INFO-H-600 - Project Report
Group 3

Bilal Kostet, Pierre Hosselet, Antoine Somerhausen
Pacome Van Overschelde, Romain Vandepopeliere

January 20, 2023

Contents

1 Introduction **1**
1.1 Comment on the json file . . . 2
1.2 Comment on the csv file . . . 2

2 Extraction of the zones from the json file **2**
2.1 Dask . . . 2
2.2 PySpark . . . 5
2.3 Improvement . . . 5

3 First question **5**
3.1 Dask . . . 5
3.2 Comment on the parallelization in Dask . . . 6
3.3 PySpark . . . 7

4 Second question **8**
4.1 Dask . . . 8
4.2 Comment on the parallelization in Dask . . . 11
4.3 PySpark . . . 18

5 Results and timing of the computations for Dask and PySpark **20**
5.1 First question . . . 20
5.2 Second question . . . 20
5.3 Timing . . . 21

1 Introduction

In this work, we have been provided with a .csv file containing data from GPS unit of mobile phones of drivers, each line of the file giving the position of the driver. A .json file is also included, dividing the area where those GPS are located into separate zones. The goal is to process the data contained in the file to describe the driving habits of drivers in these zones.

Each line of the .csv file contains the following information: a unique driver ID (string), a timestamp (Timestamp) corresponding to the time of the event, the latitude of the driver (Double) and the longitude of the driver (Double).

The structure of the zones data is given by the ID of the zone, followed by a polygon defined by 5 points in the plane (x,y).

We have chosen Dask as our main tool, even if for both questions we decided to code our data processing methods in both Dask and PySpark, in order to compare the two frameworks. We will however mostly discuss the parallelization strategy for Dask, as this package provides tools to visualize easily how our data processing pipeline is parallelized.

As a final remark, note that we could have changed the configuration when starting both the Dask client or the PySpark session. Indeed, we can customize a lot of features such as the driver memory, the workers node distribution or the partition size of the dataframes that we are manipulating. However, there is no need to deviate too much from the default configuration due to the relatively small size of the original driver dataset.

1.1 Comment on the json file

The last of these 5 points is always the same as the first one. We can also easily verify by plotting randomly sampled polygons that they are always squares. This could lead to a simplified process to check if a data point fits into a certain zone square. For the sake of generality, we will however not make this assumption so that our code would work with any type of polygon provided.

1.2 Comment on the csv file

The .csv file containing the drivers data is a bit less than 250 MB, which means that it could fit entirely into the RAM of modern computers. The Dask documentation indicates : "Dask DataFrame may not be the best choice in the following situations: If your dataset fits comfortably into RAM on your laptop, then you may be better off just using pandas. There may be simpler ways to improve performance than through parallelism". In this case, we still proceeded with parallelization for a few reasons: firstly, the point of the project was to learn how to parallelize programs. Secondly, we may want our code to be scalable to data bigger than the size of our RAM, scalability being one of the main advantages of parallelization frameworks such as Dask and PySpark.

Before getting into deeper manipulations of the .csv file, we can make some general observations on the data:

1. For the same driver, there can be several data. In average, there are 278 datapoints per driver.
2. The times are between 12:00 and 19:00 of the same day.
3. Some geographical locations are out of the provided zones.

2 Extraction of the zones from the json file

2.1 Dask

Extracting data from a .json file is a simple feat in pandas thanks to the pandas.json_normalize API. This is however not an easy task in Dask, as the json_normalize was not adapted in this framework. In order to extract the zones in an exploitable way from the `zones.json` file, we thus used the following code:

```
def polyzone(dbag):
    polygonpts=[]
    for i in range(len(dbag['polygon'])):
        polygonpts.append((float(dbag['polygon'][i]['lat']),float(dbag['
            polygon'][i]['lng'])))
```

```
    return {
        'id_zone' : dbag['id_zone'],
        'polygonpts' : polygonpts
    }
zones=db.read_text('ProjectData/zones.json').map(json.loads)
zones=zones.pluck('zones').flatten().map(polyzone)
dzones=zones.to_dataframe().set_index('id_zone',sorted=True)
```

In this code, we first read the `zones.json` file as a string into a Dask Bag, a versatile Dask data structure, easier for pre-processing. This string is mapped into a dictionary thanks to the `json.loads` function; this dictionary is nested, and all of the data is under the first key, `zones`, so that we plug this key, and we then apply the custom function `polyzone` to extract the data. This function extracts the identifier of a zone through the value of the id_zone key quite easily. This zone identifier is simply an integer going from 1 to 51. It also creates a list called `polygonpts`, where we will append tuples containing the values of latitude and longitude for each point forming the polygon that defines the concerned zone. The data is now formatted conveniently and we export this structure into a Dask dataframe, which is more efficient as it uses pandas routines for processing. We also set the zone ID column as an index in our dataframe: operations such as sorting are much more efficient on an index than on a regular column.
Considering the small amount of zones in this file (51), this is probably not necessary, and we could probably use a pandas dataframe instead of a Dask dataframe, but we still chose to proceed this way for the sake of scalability.

In order to manipulate these zones, we will use the Python package `Shapely`, which allows us to define new objects: Points and Polygons. A Point is a tuple of values defining the x-y coordinates of the points, and a Polygon is a list of Points defining the Polygon. This package is highly useful in our case as the object class Polygon possesses a method called `contains()`, which allows to check if a Point is contained within a Polygon with the syntax Polygon.contains(Point).

We may want to visualize where these zones are located in reality, as it could help us confirm our interpretation of the data. For this, we will be using the packages `GeoPandas` and `contextily`. For this, we have to recreate the `polygonlist` list, inverting latitude and longitude compared to before, as contextily takes longitude before latitude, as opposed to usual conventions. We could have taken the previous ptslist and inverted the values contained in the tuples, as this would probably have been more efficient. However, this part is not computationally heavy, and only needs to be done once to create the map, so optimizing is not as crucial.

```
polygonlist = []
for i,j in dzones.iterrows():
    ptslist = []
    for item in j.values.tolist()[0]:
        ptslist.append((item[1], item[0]))
    polygonlist.append(Polygon(ptslist))
```

After that, we will export our list of Polygons into a GeoPandas dataframe (a GeoDataFrame). The key crs (coordinate reference system) takes the value epsg:4326, which corresponds to the latitude/longitude coordinate system. We then assign a new column `NAME` to this dataframe, with integer increasing by one at each new polygon, labelling the zones from 1 to 51. We convert the crs of this dataframe to the Web Mercator projection coordinates, which is the standard for web map applications. After that, we define a column named `coords` by a shapely Polygon method named representative_point(). As defined in the manual, this method "returns a cheaply computed point that is guaranteed to be within the geometric object". This point will be used as coordinates for the label of the zone (the `NAME` value). We now only have to plot these zone with the GeoPandas method .plot(), we annotate the zones with their labels with plt.annotate(), and we finally add a background map of the zone thanks to the contextily function add_basemap(), which creates

a background map from the coordinates of our plotted polygons.

```
gdf = gpd.GeoDataFrame(crs='epsg:4326', geometry=polygonlist)
gdf = gdf.assign(NAME=range(len(gdf.geometry)))
gdf.NAME = gdf.NAME.add(1)
gdf_web = gdf.to_crs(epsg=3857)
gdf_web['coords'] = gdf_web['geometry'].apply(lambda x: x.representative_point
    ().coords[:])
gdf_web['coords'] = [coords[0] for coords in gdf_web['coords']]
ax = gdf_web.plot(figsize=(10, 10), alpha=0.5, edgecolor='k')
plt.axis('off')
for idx, row in gdf_web.iterrows():
    plt.annotate(text=row['NAME'], xy=row['coords'], horizontalalignment='
        center')
cx.add_basemap(ax, alpha=1, source=cx.providers.OpenStreetMap.DE, zoom=10)
```

We obtain the map shown in Fig. 1, indicating that the region covered by our zones is in fact Lima, the capital of Peru. An interesting fact is that Lima is famous for being one of the cities with the worst traffic jams in the world.

