Streams Processing - Project 1
Taxi Trips from New York City

Introduction
In this project, we processed a dataset from ACM DEBS 2015 Grand Challenge, which contains information
about taxi trips in New York City. Each record in the dataset corresponds to one taxi ride and contains multiple variables, such as the taxi identifier, the date and time of the pickup and the dropoff of the ride, the duration and distance of the ride, the latitude and longitude of the pickup location and dropoff location as well as information related to payment. Thus, using this information, we tried to answer
several questions that might be useful to the taxi drivers, namely when choosing the areas where they should wait to pickup clients.
To do that, we used the base Spark Streaming system for some queries, and the Structured Spark Streaming framework for the others. We started by processing the dataset in order to exclude locations outside the stipulated bounds and null values that might affect computations and to convert longitude and latitude values to cell grids. Afterwards, we used the processed dataset to answer the questions. All the computations performed throughout the project are explained in the next sections. In the end of each query, we present an example of the results, obtained using the publisher publisher-debs.sh.

Util Functions
In order to keep a clean and simple code as well as a bigger cohesion inside the project itself, we created some functions that abstract basic operations such as filters, conversions, etc. This is done for both structured and non structured technologies.
First we will address the non structured streaming. The first function has three different purposes: to filter empty lines, to filter locations that are outside of the target city, making use of some specific boundaries, and to filter some values that are less or equal to zero in variables where it does not make sense. The boundaries are also defined right before the initialization of the function itself. These are outside of the function so they can be accessed by other functions as well. This function is meant to be used inside a filter RDD function, so it outputs a boolean value stating if that row is a valid one.

```
================================================================================
# Longitude and latitude from the upper left corner of the grid, to help conversion
init_long = -74.916578
init_lat = 41.47718278
# Longitude and latitude from the lower right boundaries for filtering purposes
limit_long = -73.120778
limit_lat = 40.12971598
```

1

```
def apply_filters(line):
# Split the line for a ,
splitted_line = line.split(',')
# Return boolean
return (
(len(line) > 0) and \
(float(splitted_line[6]) > init_long) and \
(float(splitted_line[6]) < limit_long) and \
(float(splitted_line[7]) > limit_lat) and \
(float(splitted_line[7]) < init_lat) and \
(float(splitted_line[8]) > init_long) and \
(float(splitted_line[8]) < limit_long) and \
(float(splitted_line[9]) > limit_lat) and \
```

```
(float(splitted_line[9]) < init_lat) and \
(float(splitted_line[5]) > 0) and \
(float(splitted_line[4]) > 0) and \
(float(splitted_line[16]) > 0)
)
```

==========

The second function has the goal of returning the same rows with the additional information of the cell ids (these are based on the locations), and has two modes with different sizes to be returned. This allows to create a cell grid of 300 x 300 (where each cell is a square of 500m by 500m), as well as a cell grid of 600 x 600. Just like the previous one, this function is meant to be used inside a RDD function, this time the map.

==========

```
def get_areas(line, _type = "bigger"):
# Split the line for a ,
splitted_line = line.split(',')
line = splitted_line
# Longitude and latitude that correspond to a shift in 500 meters
long_shift = 0.005986
lat_shift = 0.004491556
# Longitude and latitude that correspond to a shift in 250 meters
if _type == "smaller":
long_shift = long_shift / 2
lat_shift = lat_shift / 2
return (
line[0], line[1], line[2], line[3], line[4], line[5], line[6],
line[7], line[8], line[9], line[10], line[11], line[12], line[13], line[14], line[15], line -
[16],
str(math.ceil((float(line[6])-init_long)/long_shift)) + "-" + str(math.ceil((init_lat-float -
(line[7]))/lat_shift)),
str(math.ceil((float(line[8])-init_long)/long_shift)) + "-" + str(math.ceil((init_lat-float -
(line[9]))/lat_shift))
)
```

==========

The next functions will address the structured streaming technology. Note that the functions below receive whole dataframes, instead of one row as was described in the previous ones, so that the objective computation of the function is encapsulated inside itself only. This could not be done with the non structured functions due to context problems.

This function is yet another that filters the locations and the same values as the first shown above. We joined these goals into one function so that the split of the values do not occur many times.

