pickup_datetime	The pickup time, useful to get the year.
Passenger_count	The number of passengers in the vehicle.
RatecodeID	The final rate code in effect at the end of the trip. 1 = Standard rate 2 = JFK 3 = Newark 4 = Nassau or Westchester 5 = Negotiated fare 6 = Group ride
total_amount	The total amount charged to passengers. Does not include cash tips.

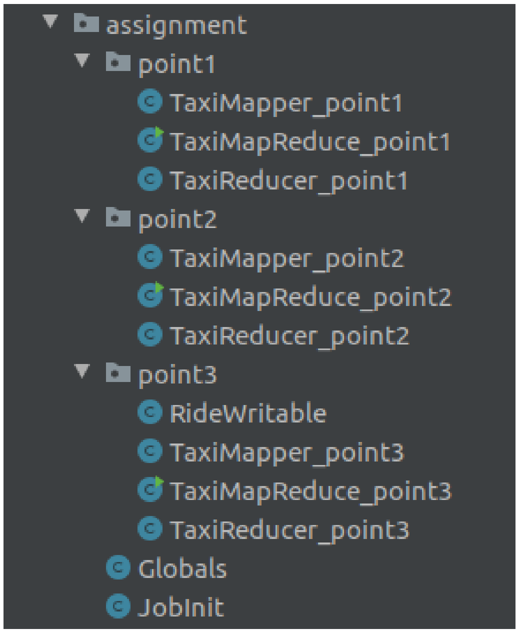
While the fields of interest exclusively extracted from yellow files are:

Field	Description
PULocationID	TLC Taxi Zone in which the taximeter was engaged.
DOLocationID	TLC Taxi Zone in which the taximeter was disengaged.
trip_distance	The elapsed trip distance in miles reported by the taximeter.

A comprehensive description of all the fields is available at nyc.gov website both for yellow [3] and the green [4] cabs.

Structure of the program

The software has the principal package named “assignment”, as depicted in the *Figure 1,* with sub-packages containing the main, map and reduce classes aimed to resolve the points of the requests.



```
▼ assignment
  ▼ point1
      TaxiMapper_point1
      TaxiMapReduce_point1
      TaxiReducer_point1
  ▼ point2
      TaxiMapper_point2
      TaxiMapReduce_point2
      TaxiReducer_point2
  ▼ point3
      RideWritable
      TaxiMapper_point3
      TaxiMapReduce_point3
      TaxiReducer_point3
    Globals
    JobInit
```

Figure 1 - Hierarchy of the software packages

“RideWritable” extends the “WritableComparable” of Hadoop and acts as a container for the three parameters of interest of the 3rd point: *PULocationID, DOLocationID* and trip*_distance*.
The creation of *RideWriteble* has been considered unavoidable because the data exchange between Map and Reduce tasks takes place only through key-value pair.

“Globals” is a class conceived to be the container of the “constants” of the program as described below:

Constant name	Description
DELIMITER	The delimiter for the csv files.
GREEN_TAXI	Mnemonic constant to distinct the type of taxi based on the starting 5 letters of the file name.
YELLOW_TAXI	
REGEX_OLD	Regex to identify yellow and green cabs files produced before July 2016.

“JobInit” is a class which provides the static function runJob() used as an interface to initialize and run the job from the various main by passing only different parameters.

Explaining the algorithm

The three objectives will be achieved using Apache Hadoop using a java "main" for each of them, using the paradigm map-reduce.

- For the *map* algorithm, the technique to skip the header row is checking if the row starts with the word "Vendor" which is common in every header of the files taken into account.

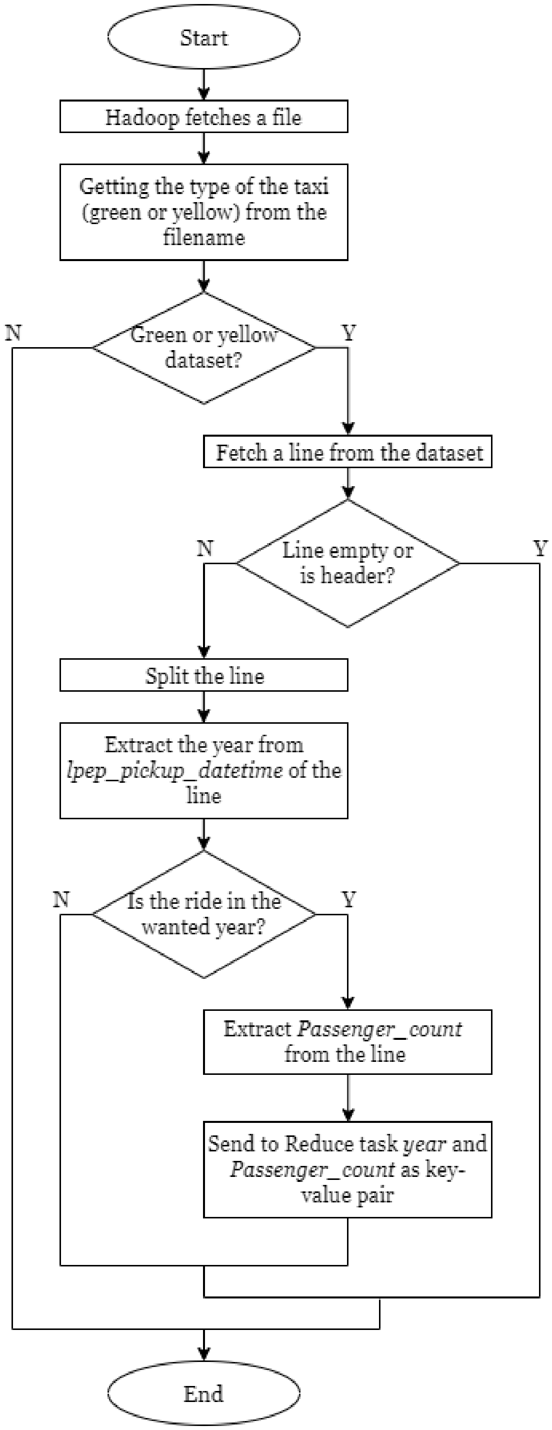
- As said in the *Data dictionary* section, the headers of the files of yellow and green cabs antecedent to July 2016 are structured in a different way (some diverse field and disposition), so the regex "`.*_tripdata_2016-0[1-6].csv`" is used in the Point1 and Point2 tasks to switch to the right field inside the split string of the fetched row, while in the Point3 is used to avoid these files because of the lack of the required fields (*PULocationID*, *DOLocationID*).

- In every *map* task, there is always a check about the year since the user can specify in the command line which years should the algorithms work with (2018 by default), moreover in some files which are supposed to refer to a specific year (having the filename as reference), are "contaminated" by rows related to different a year, so the verification is mandatory.

Main name: *TaxiMapReduce_point1*

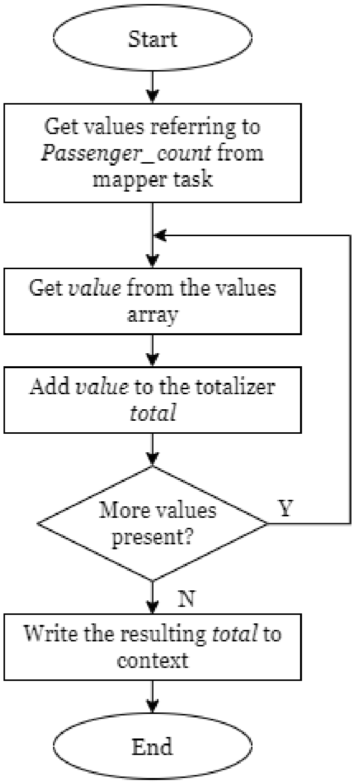
Package: *assignment.point1*

TaxiMapper_point1: ***map flow chart:***



Start
Hadoop fetches a file
Getting the type of the taxi (green or yellow) from the filename
Green or yellow dataset?
N
Y
Fetch a line from the dataset
Line empty or is header?
N
Y
Split the line
Extract the year from *lpep_pickup_datetime* of the line
Is the ride in the wanted year?
N
Y
Extract *Passenger_count* from the line
Send to Reduce task *year* and *Passenger_count* as key-value pair
End