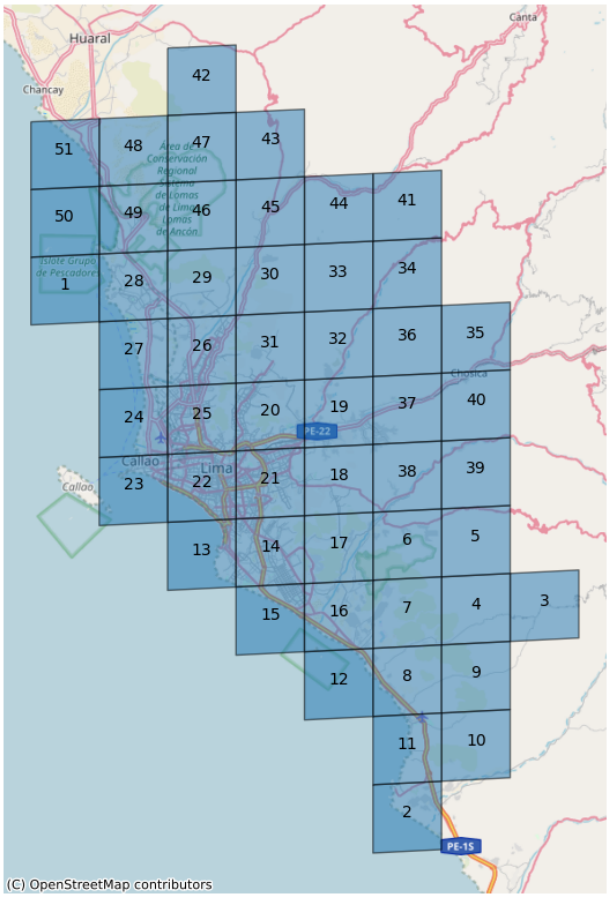

Figure 1: Map of the zones.

2.2 PySpark

Even if PySpark has a function that can read a .json file, we choose to read it via pandas for the sake of simplicity, since the PySpark part is not our main goal in this work. Also, the dataframe has 51 rows, hence is quite small. The interest of using PySpark will of course be in the processing of the big driver dataframe. As we know, pandas gives us have access to `json.normalize()` to easily create the dataframe. One could completely unpack the nested rows but it won't be necessary for our `findzone` function.

2.3 Improvement

To determine in which zone a point is located, in the worst case, we will have to perform 51 operations, i.e. one operation per zone. This is actually the computationally heaviest part of our project.
We can look for potentially more efficient strategies to find in which zone a point is located than to list them in order.

We propose some ideas that we could have put in place:

- **First suggestion:** If we assume that our dataset is uniformly distributed, we could take a small statistically representative sample and look at the most visited areas on this sample. To determine in which area a point is located we would follow the order determined by the sample.
- **Second suggestion:** Create a binary tree. Each of the first two branches correspond to a combination of the two subgroups of zones. This is particularly interesting in our case, as the package shapely allows to obtain the union of multiple Polygons. It would be then easy to verify if a Point is in this Polygon made up of the union of the subset of zone Polygons. Proceeding along the branches, we gradually reduce the subset of zones forming the unioned Polygon. There would be then a maximum of 6 operations to determine in which zone we are.
- **Third suggestion:** Merging together the two previous suggestions. From the first suggestion we would create a binary tree where we group the most probable zones together, so that the probabilities on each branch are distributed as evenly as possible and we optimize the average amount of operations needed to find the zone.

3 First question

In this question, we are asked to find the 10 zones that are visited by the most drivers. This question leaves some room for interpretation: should a driver be counted only once per zone, should we could each time the driver visits the zone or should we count every GPS information?

Each approach has advantages and disadvantages. As we don't have much information on how the GPS points were taken, if we do not restrict the data, some drivers might have more weight than others on the results if, for example, the number of GPS points taken is related to the quality of the network and therefore could be correlated with the proximity of a network antenna. The first approach does not take into account the time spent in the area, so that it makes no difference if a driver is just passing through or if he spends all day in that zone.

We however chose to proceed with this case, where each unique driver is only counted once per zone visited, as we feel it answers better the question.

3.1 Dask

We first create a list called `polygonlist` containing all the Polygon objects defining the zones, creating it from the coordinates obtained in the `dzones` dataframe. We load the data from the GPS units into a

dataframe called `gps`, where each of the different informations contained into the csv are loaded into separate columns, respectively named `driver`, `timestamp`, `latitude` and `longitude`. We specify manually the types of the data stored in our columns, in order to avoid any possible ambiguity and issues with Dask misunderstanding it. We chose `float64` for the latitude and longitude columns, the driver identifier is given by a `string`, and we use the parse_dates option to specify that the column timestamp has a type `datetime64[ns]`.

We apply the custom function `findzone` to each row of this `gps` dataframe to create a new column in our dataframe named `zone`, containing the identifier of the zone. The function `findzone` simply takes the values of the latitude and longitude columns of the row, creates a Point with it, and then checks if this Point fits in any of the Polygons. If so, it returns an integer corresponding to the identifier of the zone. If the Point does not fit into any Polygon, which is possible because of GPS jittering, the function returns 0 as zone identifier.

Now that our dataframe has a column indicating the zone in which every driver position lies, we can compute the top ten of most visited zones. We restrict our dataframe to the only two columns of interest, `driver` and `zone`, and apply the method drop_duplicates() to only keep unique pairs of (driver,zone). Restricting ourselves to the columns of interest increases significantly the speed of the computation, rather than using the (subset=[columns]) key in drop_duplicates(). We then group the remaining values by values of the zone column with groupby('zone'), to which we append the count() aggregation method that counts how many values were in each group of same zone value. This gives us a dataframe with the zone identifier as an index and the number of unique drivers having visited that zone in the `driver` column. Finally, we obtain the 10 largest values in this dataframe with .nlargest(10). This is more performant than sorting all the values with sort_values(), but will not make a noticeable different for only 51 zones like in this case.

One comment: this process includes the zone identifier 0 which corresponds to GPS points not belonging to any zone, due to GPS jittering. If we wanted to exclude this zone, we could simply add the line `gps = gps[gps.zone > 0]` right after defining the column `zone` with the function `findzone`.
If we wanted to obtain this count not for unique drivers, but for every GPS value in each zone, we would only need the remove the .drop_duplicates() function.

```
def findzone(row):
    point=Point(row.latitude,row.longitude)
    for i in range(len(polygonlist)):
        polygon=Polygon(polygonlist[i])
        if polygon.contains(point):
            return i+1
    return 0

polygonlist=[]
for i,j in dzones.iterrows():
    polygonlist.append(Polygon(j.values.tolist()[0]))

gps=dd.read_csv('ProjectData/drivers.csv', dtype={'latitude' : 'float64', '
    longitude' : 'float64', 'driver' : 'string'},parse_dates=['timestamp'])
gps['zone'] = gps.apply(findzone, axis=1).astype('int32')
gps_topten = gps[['driver','zone']].drop_duplicates().groupby('zone').count().
    nlargest(11)
```

3.2 Comment on the parallelization in Dask

For both approaches (dropping the duplicates to get unique drivers, or not dropping to get every GPS point), a chart visualization of the data processing pipeline is shown in Section 4.2, Fig. 2.