==========

```
def filter_locations_and_integers(lines):
split_lines = split(lines["value"], ",")
lines = lines.filter(split_lines.getItem(6) < limit_long) \
.filter(split_lines.getItem(6) > init_long) \
.filter(split_lines.getItem(7) < init_lat) \
.filter(split_lines.getItem(7) > limit_lat) \
.filter(split_lines.getItem(8) < limit_long) \
.filter(split_lines.getItem(8) > init_long) \
.filter(split_lines.getItem(9) < init_lat) \
.filter(split_lines.getItem(9) > limit_lat) \
.filter(split_lines.getItem(5) > 0) \
.filter(split_lines.getItem(4) > 0) \
.filter(split_lines.getItem(16) > 0)
```

```
return lines
```
==========

Next, we defined a function that gets the areas for the rows in the dataframe structure. Thus, we get the dataframe and the mode with which we would like to create the cell map. We compute the cells for the longitudes and latitudes and then we concatenate them.
==========

```
def get_areas_df(lines, _type = "bigger"):
# Longitude and latitude that correspond to a shift in 500 meters
long_shift = 0.005986
lat_shift = 0.004491556
# Longitude and latitude that correspond to a shift in 250 meters
if _type == "smaller":
long_shift = long_shift / 2
lat_shift = lat_shift / 2
split_lines = split(lines["value"], ",")
lines = lines \
.withColumn("cell_pickup_longitude", ceil((split_lines.getItem(6).cast("double") - init_long) / -
long_shift)) \
.withColumn("cell_pickup_latitude", -ceil((split_lines.getItem(7).cast("double") - init_lat) /  -
lat_shift)) \
.withColumn("cell_dropoff_longitude", ceil((split_lines.getItem(8).cast("double") - init_long)  -
/ long_shift)) \
.withColumn("cell_dropoff_latitude", -ceil((split_lines.getItem(9).cast("double") - init_lat) / -
lat_shift))
lines = lines \
.withColumn("cell_pickup", concat_ws("-", lines["cell_pickup_latitude"], lines[" -
cell_pickup_longitude"])) \
.withColumn("cell_dropoff", concat_ws("-", lines["cell_dropoff_latitude"], lines[" -
cell_dropoff_longitude"])) \
.drop("cell_pickup_latitude", "cell_pickup_longitude", "cell_dropoff_latitude", " -
cell_dropoff_longitude")
return lines
```
==========

Some queries require the areas but others need the routes, i.e. the concatenation of pickup cell and the drop off one. This is exactly what is done in the next function.
==========

```
def get_routes(lines):
lines = lines.withColumn("route", concat_ws("/", lines["cell_pickup"], lines["cell_dropoff"]))
return lines
```
==========

An abstraction covering some functions already specified here was created for every computation that will be used inside all queries. We start by removing empty lines inside the dataframe and then we call the filtering function, leaving us with all valid rows for every solution we are attempting to reach. This does not include the creation of areas and routes because that is not needed in every query.
==========

```
# Function that does basic pre processing on the data for the dataframes examples
def pre_process_df(lines):
# Filter empty rows
lines = lines.na.drop(how="all")
# Filter locations outside current range or bad inputs. Also filter distance time,
# time and total amounts that are less than 0.
lines = filter_locations_and_integers(lines)
return lines
```

==========

We created a dynamic way of getting only the desired columns for every case. This function receives the dataframe and then the columns that we wish to retain from the data. This is done by receiving an array of tuples that has the name of the column in the first position and then the data type in the second one. We define an array with all the column names inside the function itself, and then, with a cycle, we attach the columns received in the function argument by getting the index in the name vector that corresponds to the target column name.