TaxiReducer_point1: ***reduce flow chart:***

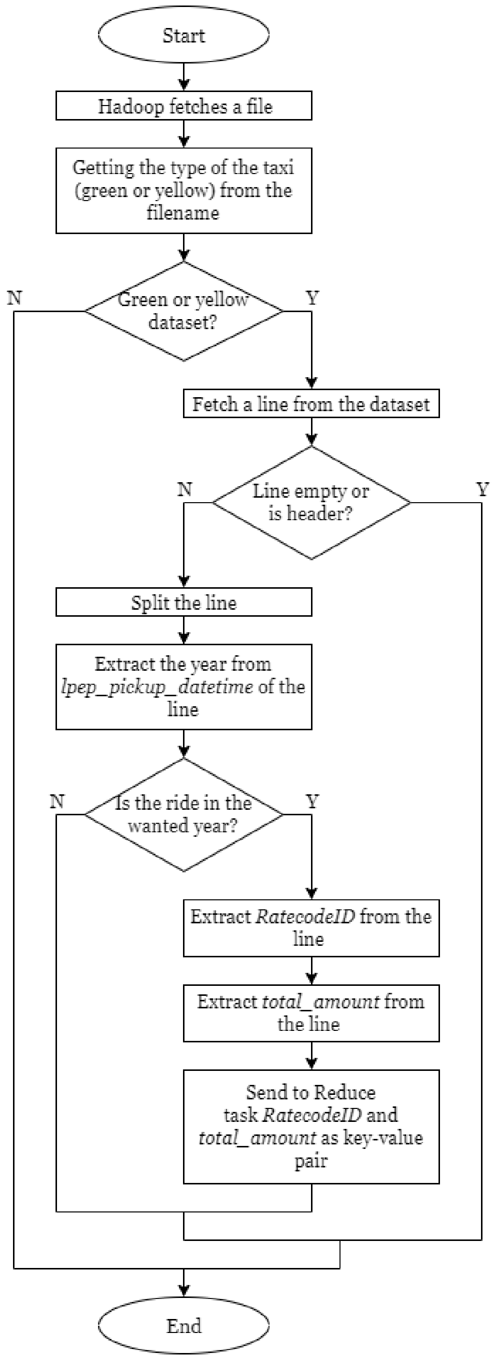


Start
Get values referring to *Passenger_count* from mapper task
Get *value* from the values array
Add *value* to the totalizer *total*
More values present?
Y
N
Write the resulting *total* to context
End

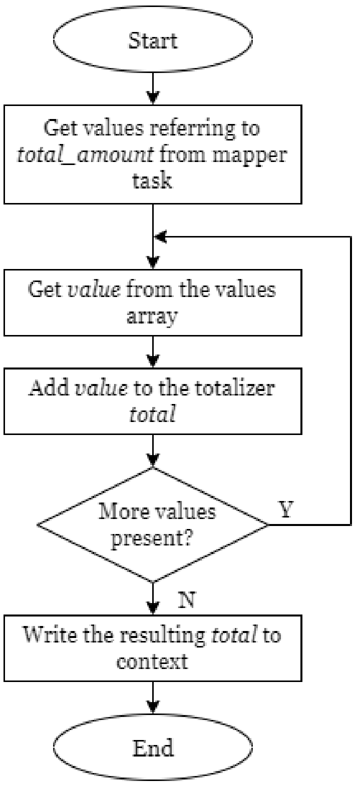
Main name: *TaxiMapReduce_point2*

Package: *assignment.point2*

TaxiMapper_point2: ***map flow chart:***


Start
Hadoop fetches a file
Getting the type of the taxi (green or yellow) from the filename
Green or yellow dataset?
N
Y
Fetch a line from the dataset
Line empty or is header?
N
Y
Split the line
Extract the year from lpep_pickup_datetime of the line
Is the ride in the wanted year?
N
Y
Extract RatecodeID from the line
Extract total_amount from the line
Send to Reduce task RatecodeID and total_amount as key-value pair
End

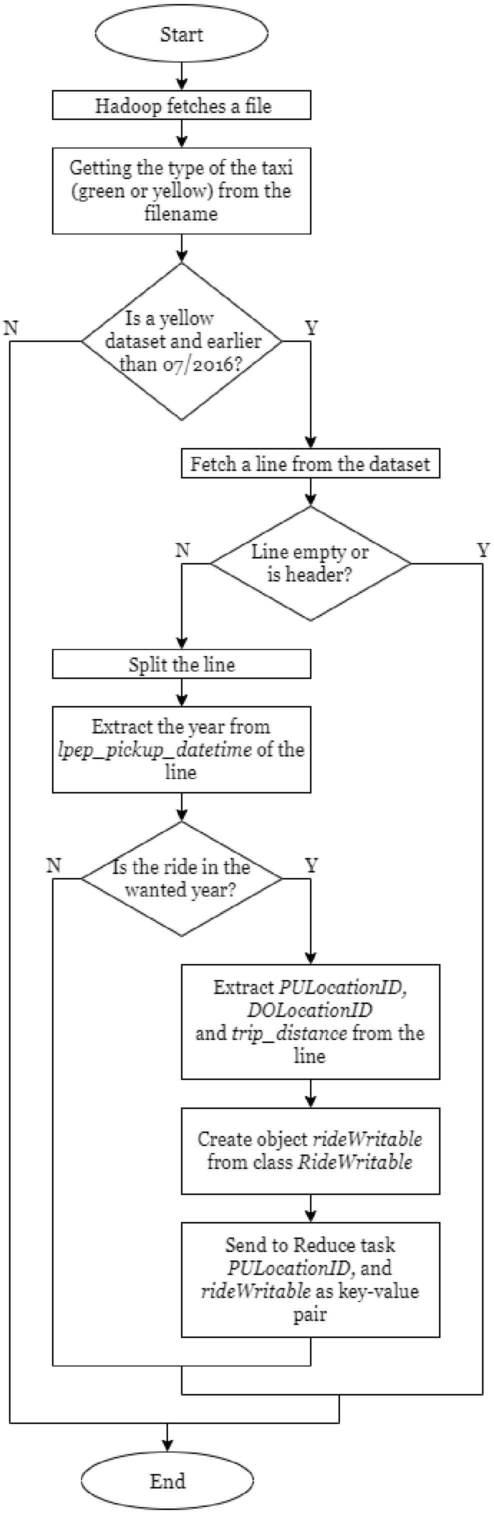
TaxiReducer_point2: ***reduce flow chart:***


Start
Get values referring to total_amount from mapper task
Get value from the values array
Add value to the totalizer total
More values present?
Y
N
Write the resulting total to context
End

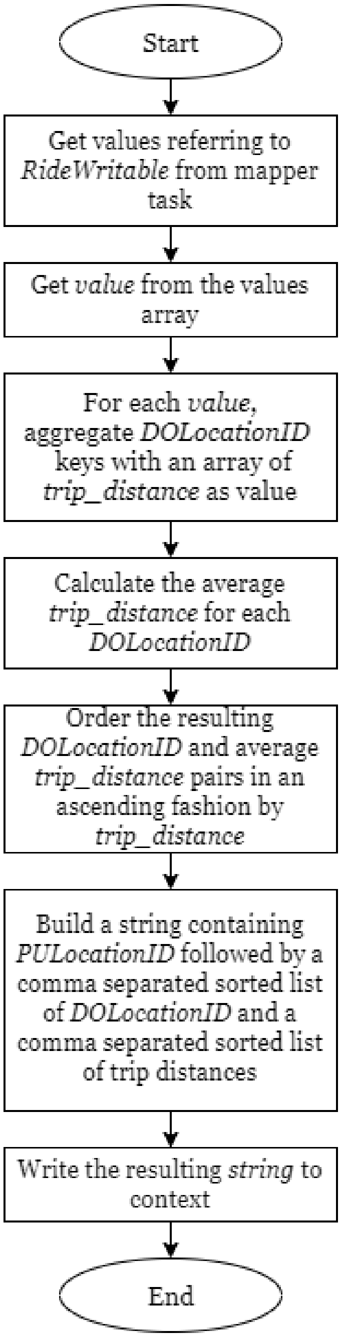
Main name: *TaxiMapReduce_point3*

Package: *assignment.point3*

TaxiMapper_point3: ***map flow chart:***


Start
Hadoop fetches a file
Getting the type of the taxi (green or yellow) from the filename
Is a yellow dataset and earlier than 07/2016?
N
Y
Fetch a line from the dataset
Line empty or is header?
N
Y
Split the line
Extract the year from lpep_pickup_datetime of the line
Is the ride in the wanted year?
N
Y
Extract PULocationID, DOLocationID and trip_distance from the line
Create object rideWritable from class RideWritable
Send to Reduce task PULocationID, and rideWritable as key-value pair
End