The beginning of the parallelization is identical. In both cases, it consists in reading the .csv file and assigning to each driver a zone. We observe that in the first case the groupby is performed by all workers, before being grouped on a worker to aggregate the results, while in the second case the groupby is performed directly by a single worker.
Taking a look at the data obtained after the drop_duplicates(), it appears that it reduces drastically the amount of data. This is probably why in the second algorithm, after having done the drop_duplicates(), Dask estimates that it is less expensive to leave it on the same worker and have it apply the groupby than to shuffle the data and split it again on several workers and to do the task in parallel. If our file size was much larger, for example if we had to process a much larger amount of different drivers, Dask could have decided to parallelize again on different workers after the drop_duplicates() to perform the the groupby().

3.3 PySpark

The first thing to do is to define a function that will use the package `Shapely` in order to know in which zone a coordinate point is. We must create a `Shapely.Point` and the `Shapely.Polygon` objects out of the zone coordinates. It will go through all the rows of the zones dataframe, so that we know for sure that this will be expensive timewise.

```
def find_zone(latitude,longitude):

    #create a shapely Point object from the driver's coordinates
    point = sg.Point(latitude,longitude)

    #checking if the point is in a given zone
    for i in range(len(zones)):
        polygon= sg.Polygon( (x['lat'],x['lng']) for x in zones['polygon'].
            iloc[i])
        if polygon.contains(point):
            return int(zones['id_zone'].iloc[i])

    #if no zone contains the point
    return None
```

But, in order to use in a PySpark dataframe a function which is not from the `pyspark.sql` library, we need to convert it into a User Defined Function (= `udf`). Now we can add the zone column to the driver dataframe. We choose to get rid of the GPS points that are not within the 51 zones, this is easily done with a `na.drop()` that will throw away the zone values that are null.

```
driversz = drivers.withColumn('zone', find_zone_udf(col('latitude'), col('
    longitude'))).na.drop()
```

In PySpark, the build-in functions of the `pyspark.sql` library are very well optimized. It is however not the case for the custom functions that the user can define. As said in the PySpark documentation, they can be detrimental to optimization and make the data processing really slow. The creation of the zone column takes no so much time per say; what really takes time is a wide transformation performed on a dataframe which has undergone an `udf` transformation. Recall that the Spark transformations are divided into two parts : wide and narrow. The wide transformations need redistribution of the data partitions in the network between the executing nodes, whereas the narrow transformations don't. The `groupBy()` and `join()` operations are two wide transformations that we will use extensively in this work. We noticed how slow it is to use such transformations that need a shuffle on our new dataframe `driversz`. The timing and performances will be mostly discussed in Sec5, but one can anticipate : this Question 1 takes 35min to run and is supposed to be the quickest one. For this very reason, we chose for this PySpark part to write a new .csv file with the zone column included and to use it as a new starting point.[1] The writing with the help of

[1]This will of course cut the data pipeline so that it will be shorter than in the Dask case.

the `write()` function takes approximately 30min to run but we will save time at the end of the day, as we will see later on. Note that each worker node writes his own data partition, so that we end up with 8 .csv files that we should recombine in one. One could force the `write()` function to give us one .csv file only, but one must be sure that the executer node can handle the whole dataset without running into an *out of memory* error; our new PySpark dataframe with the zones is called `driversZ`.

Now we can properly answer the Question 1. We exhibit the different pairs (driver,zone) that are in the dataframe we group them by zone by counting how many unique drivers there are by zone.

```
driversByZone = driversZ.dropDuplicates(['driver','zone']).groupBy('zone').
    count().cache()
```

A useful feature of Spark is that one can cache a result so there will be no need to recompute the result for further operations which involve this result. One must however be careful : it is recommended to use the `cache()` function on small datasets only, typically dataset that have already undergone a size reduction through a filtering or a grouping operation for instance (otherwise the memory will be saturated). Also, since it costs time to save the information, one must be sure that the output will be reused afterwards. These two conditions are reunited in this case, so that we can surely cache our result. Note that `cache()` is a transformation and not an action : the dataframe is only cached when an action is called. In order to properly answer the question and to cache the resulting dataframe, we write :

```
driversByZone.orderBy('count', ascending=False).show(10)
```

Results will be discussed in Sec.5.

4 Second question

For the second question, we were left free to propose other analyses that could be performed with the same data. For these analyses, we remind that the zones shown in Fig. 1 all have the same area and have quite a big area compared to the city they canvas. We have chosen to perform the following ones:

- **First analysis**: How much time do drivers spend in each zone in average? This could help us identify zones of traffic congestion. What is the average distance traveled by drivers in each zone? This could indicate a zone where drivers go around in a complex network of small routes, staying in the same zone, as in a very urbanized city center. What are the mean speeds of drivers in each zone? Can we identify zones corresponding to high-speed highways?

- **Second analysis**: For each zone, how many are visited in average by its drivers? This can give us an indication of which zones are merely zones of transit to other zones, and which are dense urbanized zones where people will not have to travel far and will typically stay inside or near their zone.

- **Third analysis**: For each zone, how many times will its drivers change zones in average?

4.1 Dask

- **First analysis**:

We begin by cleaning the `gps` dataframe into `gpsclean` by removing the lines corresponding to drivers not belonging to any of the 51 zones (indicated by a zero value in the `zone` column).

We proceed to groupby this dataframe by pairs of (driver, zone) values. We then apply the custom defined function `speed_compute` to each of the sub-dataframe composed of all the values for a specific pair of (driver, zone).

This function will compute three things: the average time spent by drivers in a each zone, the average distance driven by drivers in each zone, and finally the average speed of drivers in each zone.

To do so, we first sort the sub-dataframe by timestamps, so that the driver positions are chronologically ordered. We compute the time between two drivers positions in a new column named `delay`, found by the difference between each row and the previous one of the sub-dataframe of the timestamp value, converted to seconds by the method .dt.totalseconds(). The first row returns NaN, by default. Computing the distances between each driver location is more complicated. To determine the distance between two pairs of (latitude, longitude) coordinates, we have decided to compute them in the `distances` column using the Haversine formula. This approximation assumes the Earth to be a sphere of radius R = 6371 km, which is precise up to 0.5%:

$$d = 2R \arcsin\left(\sin^2\left(\frac{\text{lat}_2 - \text{lat}_1}{2}\right) + \cos\text{lat}_1 \cos\text{lat}_2 \sin^2\left(\frac{\text{long}_2 - \text{long}_1}{2}\right)\right).$$