==========

```
# Get only the specified columns
# Columns is an array with tuples inside, the tuples got the column name and the respective type
def get_columns(lines, columns):
# Array with columns for index
whole_columns = ["medallion","hack_license","pickup_datetime", "dropoff_datetime", "trip_time",
"trip_distance", "pickup_longitude", "pickup_latitude", "dropoff_longitude", " -
dropoff_latitude",
"payment_type", "fare_amount", "surcharge", "mta_tax", "tip_amount", " -
tolls_amount", "total_amount"]
split_lines = split(lines["value"], ",")
for column in columns:
lines = lines.withColumn(column[0], split_lines.getItem(whole_columns.index(column[0])). -
cast(column[1]))
return lines
```

==========

Query 1 - Most Frequent Routes

In this query, the goal was to find the top 10 most frequent routes during the last 30 minutes, using a 300 x 300 cell grid. To do that, we used Structured Spark Streaming.
We started by creating a sliding window based on the dropoff datetime with a duration of 30 minutes and a slide of 10 minutes, to obtain the routes for the last 30 minutes. We grouped the results by that window and by the route (obtained using the get_routes() function) and counted the records. After that, we ordered the results by the window and the counts in descending order, to obtain the top counts
for each window, and limited the results to show only the first 10 routes.

Note that we also defined a watermark of 30 minutes, to deal with late data. This watermark excludes data that arrives later than the threshold of 30 minutes.

==========

```
# Count the occurrences for each route
most_frequent_routes = lines.withWatermark("dropoff_datetime", "30 minutes") \
.groupBy(window(lines.dropoff_datetime, "30 minutes", "10 minutes")," -
route") \
.count() \
.orderBy("window","count",ascending=False) \
.limit(10)
```

==========

Below, we show an example of the results, that allow taxi drivers to know which areas and routes have a highest number of clients.

Query 2 - Most Profitable Areas

In this query we were asked to find the most profitable areas for taxi drivers. We defined the profitability
of an area as the ratio between the profit generated in the area within the last 15 minutes and the number
of dropoffs in this same area. The profit generated in an area is computed by calculating the average fare and tip for trips that started in the area (that is, have this area indicated as the pickup area) and ended within the last 15 minutes. We decided to divide this average profit by the number of dropoffs in the last 30 minutes, so that we penalize areas where there are a lot of dropoffs, as this will lead to more competition between taxi drivers and it is consequently harder for a driver to get a client.
For this problem we used a cell size of 250 m x 250 m, i.e., we are representing New York in a 600 x 600 grid.
To answer this query we used Spark Structured Streaming.
We started by selecting the relevant variables we would be using, namely dropoff datetime, pickup datetime, cell dropoff, cell pickup, fare amount and tip amount. Then we created a new column that calculates the profit, by adding the tip amount to the fare amount.

==========

```
lines = lines.withColumn("profit", lines["fare_amount"] + lines["tip_amount"]) \
.drop("value")
```

==========

Based on this dataframe we created two new ones:

• Profit_average: For computing the average profit per pickup area in a 15 minute window. To do this we grouped by the cell pickup area and then by the 15 minute window. We then aggregated the rows to compute the average profit for these groups.
• Taxis_total: For computing the total number of dropoffs per area in a 30 minute window. To do this we grouped by the cell dropoff area and then by the 30 minute window. We then aggregated the rows by simply counting them.

==========

```
profit_average = lines.groupBy(lines.cell_pickup,window(lines.dropoff_datetime,"15 minutes"))\
.agg(avg(lines.profit).alias("avg_profit"))
taxis_total = lines.groupBy(lines.cell_dropoff,window(lines.dropoff_datetime,"30 minutes"))\
.count()
```

After creating these two dataframes, we output their result to memory in order to define a new computation on this data.

```
query = profit_average \
.writeStream \
.queryName("profit") \
.outputMode("complete") \
.format("memory")\
.start()
query2 = taxis_total \
.writeStream \
.queryName("taxis") \
.outputMode("complete") \
.format("memory") \
.start()
```

==========

Using SQL, we joined the two results on the cell area, selected the relevant columns and created

a new "profitability" column as the ratio between the average profit and the total amount of dropoffs in the area. This result was ordered by window and profitability, with a descending order.
We decided to show 5 results so that we get for each 15 minute window, the top 5 most profitable areas for taxi drivers.

==========

```
for i in range(6) :
spark.sql('''select profit.cell_pickup AS cell, profit.window, taxis.window, profit.avg_profit  -
/ taxis.count AS profitability
from taxis join profit on
taxis.cell_dropoff = profit.cell_pickup
ORDER BY profit.window, profitability DESC''')\
.show(5,False)
query.awaitTermination(10)
```

Below, we show an example of the results, that could help the taxi drivers choose the area where to wait for clients.