TaxiReducer_point3: ***reduce flow chart:***


Start
Get values referring to RideWritable from mapper task
Get value from the values array
For each value, aggregate DOLocationID keys with an array of trip_distance as value
Calculate the average trip_distance for each DOLocationID
Order the resulting DOLocationID and average trip_distance pairs in an ascending fashion by trip_distance
Build a string containing PULocationID followed by a comma separated sorted list of DOLocationID and a comma separated sorted list of trip distances
Write the resulting string to context
End

Code and usage

Command for the Point 1

```
$HADOOP_HOME/bin/hadoop jar <path to mapreduce_assignment.jar>
assignment.point1.TaxiMapReduce_point1 <input folder> <output folder>
<years>
```

Command for the Point 2

```
$HADOOP_HOME/bin/hadoop jar <path to mapreduce_assignment.jar>
assignment.point2.TaxiMapReduce_point2 <input folder> <output folder>
<years>
```

Command for the Point 3

```
$HADOOP_HOME/bin/hadoop jar <path to mapreduce_assignment.jar>
assignment.point3.TaxiMapReduce_point3 <input folder> <output folder>
<years>
```

Where:

`<input folder>` is the folder in which the dataset files are located.

`<output folder>` is the folder in which the result of the processing is saved.

`<years>` is an optional parameter: by avoiding this, the software will process the rows related to the year 2018 only; otherwise the user can specify the years affected by typing them one by one divided by a comma.

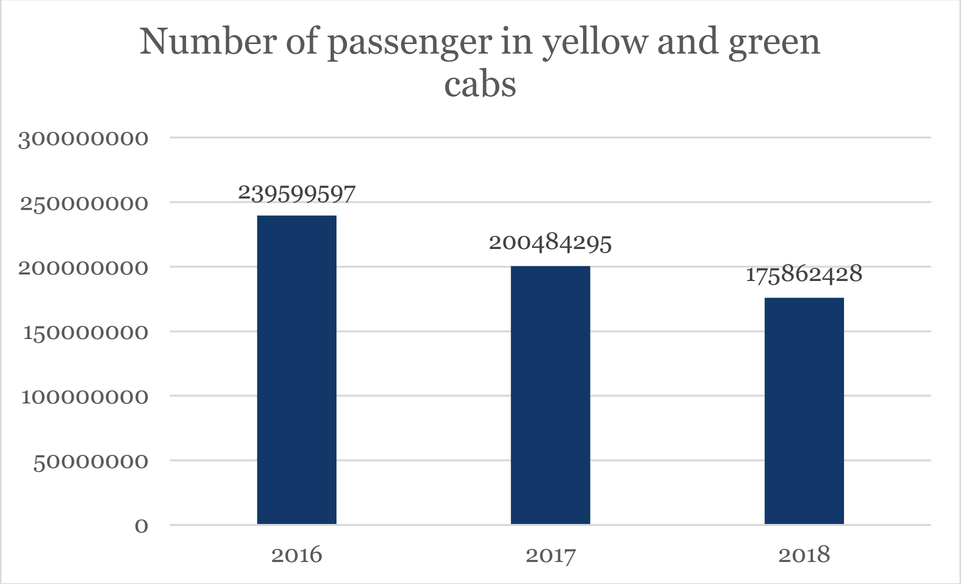
> es. `input output 2017,2018`

Notes

- Although still used very often in the Hadoop java projects, `StringTokenizer` results to be deprecate and it's been substituted with the function `split` of the `String` java class [5].

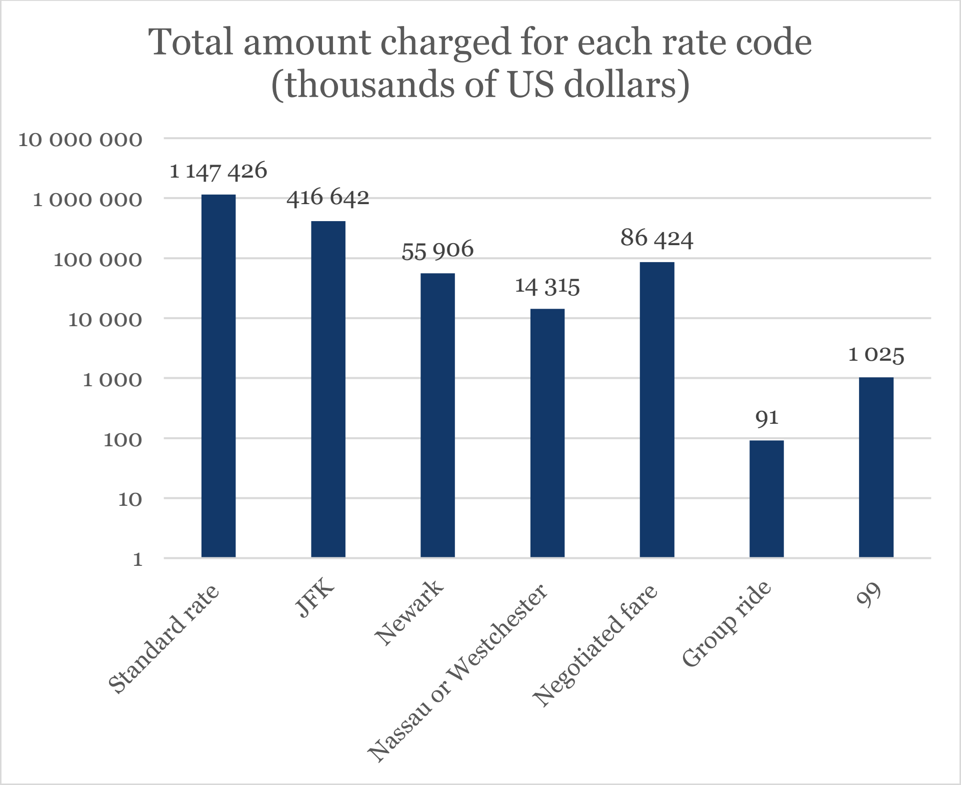
Results

Point 1



The number of passengers has followed a descending trend over the three years of analysis, with a major drop from 2016 to 2017 (-16.32%), followed by an ulterior loss of (-12.28%) from 2017 to 2018.

Point 2



The rate code for which the driver's gross margin results higher is the *1*, corresponding to *Standard rate* with 1.14 billion US dollars, followed by *JFK* with 416 million US dollars. The value *99* is not documented in the Data dictionary but it's reasonable to think of it as the "all other cases" option.

Point 3

Here is displayed the tabular version of just one line of the output file as an example.
It's been evaluated to truncate the distance to the second decimal digit, since it could make the reading and understanding less heavy for the reader.