The "instantaneous" speed of the drivers at their position is then computed by dividing the column `distances` with `delay`, and multiplying the values by 3600 to obtain speeds in km/h. These "instantaneous" speeds will not be used to compute the average speed. Instead, we will only use them to clean the dataframe further of rows with infinite speeds (meaning that the timestamp was identical for both rows, so it was probably a duplicate or a glitch), as well as filtering rows with speeds over 200 km/h. This second filter is needed because we noticed some glitches where coordinates would jump erratically between two far away positions in a short amount of time (a 15 km jump in one second by driver 13aa2cb9, for example). The third filter is used because some drivers visit the same zone multiple times, so that in this sub-dataframe containing the data belonging to a pair (driver, zone), we might see a high jump in timestamp corresponding to the return of the driver in this same zone after the first set of values coming from its first visit. In order to avoid complicated operations within the restrictions of the tools proposed by Dask, and by exploring the data, we have chosen as a criteria to filter values of the delay above 20 min. The rows corresponding to a re-entry in the zone would be lost in any case, as previously with the .diff() method.
We obtain the total time spent in seconds by the driver in the zone by summing all the values in the `delay` column. This gives a scalar that we store in the column `total_time`. This is not optimal, as this value is repeated for each row of the sub-dataframe. This will be causing issues afterwards, as we will need to compute the mean of these total times. A (driver, zone) pair with more point will have a `total_time` weighing more in the average than another with less data points. To avoid this issue, we only keep the value in the first row of the column, and replace all the following rows with NaNs (we will see why when computing the mean speed). It should be tested whether this way is faster or not that defining a new dataframe containing the total time uniquely for each (driver, zone) pair.
We proceed in the exact same manner to compute the total distance traveled by each driver in each zone, summing the values of the `distances` column into the `total_distance` one, and replacing all the values except in the first row with NaNs.
We can finally compute our value for the average speed of each driver in each zone by dividing the total distance value by the total time, and multiplying by 3600 to obtain speeds in km/h. This is however way faster than computing the mean speed for the driver in the zone from the "instantaneous" speeds. Indeed, this would require to computing a weighted average of the `speeds` column values, taking as weight the times in the `delay` column. Taking a weighted average is much more computationally heavy than dividing two columns.
To speed up further computations, we drop all the columns containing information we don't need anymore. We grouped by the columns `driver` and `zone`, which creates new indices in our dataframe following the values of these two columns for each group. However, we will want to group again by `zone` in the future. Because of this, we will not drop the column `zone` but instead rename it to `dzone`, to avoid any confusion between the index and the column.
Following this new `groupby`, we apply the mean() aggregation method to obtain the mean of the columns left: `total_time`, `total_distance` and `meanspeed`. This is where filling those columns with NaNs comes in handy, as NaNs are not taken into account while computing the mean by dask and pandas routines. We end by sorting this dataframe by decreasing values of `meanspeed`.

```
gpsclean=gps[gps.zone > 0]
```

```
def speed_compute(x):
    x=x.sort_values('timestamp')
    x['delay'] = x.timestamp.diff().dt.total_seconds()
    x['distances'] = da.arcsin(da.sqrt(da.sin(da.radians(x.longitude.diff().
        mul(0.5))).pow(2).mul(da.cos(da.radians(x.latitude))).mul(da.cos(da.
        radians(x.latitude.shift(periods=1)))).add(da.sin(da.radians(x.
        latitude.diff().mul(0.5))).pow(2)))).mul(2*6371)
    x['speeds'] = x.distances.div(x.delay).mul(3600)
    x=x[~(da.isinf(x.speeds) | (x.speeds > 200) | (x.delay > 900))]
    x['total_time'] = x['delay'].sum()
    x['total_time'].iloc[1:] = np.nan
    x['total_distance'] = x['distances'].sum()
    x['total_distance'].iloc[1:] = np.nan
    x['meanspeed'] = x['total_distance'].div(x['total_time']).mul(3600)
    x=x.drop(columns=['driver','latitude','longitude','distances','delay','
        timestamp','speeds']).rename(columns={'zone':'dzone'})
    return x

gps_speeds=gpsclean.groupby(['driver','zone']).apply(speed_compute).groupby('
    dzone').mean().sort_values('meanspeed',ascending=False)
```

- **Second analysis**:

In this case, we want to obtain the count of how many different zones are visited in average by its drivers. We do not take into account multiple passages of a driver through a zone, and only consider whether a driver has ever visited a zone or not.
We start by restricting ourselves to the only two columns that we will need, `driver` and `zone`. We apply .drop_duplicates() to these two columns to keep only unique pairs of (driver, zone) values. We then group by `driver`, which means that we obtain a new dataframe with the driver ID as index, and where groups are formed for each unique driver containing a row for each zone the driver has visited.
We could apply the .count() method to our groupby(), but this would also count the values in the `zone` column, and we would lose the information of the zones visited by the drivers. To avoid this, we instead use the .apply() method to apply to these groups a custom function `nb_zone` that returns a new column `count` containing in every row the same value, the number of zones visited by the driver in the group we consider. This allows us better control on which column is aggregated, dropped, or kept intact. The value of the count for each driver is repeated on every row of its group, but this feature is now welcome, as we next group by `zone`. We obtain groups for each zone identifier containing all different values of `count` we previously computed for each unique driver. The mean() aggregation method is applied to these groups, and we obtain a dataframe with the zone identifier as index, and a single column `count` corresponding to the average number of zones visited by drivers who went through the zone defined by the index, column on which we sort our rows.

```
def nb_zone(x):
    x['count'] = len(x)
    return x
gps_meanvisited = gpsclean[['driver','zone']].drop_duplicates().groupby('
    driver').apply(nb_zone).groupby('zone').mean().sort_values('count',
    ascending=False)
```

- **Third analysis**:

```
def nb_zone2(x):
    x = x.sort_values('timestamp', ascending=False)
    x['count'] = x.zone.diff().ne(0).cumsum().max()
    return x
gps_meanexchange = gpsclean[['driver','zone','timestamp']].groupby('driver').
    apply(nb_zone2)
gps_meanexchange = gps_meanexchange.drop_duplicates(subset=['driver','zone']).
    groupby('zone').mean().sort_values('count', ascending=False)
```

We will now try to take into account the possible returns of drivers into zones. Indeed, it is possible that in a certain small and close by subset of zones, we move around a lot but we never go very far.

This time, we cannot start with a drop_duplicates(). We will have to start with a groupby apply on the drivers and order them according to the time, then count the number of changes of zone to finish by assigning as before a new column with this data. The number of zone changes is counted using diff().ne(0).cumsum().max(). After that we will apply a drop duplicate to not distort the groupby.mean() on zone.

4.2 Comment on the parallelization in Dask

To comment on the parallelization, we go from the simplest graph to the most complicated.

- **Second analysis**:

We can see in Fig. 3 that Dask starts with multiprocessing and goes to multithreading at the end. As before, before the drop_duplicates(), the amount of data that Dask has to handle is large enough that it feels it is worth the effort to have several workers working at the same time at first, but after a drop_duplicates(), it is a better tradeoff to only use one worker. However, we observe that towards the end, the single worker left splits into two parallel tasks: to calculate the mean, we need to compute separately the sum of our elements and a count of our elements.

- **Third analysis**:

In the third analysis, shown in Fig. 4, we see again that the processes stay parallelized between our workers as long as no aggregation function is used. When the first groupby() is applied, this leads to a widespread shuffle of the data between all the workers.
After the drop_duplicates() is applied, the same behaviour as previously is shown, where the aggregation leads to the data being regrouped on a single worker. This single worker performs the second groupby() alone, and since we applied again a mean() aggregation method, it splits into two parallel threads to compute the mean through the sum and the count.

- **First analysis**:

The graph of the first analysis, shown in Fig. 5, is identical (up to speed_compute() function) to that of the third analysis up to nbzone2, becoming here speed_compute since they are both just applying a custom function to a groupby(). After that, in the second analysis, the first groupby() was followed by a drop_duplicates(), which forced the data to be aggregated to a single worker. This is not the case here, and the processes stay parallelized. Each worker applying the speed_compute function to its subset of the data also splits into two threads to compute the sum and count of this subset. It is only then that all of the partial sums and counts are aggregated to compute the total mean.
Since no aggregation was needed, it was in this case visibly more efficient to stay parallelized, as opposed to the previous case where an aggregation was needed, so that resplitting the data into workers was less interesting than leaving it to a single worker. Additionally, there was no drop_duplicates() in this case, so

we had significantly more data to process, adding to the attractivity of keeping things parallelized.

- **Total computation**:

One of the recommended practices in Dask is to be careful about the way we compute our delayed structures. Indeed, if the different dataframes and operations we perform share common dependencies, running them separately will lead to repeating the same process multiple times inefficiently. Instead, it is better practice to compute everything with shared dependencies at the same time, in a same dask.compute() instance.
In our case, all of the analyses we have performed share at least one common dependency: the computation of the initial zones of each driver data point. For this reason, it is actually more interesting to run all 4 algorithms at the same time:

```
dask.compute(gps_mostvisited, gps_speeds, gps_meanvisited, gps_meanexchange)
```

The graph of this common computation is shown in Fig. 6. It is quite complex, but we can notice that it corresponds to the graphs of each of the four quantities mentioned, stitched together. The start is common, as it corresponds to the computation of the initial zones of each driver. It starts to split at the level of the "drop-duplicates-chunk" and "gt" tasks, where the `gps_mostvisited` is not cleaned and undergoes immediately a drop_duplicates(), while all the other computations first go through a filtering, verifying that we keep only values of `zone` > 0.
As the computation of the zones of the drivers is actually the longest part of our project, sharing this step instead of repeating this each time can have a dramatic impact on the duration of our code.