==========

Query 3 - Slow Areas

The goal of this query was to detect the slowest areas. To do that, we computed the average idle time of taxis for each area, using a 300 x 300 cell grid. We used Spark Streaming to solve this query.
We started by creating tuples with only the needed information: hack license, pickup area, dropoff area, pickup datetime and dropoff datetime.

==========

```
# (taxi, (pickup area, dropoff area, pickup datetime, dropoff datetime))
datetime_per_taxi = lines_areas.map(lambda line: (line[1],(line[17],line[18],line[2],line[3])))
```

==========

We only want to compute the idle time for taxis that are available. Thus, we used the tuples as input to an update function that, for each taxi (hack license), computes the time interval between the dropoff of a ride and the pickup of the following ride and checks if that interval is longer than one hour. Afterwards, we only need to discard the taxis that are not available.

==========

```
# Input: (taxi, (pickup area, dropoff area, pickup datetime, dropoff datetime))
def updateFunctionAvailableTaxis(newValues, runningObj):
if runningObj is None:
runningObj = (newValues[0][0],newValues[0][1],newValues[0][2],newValues[0][3],0)
for v in newValues:
is_available = 1
if datetime.strptime(runningObj[3], "%Y-%m-%d %H:%M:%S")-datetime.strptime(v[2], "%Y-%m-%d  -
%H:%M:%S") > timedelta(hours=1):
is_available = 0
runningObj = (v[0], v[1], v[2], v[3], is_available)
return runningObj
# Output: (taxi, (pickup area, dropoff area, pickup datetime, dropoff datetime, is_available))
[...]
# (taxi, (pickup area, dropoff area, pickup datetime, dropoff datetime, is_available))
# Filter them to keep only tuples with the parameter is_available = 1
available_taxis = datetime_per_taxi.updateStateByKey(updateFunctionAvailableTaxis) \
.filter(lambda a: a[1][4] == 1) \
.map(lambda a: (a[0],(a[1][0], a[1][1], a[1][2], a[1][3])))
```

==========

The idle time of a taxi is the time mediating between the dropoff of a ride and the pickup of the

following ride. To compute that time interval for each taxi, we extended the tuples of each ride with the dropoff area and the dropoff datetime of the ride before that one, using an update function.
This means that, in the first time a taxi occurs in the stream, there will be no ride before that one, so the dropoff area and the dropoff datetime of the last ride will be zero. To avoid those cases, we filter the resulting tuples to keep only those with a dropoff area of the last ride different from zero.

```
# Input: (taxi, (pickup area, dropoff area, pickup datetime, dropoff datetime))
def updateFunction(newValues, runningObj):

if runningObj is None:
runningObj = (newValues[0][0],newValues[0][1],newValues[0][2],newValues[0][3],0,0)
else:
for v in newValues:
runningObj = (v[0], v[1], v[2], v[3], runningObj[1], runningObj[3])
return runningObj
# Output: (taxi, (pickup area, dropoff area, pickup datetime, dropoff datetime, old dropoff area,  -
old dropoff datetime))
[...]
# (taxi, (pickup area, dropoff area, pickup datetime, dropoff datetime, old dropoff area, old  -
dropoff datetime))
# We filter out the tuples in which the old dropoff time is 0
extended_rides = available_taxis.updateStateByKey(updateFunction) \
.filter(lambda a: a[1][4] != 0)
```

In order to compute the idle time in each area, the old dropoff area must be the same as the new pickup area. Furthermore, the old dropoff datetime must be smaller (earlier) than the new pickup datetime, because the taxi driver first has to make the dropoff and only after that can he make the new pickup. This is not always the case, probably due to some errors in the dataset, so we filtered the tuples in which that happens.

```
# (taxi, (pickup area, dropoff area, pickup datetime, dropoff datetime, old dropoff area, old  -
dropoff datetime))
# New pickup area must be the same as old dropoff area
# Old dropoff datetime must be < than new pickup datetime
processed_extended_rides = extended_rides.filter(lambda a: a[1][0] == a[1][4]) \
.filter(lambda a: datetime.strptime(a[1][5], "%Y-%m-%d %H:%M:%S"). -
timestamp() < datetime.strptime(a[1][2], "%Y-%m-%d %H:%M:%S"). -
timestamp())
```