PULocationID: 1	
DOLocation	trip_distance (miles)
264	0.35
1	1.53
243	6.19
42	7.79
4	9.12
265	9.46
229	10.44
23	11.11
118	11.27
43	11.42
233	11.72
156	12.7
114	12.96
125	13.11
144	13.37
13	13.84
68	13.86
231	14.31
79	14.32
107	14.44
251	14.59
12	14.6
148	14.62
163	14.76
158	14.8
232	14.85
45	14.85
187	14.91
164	15.06
87	15.09
249	15.15
113	15.25
211	15.36
244	15.36

48	15.43
230	15.6
206	15.75
141	15.76
66	15.78
90	15.8
145	15.8
65	15.96
234	16.07
261	16.15
88	16.23
214	16.3
256	16.52
209	16.61
100	16.7
246	16.84
161	16.89
166	17.21
40	17.4
238	17.47
170	17.67
186	17.84
25	17.92
239	17.96
137	18.1
237	18.11
172	18.13
143	18.28
142	18.29
236	18.34
49	18.5
225	18.53
37	18.6
162	18.79
50	18.98
112	19.54
151	19.57
14	19.86
84	20.04
24	20.15
33	20.39
7	21.3
82	21.6
41	21.71
181	21.73
228	22.12
255	22.2

83	22.52
169	22.81
119	23.1
123	23.42
26	23.6
80	23.62
257	23.67
154	23.8
263	24.13
56	24.3
116	24.63
75	24.81
74	24.9
89	24.99
197	25.2
91	25.29
140	25.68
61	25.73
152	25.79
138	26.18
178	26.67
55	26.69
129	26.96
93	27.1
57	27.88
132	28.28
21	28.3
262	28.33
223	29.91
76	30.16
92	30.38
121	31.27
67	39.27
117	55
81	56.49

Performance comparison

The program instances are run on the following virtual machine:

The operative system is Ubuntu 18.4.4 hosted by VirtualBox vers. 6.0.14 which are given:

2 cores;
4096 MB RAM;
50GB disk space;

Equipped with Hadoop vers. 3.2.1, Java Open JDK vers. 1.8.0

The host machine with:

Intel© Core™ i7 6500U and SSD.

Time for completion

years: 2018

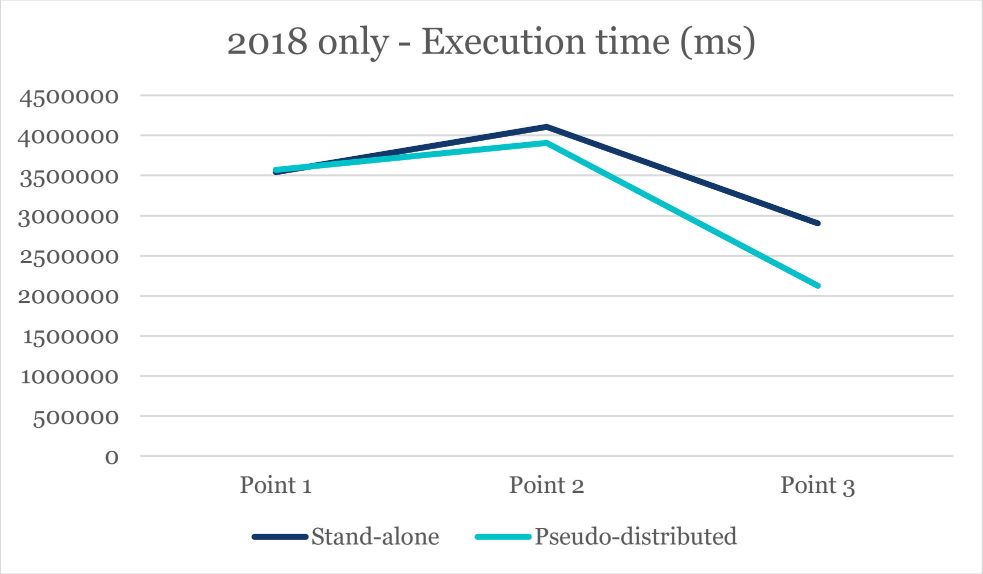
Task	Execution time in stand-alone mode (ms)	Execution time in psuedo-distributed (ms)
Point 1	3540740	3569832
Point 2	4106613	3906613
Point 3	2902725	2123984

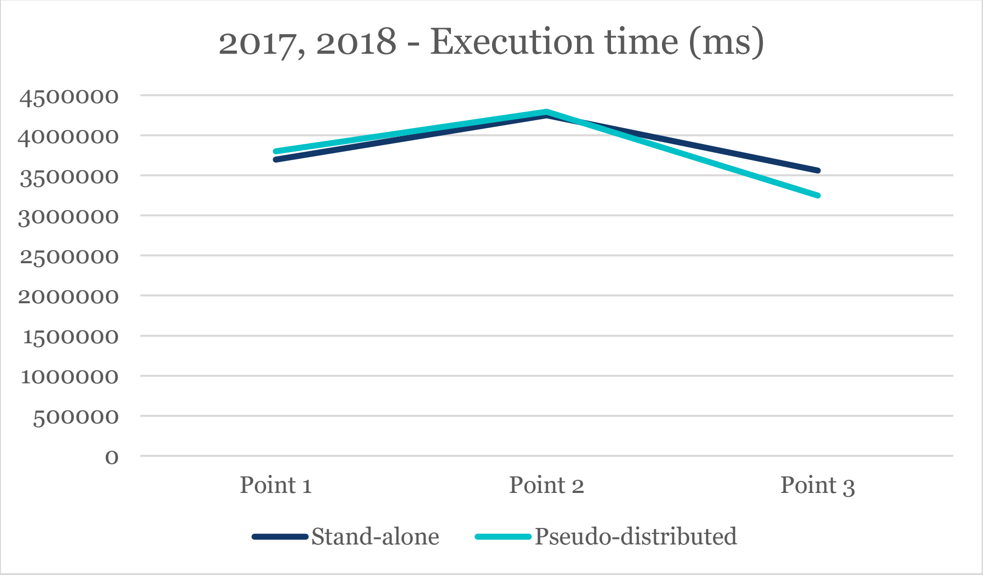
years: 2017, 2018

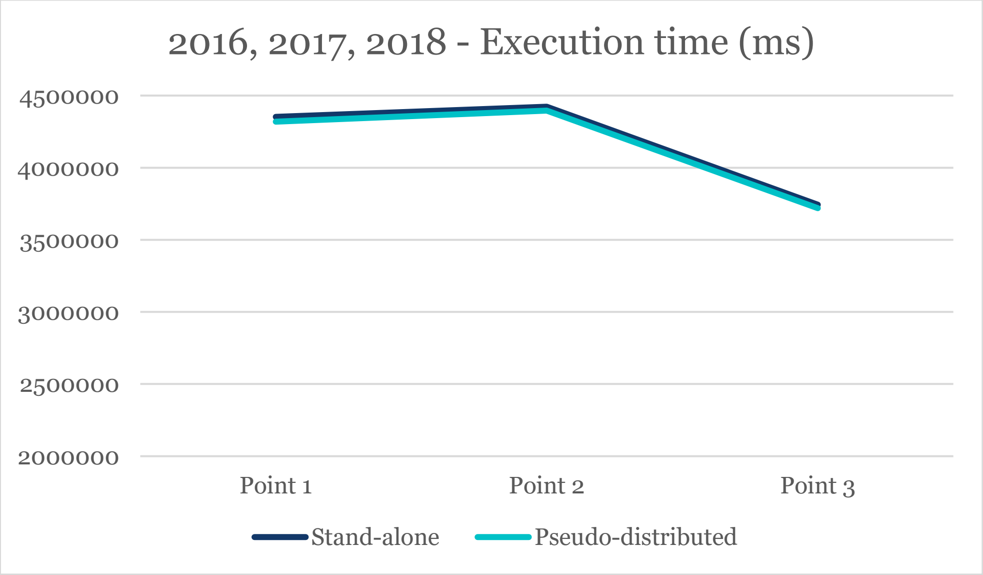
Task	Execution time in stand-alone mode (ms)	Execution time in psuedo-distributed (ms)
Point 1	3697024	3799800
Point 2	4248917	4293623
Point 3	3559784	3247968

years: 2016, 2017, 2018

Task	Execution time in stand-alone mode (ms)	Execution time in psuedo-distributed (ms)
Point 1	4352669	4319837
Point 2	4422663	4395583
Point 3	3743565	3719525


2018 only - Execution time (ms)
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
Point 1
Point 2
Point 3
Stand-alone
Pseudo-distributed


2017, 2018 - Execution time (ms)
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
Point 1
Point 2
Point 3
Stand-alone
Pseudo-distributed


2016, 2017, 2018 - Execution time (ms)
4500000
4000000
3500000
3000000
2500000
2000000
Point 1
Point 2
Point 3
Stand-alone
Pseudo-distributed

The pseudo-distributed mode seems to be faster on high intensive tasks as the Point 3 (up to 26.8% circa), where an extended class is passed as a value and the reduce part is made up of more loops and arrays variables.