This graph gives us some leads of possible optimizations of the parallelization of our code. A very interesting detail is that the 4 calculations all end up on the same worker (0) after being aggregated. We assume that when is not necessary to use multiple workers, Dask assigns the task to the worker (0) by default. Since other workers were involved in the processes before the aggregation, it should not cost anything to assign the end of each of the calculation to other workers. We assume then that we could improve the parallelization of this code by performing in this way, assigning the end of each separate calculation post-aggregation to distinct workers.
We could also probably have improved the parallelization further by fine-tuning the dask.optimization toolbox settings. This toolbox allows, for example, to automatically detect and cull unnecessary steps or processes. Most of these optimization tools are already applied by default when using Dask structures and functions.
Another specificity is that grouping by a column that is not an index is also quite expensive as it requires shuffling the entire data. Grouping on an index is much more efficient. However, setting an index from a column also requires shuffling the data. It still might be interesting to perform this index setting once at the beginning of a computation, as in our case, since we end up grouping by the same columns (`driver` and `zone`) almost all the time.

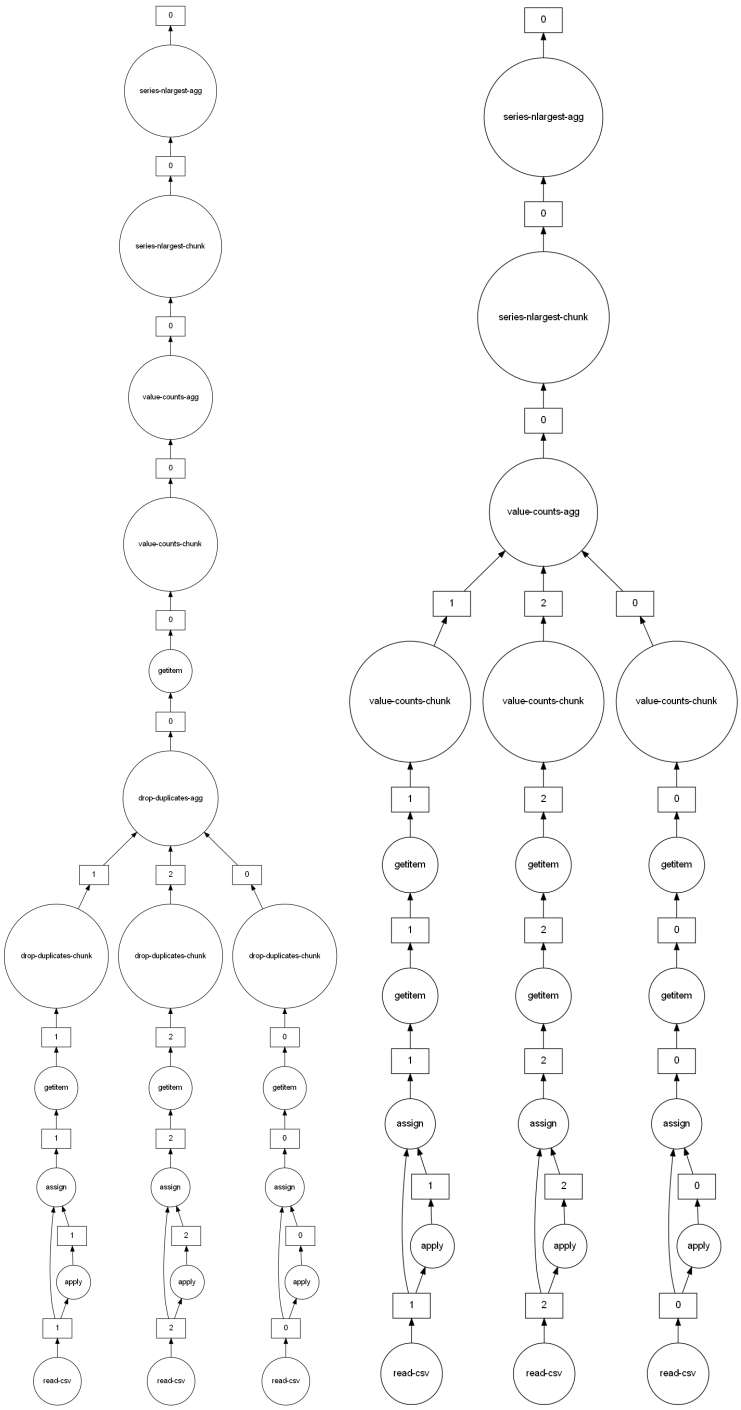

Figure 2: Left: Parallelization chart of the code for a count of unique drivers, with .drop_duplicates(). Right: Parallelization chart of the code without .drop_duplicates()

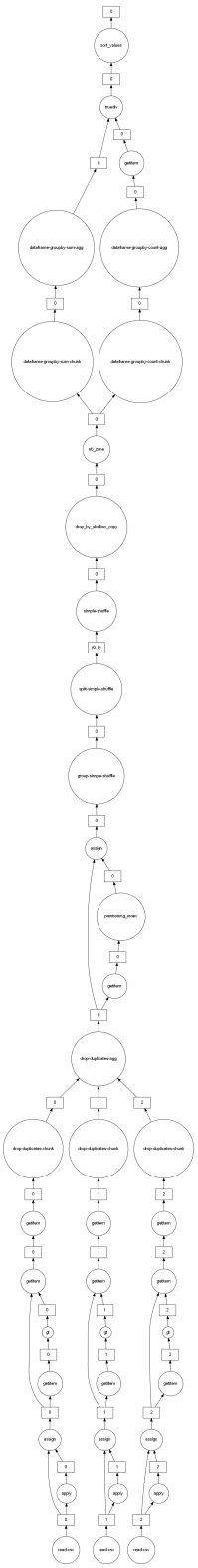

Figure 3: Parallelization chart of the computation of the average amount of zones visited by the drivers having visited a specific zone.

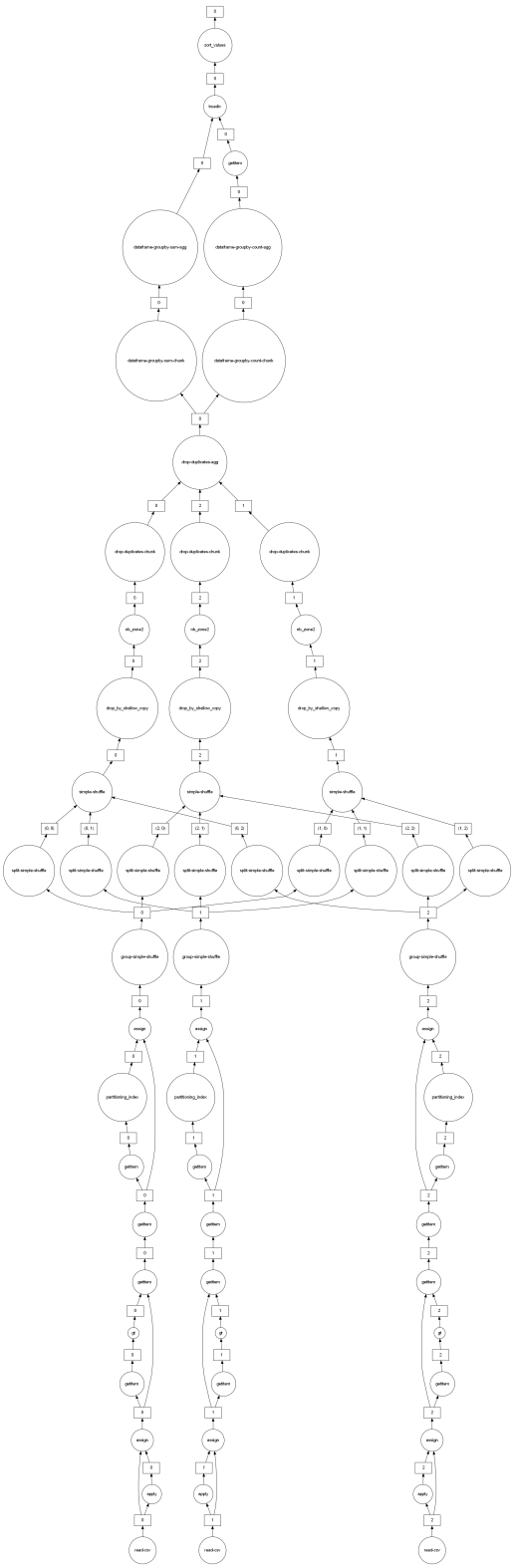

Figure 4: Parallelization chart of the computation of the average amount of times the drivers having visited a specific zone will change zones.