Limitations

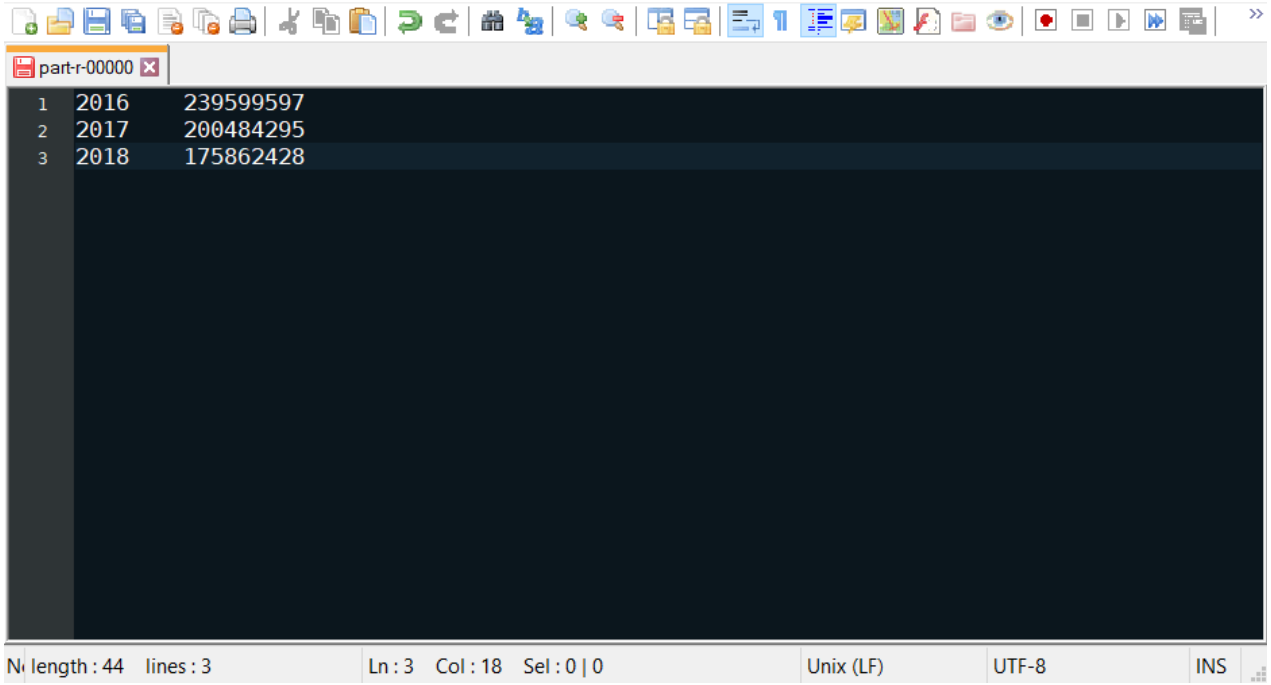
The results concerning the execution times are biased by the state of the host machine which was loaded with different tasks in background, causing a not constant access time and bandwidth availability on the SSD on the host operating system, so these must be taken as an overall idea of the average execution time for the machine used.

References

[1] https://en.wikipedia.org/wiki/New_York_City_Taxi_and_Limousine_Commission
[2] https://www.quora.com/What-is-the-difference-between-Green-Cabs-and-Yellow-Cabs
[3] https://www1.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf
[4] https://www1.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_green.pdf
[5] https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#split(java.lang.String)

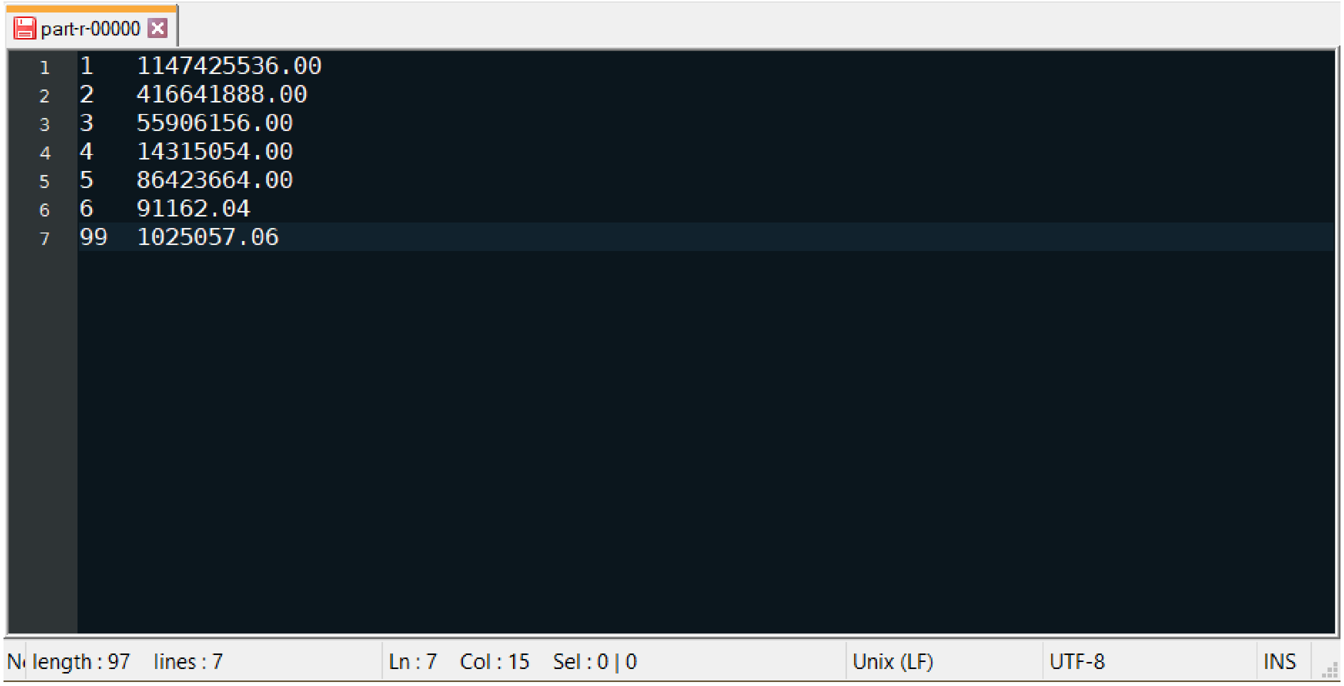
Appendix

Screenshot of resulting file for the Point 1



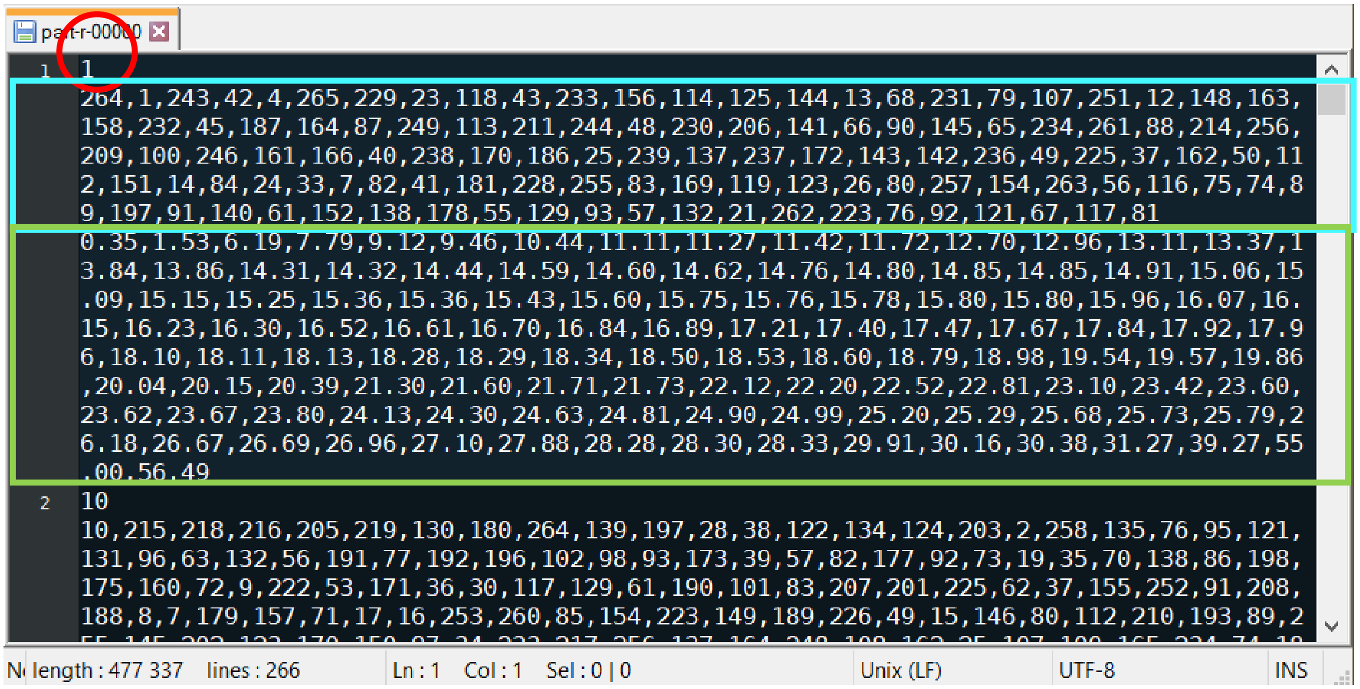
```
2016	239599597
2017	200484295
2018	175862428
```

Screenshot of resulting file for the Point 2



```
1    1147425536.00
2    416641888.00
3    55906156.00
4    14315054.00
5    86423664.00
6    91162.04
99   1025057.06
```

Screenshot of resulting file for the Point 3



```
1
264,1,243,42,4,265,229,23,118,43,233,156,114,125,144,13,68,231,79,107,251,12,148,163,
158,232,45,187,164,87,249,113,211,244,48,230,206,141,66,90,145,65,234,261,88,214,256,
209,100,246,161,166,40,238,170,186,25,239,137,237,172,143,142,236,49,225,37,162,50,11
2,151,14,84,24,33,7,82,41,181,228,255,83,169,119,123,26,80,257,154,263,56,116,75,74,8
9,197,91,140,61,152,138,178,55,129,93,57,132,21,262,223,76,92,121,67,117,81
0.35,1.53,6.19,7.79,9.12,9.46,10.44,11.11,11.27,11.42,11.72,12.70,12.96,13.11,13.37,1
3.84,13.86,14.31,14.32,14.44,14.59,14.60,14.62,14.76,14.80,14.85,14.85,14.91,15.06,15
.09,15.15,15.25,15.36,15.36,15.43,15.60,15.75,15.76,15.78,15.80,15.80,15.96,16.07,16.
15,16.23,16.30,16.52,16.61,16.70,16.84,16.89,17.21,17.40,17.47,17.67,17.84,17.92,17.9
6,18.10,18.11,18.13,18.28,18.29,18.34,18.50,18.53,18.60,18.79,18.98,19.54,19.57,19.86
,20.04,20.15,20.39,21.30,21.60,21.71,21.73,22.12,22.20,22.52,22.81,23.10,23.42,23.60,
23.62,23.67,23.80,24.13,24.30,24.63,24.81,24.90,24.99,25.20,25.29,25.68,25.73,25.79,2
6.18,26.67,26.69,26.96,27.10,27.88,28.28,28.30,28.33,29.91,30.16,30.38,31.27,39.27,55
.00,56.49
10
10,215,218,216,205,219,130,180,264,139,197,28,38,122,134,124,203,2,258,135,76,95,121,
131,96,63,132,56,191,77,192,196,102,98,93,173,39,57,82,177,92,73,19,35,70,138,86,198,
175,160,72,9,222,53,171,36,30,117,129,61,190,101,83,207,201,225,62,37,155,252,91,208,
188,8,7,179,157,71,17,16,253,260,85,154,223,149,189,226,49,15,146,80,112,210,193,89,2
```

Row composition for Point 3 file

Red: PULocationID
Light blue: sorted list of DOLocationID
Green: sorted list of trip_distance in ascending order relative to DOLocationID

Source code

Globals.java

```
package assignment;

public class Globals {

    //Setting the delimiter for the csv files
    public static final String DELIMITER = ",";

    //Setting menmonic constant to distinct the type of taxi based on the 
starting 5 letters of the file name
    public static String GREEN_TAXI = "green";
    public static String YELLOW_TAXI = "yello";

    //regex to identify files of yellow and green cabs produced before 
July 2016
    public static String REGEX_OLD = ".*_tripdata_2016-0[1-6].csv";

}
```

JobInit.java

```
package assignment;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

public class JobInit {

    public static void runJob(String[] args, Configuration conf, Class
class_main, Class class_mapper, Class class_reducer, Class
class_outputValue, String jobName) throws IOException,
InterruptedException, ClassNotFoundException {
        if(args.length == 3) {
            conf.set("years", args[2]);
        }
        Job job = Job.getInstance(conf, jobName);
        job.setJarByClass(class_main);
        job.setJobName(jobName);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setMapperClass(class_mapper);
        job.setReducerClass(class_reducer);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(class_outputValue);
        job.waitForCompletion(true);
    }
}
```

TaxiMapReduce_point1.java

```
package assignment.point1;

import assignment.JobInit;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.IntWritable;

import java.io.IOException;


public class TaxiMapReduce_point1 {

    public static void main(String[] args) {
        if((args.length != 2) && (args.length != 3)) {
            System.err.println("Usage:
assignement.point1.TaxiMapReduce_point1 <input path> <output path>
<year1,year2,...>(optional)");
            System.exit(-1);
        }

        Configuration conf = new Configuration();

        final long startTime = System.nanoTime();
        try {
            //Running the job for the point 1: A job returning the total
number of passengers in yellow and green taxi in 2018
            JobInit.runJob(args, conf, assignment.TaxiMapReduce.class,
assignment.point1.TaxiMapper_point1.class,
assignment.point1.TaxiReducer_point1.class, IntWritable.class,
"Count_passengers");
        } catch (IOException | InterruptedException |
ClassNotFoundException e) {
            e.printStackTrace();
        }

        final long duration = System.nanoTime() - startTime;
        System.out.println("Elapsed time (ms): " + duration / 1000000);
    }
}
```