Figure 5: Parallelization chart of the computation of the average total distance traveled, average total time spent, and average speed for each zone.

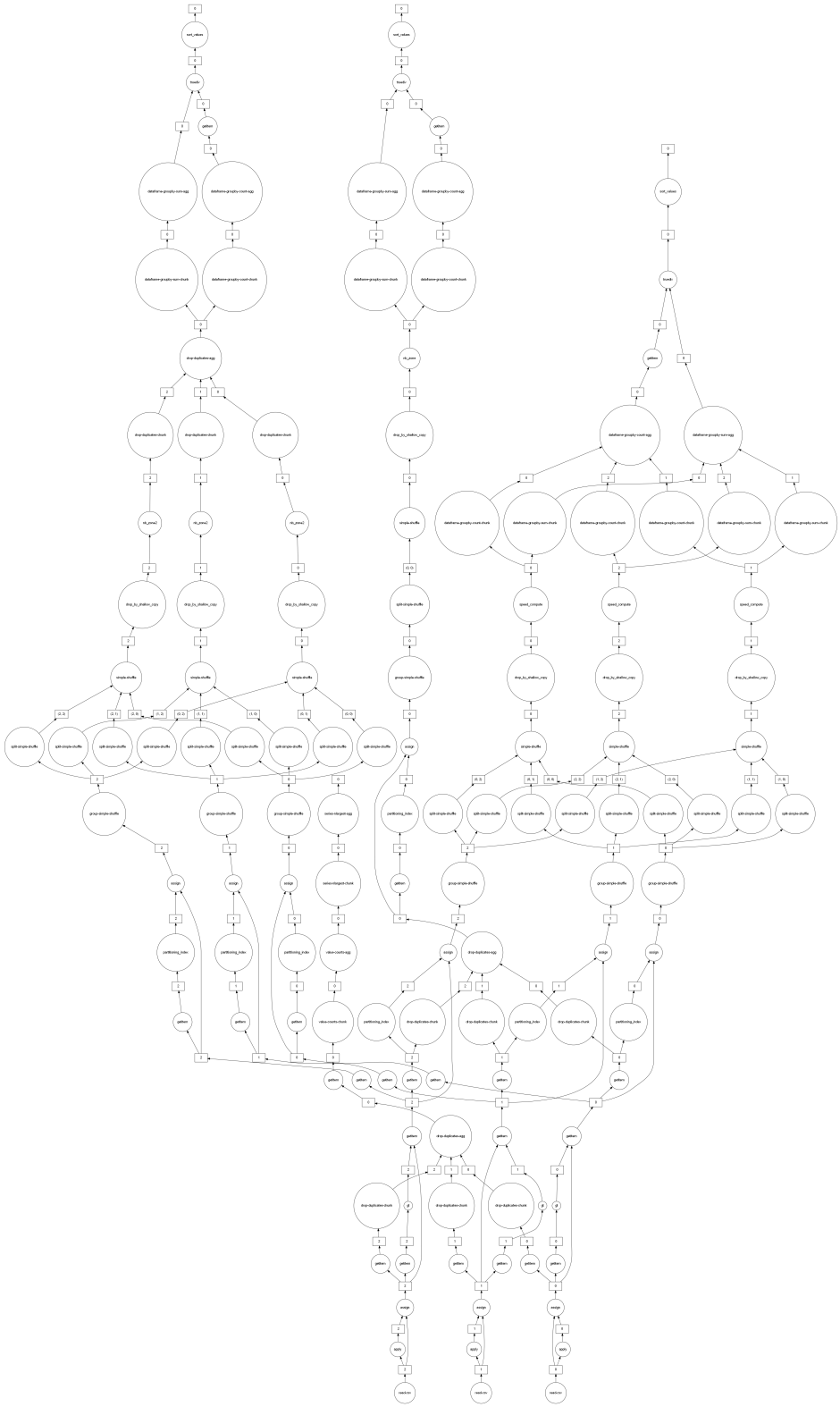

Figure 6: Parallelization chart of all the above-mentioned analyses, done together in a single dask.compute() instance.

4.3 PySpark

- **First analysis**:

Likewise, since the features that we want to address here are dynamical features, we need to find a way to compute differences between rows even if there is no notion of row index in PySpark. Fortunately, there exists a `pyspark.sql` function called `lead`. Its working principle is the following : first, one has to specify the offset parameter, say for instance offset = j where j is an integer number, then the `lead` applied to the n^{th} row will have access to the $(n+j)^{\text{th}}$ row. Practically, we will set the offset to 1, since we want to compute the "instantaneous" speeds for each change of row. In order to work properly, `lead` needs a dataset which is partitioned and ordered. More specifically, `lead` is a `Window` function and it mandatory to specify how to partition and order the partitioned data :

```
windowSpec = Window.partitionBy('driver').orderBy('timestamp')
```

In this way, `lead` will see each driver subset separately, and inside each driver subset, the datapoints are ordered by time. In particular, it will return a null value for the last row of a driver subset, which ensures that we are not mixing the drivers data.

To compute the distance between two geographical points, we use `distance.geodesic` in the package `geopy`. This is more precise than the Haversine formula, since the earth is not assumed to be a sphere here.

```
def get_distance(lat1,long1,lat2,long2):
    return geopy.distance.geodesic((lat1,long1),(lat2,long2)).m
```

This function will help us generate a Δx column, with meters as units. For the time differences, we need to extract seconds out of timestamps

```
def get_time(t1,t2):
    if (t1 == None) or (t2 == None) :
        return None
    else:
        return (t2-t1).seconds
```

This function will help us generate a Δt column, with seconds as units. Note that some precautions are taken to prevent the `.seconds` function to return an error if one of the two inputs is a null-valued timestamp. This may happen at the last row of a driver's partition. Of course, both these functions have to be converted into `udf`'s. Now we are ready to generate a dataframe containing all the distance and time increments, as well as the instantenous speeds.

```
dynamics = driversZ.withColumn('next_lat', lead('latitude', offset=1).over(
    windowSpec)) \
    .withColumn('next_lng', lead('longitude', offset=1).over(windowSpec)) \
    .withColumn('next_time', lead('timestamp', offset=1).over(windowSpec)) \
    .withColumn('deltaT', get_time_udf('timestamp', 'next_time') ) \
    .withColumn('deltaX', get_distance_udf('latitude', 'longitude','next_lat',
        'next_lng') ) \
    .withColumn('speed', col('deltaX')/col('deltaT')).na.drop()
```

Due to GPS jittering, we will get rid of unphysical values, as explained before in Sec.4.1. Since our units are meters and seconds, the cut-off that we chose are 56 m/s for the maximal instantaneous speeds and 1200s for the maximal Δt. The obtained dataframe is called `dynamicsFinal`. Note that we could have used some statistical techniques to exclude outliers but since the data has a simple physical interpretation, we decided to put the cut-off by hand.