TaxiMapper_point1.java

```
package assignment.point1;

import assignment.Globals;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

import java.io.IOException;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;
import java.util.regex.Pattern;

import static assignment.Globals.REGEX_OLD;

public class TaxiMapper_point1 extends Mapper<LongWritable, Text, Text,
IntWritable> {

    private static HashMap<String, Integer> hashMap_arrayPosition;
    private Set years_toSearch;

    @Override
    public void setup(Context context) {
        //Initialize from the beginning the position of certain features
in the array made up of exploded line, for each kind of taxi color
        hashMap_arrayPosition = new HashMap<>();
        hashMap_arrayPosition.put(Globals.GREEN_TAXI, 7);
        hashMap_arrayPosition.put(Globals.YELLOW_TAXI, 3);

        //Setting an Set that will contain all the years the user wants to
use inside the file. If no 3rd parameter is set, the default year is 2018
        years_toSearch = new HashSet();
        if(context.getConfiguration().get("years") != null) {
            String[] years =
context.getConfiguration().get("years").split(",");
            for(String year : years) {
                years_toSearch.add(year);
            }
        }
        else {
            years_toSearch.add("2018");
        }
    }

    public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
        //Getting the file name
        String fileName = ((FileSplit)
context.getInputSplit()).getPath().getName();

        //Getting the type of the taxi (green or yellow)
        String taxi_type = fileName.substring(0, 5);

        if (hashMap_arrayPosition.containsKey(taxi_type)) {
            //Line reading

            int passengerCount_fieldPosition =
hashMap_arrayPosition.get(taxi_type);

            //On green cabs files before July 2016, the position of the
field related to the number of passengers is 9 instead of 7
            if(Pattern.matches(REGEX_OLD, fileName) &&
taxi_type.startsWith("green")) {
                passengerCount_fieldPosition = 9;
            }

            String line = value.toString();

            //Avoids the first line which appears to be empty and avoids
the header by checking if the first word is "Vendor"
            if ((!line.isEmpty()) && (!line.startsWith("Vendor"))) {
                //Splitting the line using the default limiter
                String[] line_exploded = line.split(Globals.DELIMITER);

                //Extracting the year using the substring function,
casting it to Text object, assuming the second value of the row (first in
the array "line_exploded") as the right one (i.e. lpep_pickup_datetime)
                String year = line_exploded[1].substring(0, 4);

                //Accessing the Reduce section only for the selected years
(default: 2018)
                if (years_toSearch.contains(year)) {
                    //Getting the value of "passengers_count" which is
going to change depending of the type of the taxi
                    int number =
Integer.valueOf(line_exploded[passengerCount_fieldPosition]);

                    //Writing to the context the row
                    context.write(new Text(year), new
IntWritable(number));
                }
            }
        }
    }
}
```

TaxiReducer_point1.java

```
package assignment.point1;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class TaxiReducer_point1 extends Reducer<Text, IntWritable, Text,
IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {
        int total = 0;
        for (IntWritable value : values) {
            total += value.get();
        }
        context.write(key, new IntWritable(total));
    }
}
```

TaxiMapReduce_point2.java

```
package assignment.point2;

import assignment.JobInit;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.FloatWritable;

import java.io.IOException;


public class TaxiMapReduce_point2 {

    public static void main(String[] args) {
        if((args.length != 2) && (args.length != 3)) {
            System.err.println("Usage: 
assignement.point1.TaxiMapReduce_point2 <input path> <output path> 
<year1,year2,...>(optional)");
            System.exit(-1);
        }

        Configuration conf = new Configuration();

        final long startTime = System.nanoTime();
        try {
            //Running the job for the point 2: A job returning, for each 
rate code, the total amount charged to passengers of yellow and green taxi 
in 2018
            JobInit.runJob(args, conf, assignment.TaxiMapReduce.class, 
assignment.point2.TaxiMapper_point2.class, 
assignment.point2.TaxiReducer_point2.class, FloatWritable.class, 
"Charged_amount");
        } catch (IOException | InterruptedException | 
ClassNotFoundException e) {
            e.printStackTrace();
        }

        final long duration = System.nanoTime() - startTime;
        System.out.println("Elapsed time (ms): " + duration / 1000000);
    }
}
```

TaxiMapper_point2.java

```
package assignment.point2;

import assignment.Globals;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

import java.io.IOException;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;
import java.util.regex.Pattern;

import static assignment.Globals.REGEX_OLD;

public class TaxiMapper_point2 extends Mapper<LongWritable, Text, Text, 
FloatWritable> {

    private static HashMap<String, Integer> hashMap_arrayPosition;
    private Set years_toSearch;

    @Override
    public void setup(Context context) {
        //Initialize from the beginning the position of certain features 
in the array made up of exploded line, for each kind of taxi color
        hashMap_arrayPosition = new HashMap<>();
        hashMap_arrayPosition.put(Globals.GREEN_TAXI, 4);
        hashMap_arrayPosition.put(Globals.YELLOW_TAXI, 5);

        //Setting an Set that will contain all the years the user wants to 
use inside the file. If no 3rd parameter is set, the default year is 2018
        years_toSearch = new HashSet();
        if(context.getConfiguration().get("years") != null) {
            String[] years = 
context.getConfiguration().get("years").split(",");
            for(String year : years) {
                years_toSearch.add(year);
            }
        }
        else {
            years_toSearch.add("2018");
        }
    }
    public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
        //Getting the file name
        String fileName = ((FileSplit)
context.getInputSplit()).getPath().getName();

        //Getting the type of the taxi (green or yellow)
        String taxi_type = fileName.substring(0, 5);

        if (hashMap_arrayPosition.containsKey(taxi_type)) {
            //Reading the line
            String line = value.toString();

            //Avoids the first line which appears to be empty and avoids
the header by checking if the first word is "Vendor"
            if ((!line.isEmpty()) && (!line.startsWith("Vendor"))) {
                //Splitting the line using the default limiter
                String[] line_exploded = line.split(Globals.DELIMITER);

                //Extracting the year using the substring function,
assuming the second value (first in the array "line_exploded") as the
right one (i.e. lpep_pickup_datetime)
                String year = line_exploded[1].substring(0, 4);

                //Accessing the Reduce section only for the selected years
(default: 2018)
                if (years_toSearch.contains(year)) {
                    int ratecodeID_fieldPosition =
hashMap_arrayPosition.get(taxi_type);

                    int totalAmount_fieldPosition = 16;

                    //On green cabs files before July 2016, the position
of the field related to the number of passengers is 9 instead of 7
                    if(Pattern.matches(REGEX_OLD, fileName) &&
taxi_type.startsWith("green")) {
                        totalAmount_fieldPosition = 18;
                    }
                    else if(Pattern.matches(REGEX_OLD, fileName) &&
taxi_type.startsWith("yello")) {
                        ratecodeID_fieldPosition = 7;
                    }

                    //Getting the value of "RatecodeID" which is going to
change depending of the type of the taxi
                    String rateCode =
line_exploded[ratecodeID_fieldPosition];

                    //Extracting the total amount, casting it to integer,
assuming the seventeenth value of the row (sixteenth in the array
"line_exploded") as the right one (i.e. RatecodeID)
```

```
                    float totalAmount = 
Float.valueOf(line_exploded[totalAmount_fieldPosition]);

                    //Writing to the context the row
                    context.write(new Text(rateCode), new 
FloatWritable(totalAmount));
                }
            }
        }
    }
}
```

TaxiReducer_point2.java

```
package assignment.point2;

import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class TaxiReducer_point2 extends Reducer<Text, FloatWritable, Text,
Text> {
    public void reduce(Text key, Iterable<FloatWritable> values, Context
context) throws IOException, InterruptedException {
        float total = 0;
        for (FloatWritable value : values) {
            total += value.get();
        }
        String total_formatted = String.format("%.2f", total);
        context.write(key, new Text(total_formatted));
    }
}
```

TaxiMapReduce_point3.java

```
package assignment.point3;


import assignment.JobInit;

import org.apache.hadoop.conf.Configuration;

import java.io.IOException;


public class TaxiMapReduce_point3

{

    public static void main(String[] args) {

        if((args.length != 2) && (args.length != 3)) {

            System.err.println("Usage: assignement.TaxiMapReduce <input
path> <output path> <year1,year2,...>(optional)");

            System.exit(-1);

        }



        Configuration conf = new Configuration();



        final long startTime = System.nanoTime();

        try {

            //Running the job for the point 3: A job returning, for each
PULocationID, the list of related DOLocationID for yellow taxi in 2018,
ordered by increasing average trip distance

            JobInit.runJob(args, conf, TaxiMapReduce_point3.class,
TaxiMapper_point3.class, TaxiReducer_point3.class, RideWritable.class,
"Do_location");

        } catch (IOException | InterruptedException |
ClassNotFoundException e) {

            e.printStackTrace();

        }
        final long duration = System.nanoTime() - startTime;

        System.out.println("Elapsed time (ms): " + duration / 1000000);

    }

}
```