The first things that we want to compute are the average travel time and average travel distance for each zone. The idea is to sum all the Δt of a same driver, in all the zones he passed through. And similarly for the Δx. Practically, we use a `groupBy(['driver','zone'])` to get all the unique pairs (driver,zone) and specify sums over the Δt and Δx as the way of aggregating. We cache this result, too.

```
dzoneDistTime = dynamicsFinal.groupBy(['driver','zone']).agg( sum('deltaX').
    alias('dtot') , sum('deltaT').alias('ttot')).cache()
```

The only operations remaining are to group by zone (while summing again over the distances and times previously computed) and then to divide by the number of drivers in each zone (we computed and cached this in Question 1). Let us also take care of converting the results into kilometers and minuts, for the sake of clarity.

```
averageQuantities = dzoneDistTime.groupBy('zone').agg(sum('ttot').alias("
    total_time"), sum('dtot').alias('total_dist')) \
     .join(driversByZone, 'zone') \
     .withColumn('avg_time/driver_(min)', col('total_time')/(60*col('count'))
         \
     .withColumn('avg_dist/driver_(km)', col('total_dist')/(1000*col('count')))
```

The average speeds can be easily computed out of the `dzoneDistTime` dataframe that we cached just above. We just have to divide the dtot column (= total distance of a driver in a zone) by the ttot column (= total time of this driver in this zone) to get his average speed in this zone. Then, we group by zone while averaging all the speeds of the drivers. Let us also convert them into km/h.

```
averageSpeeds = dzoneDistTime.withColumn('dzoneSpeed', col('dtot')*3.6/col('
    ttot')) \
              .groupBy('zone').agg( mean('dzoneSpeed').alias('avg_speed_(km/h)')
                  )
```

- **Second analysis:**

As for the Dask part, we want to know how many zones each driver has visited. Thus, we take all the unique pairs (driver,zone) and we group them by driver, while counting the different zones through which he passed. Now we go back to the (driver,zone) pairs and add next to each driver the number of zones he visited. All the zones that a single driver has seen will of course share the same number for this new column, as we want. Eventually, we group by zone while averaging the numbers of visited zones

```
pairsDriverZone = driversZ.dropDuplicates(['driver','zone']).select(['driver',
    'zone'])
HowManyZonesByDriver = pairsDriverZone.cache().groupBy('driver').count()
pairsDriverZone.join(HowManyZonesByDriver, 'driver') \
    .groupBy('zone').agg( mean('count').alias('NumberOfZones'))
```

- **Third analysis:**

Here we want the average number of exchanges so that we have to keep track of people coming back in the same zones. We use the `lead` function (as for the speeds) in order to spot the rows where there is a change of zone. Again, we use the previous `windowSpec` which partitions the dataset by driver and orders every partitioned subset by time. By computing in the zone column the normalized difference between a row and the next one, a 1.0 will betray the presence of a zone change while a null value will mean that the driver stayed in the same zone. Note that these null values come from the fact that we divided the difference between the two rows by itself, since Spark gives a null as an output for a division by 0. Also, summing these 1.0 will give us the number of zone changes, but we will have to add 1 to the total in order to get the true total of visited zones, repetitions included. Once we have this, the reasoning is the same than for the unique visited zones just above.

```
driversZd = driversZ.select(['driver','timestamp','zone']).withColumn('
    next_zone', lead('zone', offset=1).over(windowSpec)).na.drop() \
```

```
        .withColumn('diffzone', (col('next_zone') - col('zone'))/(col('
            next_zone') - col('zone')))

HowManyZonesExchByDriver = driversZd.groupBy('driver').agg(sum('diffzone').
    alias('SumDiffZone')) \
        .withColumn('SumDiffZone+1', col('SumDiffZone')+1)
pairsDriverZone.join(HowManyZonesExchByDriver, 'driver') \
    .groupBy('zone').agg(mean('SumDiffZone+1').alias('NumberOfZonesExch'))
```

5 Results and timing of the computations for Dask and PySpark

In this section, we show the results of the computations and discuss them, as well as time performances.

5.1 First question

The first question is quite straightforward by definition and gives exactly the same results via Dask or PySpark, as we can see in Tab.1. A slight change is that in Dask we chose to discard the null zone after this computation, so that it appears in the left table. We displayed however the 11 first zones. Even if it is not shown here, only 29 out of the 51 zones contain GPS recordings. The deeper we go in this ranking, the less people there are. For instance the last one, i.e. zone 7, is such that only one driver passed through it. Needless to say that the statistics that we perform for the second question are not very meaningful for the zones that have very few drivers, due to the lack of data.

Due to the high number of drivers in the zones 21 and 22, we expect them to be located near the center of the city, since it is densely populated and there is a lot of traffic. This is confirmed by Fig.1: the further down one goes in the table, the further away from the city center the zone is.

5.2 Second question

First, let us focus on the dynamical features of Tables 2 and 3. The average travel time, travel distance and speed by zone are rather similar for Dask and PySpark even if there are some differences. The trends are the same: there are a lot of traffic jams in the city center. Indeed, the average speeds in the zones 21 and 22 are quite slow, while the average time spent there is quite big. However the average distance is quite big too, one can suppose that the density of streets is high. The zone 2 is also interesting. We see that the average speed is way bigger than for the other zones. One could guess that there is a highway road there, with not many other smaller roads. This is confirmed by Fig.1.

Let us not forget that the sample can be quite small for some zones. For instance, the results for the zone 7 do not represent any real feature of the zone, since we have the data only for a single driver there.

The differences between Dask and PySpark results arise from a combination of factors. Since the framework is not the same, the functions that we defined are not exactly the same and can work differently. For instance, in Dask we computed differences with respect to the previous row, while we did it with respect to the next row in PySpark. Hence, the delays, distances and the speeds may be different near the edges of the drivers subsets. Accordingly, the rows that we drop out because of null values are not the same. Another cause of divergence is that we used in the PySpark version a more precise method in order to compute distances on the surface of the Earth. These errors due to the spherical approximation can propagate and get amplified in the values computed afterwards from these distances.

Finally, let us dig into the connections between the zones with Table 4. The zones that are the most connected to others can be interpreted as following : they are zones that are far away from the city center, so that drivers starting from these zones have to pass by a lot a other zones in order to go to a point of interest (for instance, the city center). And conversely for the drivers who want to go to these remote areas. Take for instance the zones 12, 8, 35 and 49. We see on the map of Fig.1 that they are all away from the city center and they are crossed by major road such as highways or national roads. The 21 and 22 city center zones are

quite low in the ranking, so we can guess that drivers there don't visit other zones and mainly remain in the city center. Another nice thing to spot is that the zone 2 is high in the number of visited zones but rather low in the total zone exchanges. It agrees with the fact that people driving through zone 2 are mostly using the highway, hence are not coming back in passed zones during the trip.

There are quite a few possible other analyses that we could have performed with this data. The most interesting example to us is that we could have performed all of the previous analyses by separating the data into different hours, or periods of the day. This could have allowed to identify periods of high traffic, such as the morning rush hour, or the usual time when workers go back home. This could also allow the show which zones should be avoided depending on the time of the day.

5.3 Timing

In Dask, we can first try to compute all the data separately, in an unefficient way as explained before. As shown in the group3_vdask_separate.pdf file attached, each task takes at least 7 minutes: gps_mostvisited, gps_speeds, gps_meanvisited and gps_meanexchange take respectively 9 min 17 s, 9 min 22 s, 7 min 38 s and 7 min 34 s to compute. This is because, each time, the zones in the GPS data are recomputed and re-assigned. The best practice is to proceed as shown in group3_vdask_total.pdf, which shows that the computation of all the previously cited dataframes now only amounts to 9 min 53 s. It becomes clear now that much of the 7 minutes (the minimum amount of time found in the previous computation) corresponds to the computation of the zones, while our actual analysis takes no more than 3 minutes. All things parallelized, all of these analyses can be performed in the time that it takes to peform only one of them.

In PySpark, as we explained previously, the computations take a lot of time if we work on the initial driver dataframe in which the zone column is created via an udf function. For instance, the first question takes 35min to run. We chose to allocate 30min to the writing of a new .csv file in order to go faster subsequently. The first question now takes 11s to run. The first part of the dynamics computation in the second question take 7 min 42s to run, while the second part (i.e. the average speed) takes 1.54s, since we cached the required dataframe previously computed. The dataframes about the visits and the exchanges are pretty fast to compute; both 13s. As we can see, PySpark is really fast to compute things via functions that belong to its library, since they are really optimized. All in all, one should be aware that PySpark is doing quite impressively when the size of the dataset begins to become very important. The present work cannot really show the full potential of PySpark due to the small size of the drivers dataset.

zone	
22	10823
21	8039
14	4150
25	3638
13	3178
20	1591
23	1160
24	1064
26	747
0	547
15	512

zone	count
22	10823
21	8039
14	4150
25	3638
13	3178
20	1591
23	1160
24	1064
26	747
15	512

Table 1: Left: Results for the `gps_mostvisited` dataframe in Dask. Right: Results for `driversByZone` in PySpark

dzone	total_time	total_distance	meanspeed
2	206.000000	2.943416	68.734520
11	1751.166667	8.410023	59.956685
8	2268.333333	8.147536	30.439522
30	1020.500000	3.396694	23.760043
27	2022.272727	5.830054	23.199141
29	792.115385	2.798897	20.818950
20	2146.177876	4.317292	19.731458
15	1447.841797	3.550905	19.554468
40	754.352941	3.852713	18.390652
28	1909.116667	4.832468	17.858329
16	2346.327869	6.145424	17.818974
17	1302.421053	1.846554	17.480475
37	1503.102041	4.715008	17.353181
19	1743.627049	4.699589	16.305877
12	810.392857	1.702010	15.398160
26	2527.880857	5.547971	15.283745
49	862.666667	2.005988	15.079609
18	1737.784615	3.758791	15.069416
24	1829.854323	2.914854	13.716427
14	3748.071807	7.771217	13.170996
13	2549.153870	4.797012	13.117010
23	2148.012931	4.774195	13.107724
38	1177.000000	3.126494	12.342012
21	5672.036696	12.555548	10.939228
35	428.100000	1.566745	10.811023
22	7234.611106	17.017696	9.976861
25	3147.868059	7.574746	9.528753
31	1195.474359	1.284915	8.792135
7	350.000000	0.093051	0.957094

Table 2: Results for the `gps_speeds` dataframe in Dask.

zone	avg time/driver (min)	avg dist/driver (km)
22	75.92330530660014	14.95825744395522
21	52.470497988970436	11.178773470081994
14	29.755004016064255	6.89719232403659
27	29.624458874458874	6.292309179868985
11	29.066666666666666	8.745267377084742
16	28.298816029143897	6.543657695698078
25	26.751736301997436	6.696107607515276
26	26.23119143239625	5.479586937049127
8	26.0625	8.501636286875232
28	23.62638888888889	5.159186138508842
19	22.100751366120218	4.951896950200414
38	20.24848484848485	3.4727615119801327
23	19.50846264367816	4.777424736315237
20	19.442698512465956	4.1675496402618215
13	19.344435703796936	4.616003906233819
24	18.449733709273183	2.922989659491128
18	18.0785641025641	3.958002401680419
37	17.429931972789117	4.8839770184007225
15	13.444889322916667	3.6580007328134525
30	12.61111111111111	3.596926790828506
40	12.326470588235294	4.128118237722008
31	12.17542735042735	1.4988695824008722
17	11.75906432748538	2.083281552488605
29	11.335576923076923	2.8555806200687703
49	8.684722222222222	2.6786293389927596
12	6.7994047619047615	2.118051070667803
35	6.595	1.4005737886466085
7	5.833333333333333	0.09301459789276123
2	4.294444444444444	3.904337541739146

zone	avg speed
2	70.19798352784623
11	59.53952607743003
8	36.896887496332376
30	26.484407724141303
12	25.27155985417999
15	25.17200485499606
27	24.636710630752123
29	22.82129088318582
20	22.10889838694594
16	20.99577766424281
40	20.345577486434095
28	20.187964746304303
37	19.74989393166856
17	18.008349792442996
25	17.698093068482624
49	17.51972791634245
19	17.134066348741296
13	16.805070382999883
18	16.777034642167383
24	16.686885571654198
14	15.711401011118559
26	15.65981682608462
23	15.566055160976825
21	14.067287275992708
38	13.140226659040211
22	12.429955215420618
35	11.379321170849513
31	11.10770715159875
7	0.956721578325544

Table 3: Left: Results for the `averageQuantities` dataframe in PySpark. Right: Results for the `averageSpeeds` dataframe in PySpark.

zone	count
12	6.250000
8	6.166667
35	5.300000
49	5.250000
11	5.000000
40	4.705882
30	4.333333
2	4.333333
29	4.288462
16	4.251366
15	4.082031
27	3.987013
28	3.966667
37	3.959184
17	3.929825
24	3.909774
19	3.795082
13	3.717432
23	3.603448
20	3.539283
14	3.399277
25	3.282573
18	3.236923
31	3.217949
21	3.055853
26	2.903614
22	2.859004
38	2.363636
7	2.000000

zone	count
12	9.750000
8	9.500000
11	8.333333
30	7.666667
13	7.126809
15	6.933594
35	6.800000
49	6.750000
17	6.649123
16	6.546448
24	6.412594
23	6.230172
14	6.185301
40	6.058824
29	5.903846
20	5.717788
27	5.441558
19	5.413934
21	5.380893
18	5.212308
25	5.205058
28	5.066667
37	5.061224
22	4.855216
31	4.525641
2	4.333333
26	4.171352
38	2.727273
7	2.000000

zone	NumberOfZones
12	6.25
8	6.166666666666667
35	5.3
49	5.25
11	5.0
40	4.705882352941177
30	4.333333333333333
2	4.333333333333333
29	4.288461538461538
16	4.251366120218579
15	4.08203125
27	3.987012987012987
28	3.966666666666667
37	3.9591836734693877
17	3.9298245614035086
24	3.9097744360902253
19	3.7950819672131146
13	3.7174323473882946
23	3.603448275862069
20	3.5392834695160276
14	3.399277108433735
25	3.2825728422210005
18	3.236923076923077
31	3.217948717948718
21	3.055852717999751
26	2.9036144578313254
22	2.8590039730204193
38	2.3636363636363638
7	2.0

zone	NumberOfZonesExch
12	10.074074074074074
8	9.5
11	8.333333333333334
35	7.444444444444445
30	7.333333333333333
49	7.2727272727272725
15	7.246913580246914
13	7.241500962155228
16	7.226993865030675
17	7.0754716981132075
24	6.805443548387097
40	6.733333333333333
14	6.712426978226235
23	6.4586330935251794
29	6.3125
27	6.2615384615384615
20	6.237264480111654
28	6.083333333333333
19	5.873303167420814
37	5.738095238095238
21	5.724942975982826
25	5.67685723020483
18	5.473856209150327
22	5.45071474290591
31	5.166666666666667
26	4.949832775919733
2	4.333333333333333
38	3.375
7	2.0

Table 4: Left: Results for the `gps_meanvisited` dataframe in and for the `gps_meanexchange` dataframe (in order) in Dask. Right: Results for the number of visited zones (NumberOfZones) and the number of zone exchanges (NumberOfZonesExch) in PySpark.