TaxiMapper_point3.java

```
package assignment.point3;


import assignment.Globals;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.lib.input.FileSplit;


import java.io.IOException;

import java.util.HashSet;

import java.util.Set;

import java.util.regex.Pattern;


import static assignment.Globals.REGEX_OLD;


public class TaxiMapper_point3 extends Mapper<LongWritable, Text, Text, 
RideWritable> {

    //Initializing the feature names of yellow taxi with their positions 
inside the array of split line

    private static int TRIPDISTANCE = 4;

    private static int PULocationID = 7; //not existent before 2016-07

    private static int DOLocationID = 8; //not existent before 2016-07


    private Set years_toSearch;


    @Override

    protected void setup(Mapper.Context context) {
        //Setting an Set that will contain all the years the user wants to
use inside the file. If no 3rd parameter is set, the default year is 2018

        years_toSearch = new HashSet();

        if(context.getConfiguration().get("years") != null) {

            String[] years =
context.getConfiguration().get("years").split(",");

            for(String year : years) {

                years_toSearch.add(year);

            }

        }

        else {

            years_toSearch.add("2018");

        }

    }



    public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

        //Getting the file name

        String fileName = ((FileSplit)
context.getInputSplit()).getPath().getName();



        //Checking if the file row is related to a yellow taxi

        if(fileName.startsWith(Globals.YELLOW_TAXI) &&
(!Pattern.matches(REGEX_OLD, fileName))) {

            //Reading the line

            String line = value.toString();



            //Avoids the first line which appears to be empty and avoids
the header by checking if the first word is "Vendor"

            if((!line.isEmpty()) && (!line.startsWith("Vendor"))) {
                //Splitting the line using the default limiter

                String[] line_exploded = line.split(Globals.DELIMITER);



                //Extracting the year using the substring function, 
assuming the second value (first in the array "line_exploded") as the 
right one (i.e. lpep_pickup_datetime)

                String year = line_exploded[1].substring(0, 4);



                //Accessing the Reduce section only for the selected years 
(default: 2018)

                if(years_toSearch.contains(year)) {

                    //Extracting the value of "PULocationID", 
"DOLocationID", "trip_distance"

                    String PULocationID_current = 
line_exploded[PULocationID];



                    RideWritable rideWritable = new 
RideWritable(PULocationID_current, line_exploded[DOLocationID], 
Float.valueOf(line_exploded[TRIPDISTANCE]));



                    //Writing to the context the row

                    context.write(new Text(PULocationID_current), 
rideWritable);

                }

            }

        }

    }

}
```

TaxiReducer_point3.java

```
package assignment.point3;


import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;


import java.io.IOException;
import java.util.*;


import static java.util.Map.Entry.comparingByValue;
import static java.util.stream.Collectors.toMap;


public class TaxiReducer_point3 extends Reducer<Text, RideWritable, Text,
Text> {


    public void reduce(Text key, Iterable<RideWritable> values, Context
context) throws IOException, InterruptedException {


        HashMap hashMap_DO = new HashMap<String, ArrayList<Float>>();


        //Creation of a HashMap made up of DOLocation as key and an array
of trip distances as values, in order to make a more comfortable average
in the next step
        for (RideWritable value : values) {
            String DOLocationID_current =
value.getDOLocationID().toString();


            if(hashMap_DO.containsKey(DOLocationID_current)) {
                ArrayList arrayList_DOLocationID_trip = (ArrayList<Float>)
hashMap_DO.get(DOLocationID_current);
arrayList_DOLocationID_trip.add(value.getTrip_distance().get());

                hashMap_DO.put(DOLocationID_current,
arrayList_DOLocationID_trip);

            }

            else

            {

                ArrayList<Float> arrayList = new ArrayList<>();

                arrayList.add(value.getTrip_distance().get());

                hashMap_DO.put(DOLocationID_current, arrayList);

            }

        }



        //Unrolling the HashMap to store in another HashMap DOLocation
with the average trip distance

        HashMap hasMap_DOLocations_avg = new HashMap<String, Float>();



        Iterator it = hashMap_DO.entrySet().iterator();

        while(it.hasNext()) {

            Map.Entry entry = (Map.Entry<String,
ArrayList<Float>>)it.next();

            ArrayList<Float> arrayList_trips = (ArrayList<Float>)
entry.getValue();

            float avg = 0;

            if(arrayList_trips.size() > 1) {

                avg = calculate_average(arrayList_trips);

            }

            else {

                avg = arrayList_trips.get(0);

            }
            hasMap_DOLocations_avg.put(entry.getKey(), avg);

        }



        //Sorting the HashMap

        LinkedHashMap<String, Float> hashMap_DOLocations_avg_sorted =
sort_map(hasMap_DOLocations_avg);

        //Unrolling the LinkedHashMap

        StringBuilder stringBuilder_output_DOLocations = new
StringBuilder();

        StringBuilder stringBuilder_output_averages = new StringBuilder();

        Iterator it_final =
hashMap_DOLocations_avg_sorted.entrySet().iterator();

        while(it_final.hasNext()) {

            Map.Entry entry = (Map.Entry<String,
ArrayList<Float>>)it_final.next();

            stringBuilder_output_DOLocations.append(entry.getKey() + ",");

            stringBuilder_output_averages.append(String.format("%.2f",
entry.getValue()) + ",");

        }



        String outputString =
chopLastChar(stringBuilder_output_DOLocations) + "\t" +
chopLastChar(stringBuilder_output_averages);

        context.write(key, new Text(outputString));

    }



    public float calculate_average (ArrayList<Float> list) { //Available
on java 8

        OptionalDouble average = list

                .stream()

                .mapToDouble(a -> a)
                .average();
        return (float)average.getAsDouble();
    }


    public LinkedHashMap sort_map(HashMap<String, Float>hashMap) {
//Available on java 8
        //Let's sort this map by values in an ascending fashion
        LinkedHashMap<String, Float> sorted = hashMap
                .entrySet()
                .stream()
                .sorted(comparingByValue())
                .collect(
                        toMap(e -> e.getKey(), e -> e.getValue(), (e1, e2)
-> e2,
                                LinkedHashMap::new));
        return sorted;
    }


    public String chopLastChar(StringBuilder stringBuilder) {
        String string = stringBuilder.toString();
        return string.substring(0, string.length() - 1);
    }
}
```

RideWritable.java

```
package assignment.point3;

import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.WritableComparable;

import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

public class RideWritable implements WritableComparable {

    private Text PULocationID;
    private Text DOLocationID;
    private FloatWritable trip_distance;

    //An empty constructor is required by hadoop
    public RideWritable(){}

    public RideWritable(String PULocationID, String DOLocationID, float
trip_distance) {
        this.PULocationID = new Text(PULocationID);
        this.DOLocationID = new Text(DOLocationID);
        this.trip_distance = new FloatWritable(trip_distance);
    }

    public Text getPULocationID() {
        return PULocationID;
    }

    public void setPULocationID(String PULocationID) {
        this.PULocationID = new Text(PULocationID);
    }

    public Text getDOLocationID() {
        return DOLocationID;
    }

    public void setDOLocationID(String DOLocationID) {
        this.DOLocationID = new Text(DOLocationID);
    }

    public FloatWritable getTrip_distance() {
        return trip_distance;
    }

    public void setTrip_distance(float trip_distance) {
        this.trip_distance = new FloatWritable(trip_distance);
    }

    @Override
    public int compareTo(Object o) {
        return 0;
    }

    @Override
    public void write(DataOutput dataOutput) throws IOException {
        dataOutput.writeUTF(PULocationID.toString());
        dataOutput.writeUTF(DOLocationID.toString());
        dataOutput.writeFloat(trip_distance.get());
    }

    @Override
    public void readFields(DataInput dataInput) throws IOException {
        PULocationID = new Text(dataInput.readUTF());
        DOLocationID = new Text(dataInput.readUTF());
        trip_distance = new FloatWritable(dataInput.readFloat());
    }
}
```