Once the data was processed, we discarded the unnecessary elements, such as the hack license, keeping only the new pickup area, the new pickup datetime and the old dropoff datetime. Note that it would be the same keeping the new pickup area or the old dropoff area, since they must be equal.
We then computed the time interval between the new pickup datetime and the old dropoff datetime. This is the idle time of a ride in a given area.
In the end, we computed the average idle time for each area, by grouping the results by area and computing the average of the idle times computed before. We ordered the results by decreasing idle time, so the areas that are showed first are the ones with the highest idle time.

```
# (pickup area, pickup datetime, old dropoff datetime)
# Compute the idle time for each ride in each area
idle_time_per_ride_and_area = processed_extended_rides.map(lambda a: (a[1][0], (a[1][2],a[1][5])))\
```

```
.map(lambda a: (a[0], datetime.strptime(a[1][0], "%Y-%m-%d %H:% -
M:%S")-datetime.strptime(a[1][1], "%Y-%m-%d %H:%M:%S"))) \
.map(lambda a: (a[0], int(a[1].seconds))) \
.transform(lambda rdd: rdd.sortBy(lambda a: a[1], ascending= -
False))
# (pickup area, idle time)
# Compute the average idle time for each area
avg_idle_time_per_area = idle_time_per_ride_and_area.map(lambda a: (a[0],(a[1],1))) \
.reduceByKey(lambda a,b: (a[0]+b[0],a[1]+b[1])) \

.map(lambda a: (a[0], a[1][0]/a[1][1])) \
.transform(lambda rdd: rdd.sortBy(lambda a: a[1], ascending=False))
```

==========

Below, we show an example of the results, which can be very helpful for the taxi drivers, to avoid the slowest areas, where they have to wait more to pick up a client.
Note that the obtained idle times are very short. This may be due to the fact that the areas considered for the computation are very busy and, therefore, a taxi picks up a new passenger as soon as it drops off another.

Query 4 - Congested Areas

This query's objective was to compute which routes (pickup area joined with dropoff area) are congested.
For this, we considered that a congested route might be one that, when the taxis take that route, the rides increase in duration. For that, there should be alerts when a taxi has a peak in the duration of a ride, followed by at least 2 rides with increasing duration, and above average duration for the last 4 hours. Also, the alert should contain the location where the taxi started this ride which had the peak duration.
Note that, even though we were asked the obtain the congested areas, we performed all the computations
for routes. This is because performing the computations for the pickup area alone would aggregate in the same row very different situations, such as a ride that had the same areas both for pickup and dropoff, and situations where both locations are very far away. Since the query computation is based on the trip duration, it does not make much sense to compare such unstable values. Also, some increases in duration between rides that our query might get could simply be sequential rides that dropped off in sequentially further away areas, diverging away from our goal of computing congested cells.
The first thing to do besides getting the spark context object, is to apply the regular filters on the data so that we do not get bad inputs. We also get the areas (cells) to columns, considering a 300 x 300 grid.
Then we need to get only the columns that will be useful to us. Since we are already doing a map

we can also put the rows in the most convenient way for the next step, so we join the pickup area and drop off area into the key value, where every other column stays as a value inside a nested tuple.
We then get the routes with a higher number of sequential rides that increase in duration.

==========

```
# Get desired columns only
desired_columns = lines_areas.map(lambda line: (str(line[17]) + ":" + str(line[18]), (float(line -
[4]), line[6], line[7])))
# Get rows with number of rising rides through the beginning
counters = desired_columns.updateStateByKey(updateFunction)
```

==========

This function will get us the previously stated information, as it is applied through time for incoming rows with the same key value.
We have a counter to retain the number of rides that already increased in duration. If the new duration is higher than the one from the previous row then the counter is increased, otherwise it is reset. We also need to preserve the duration of the ride that initiated this whole chain of sequentially increasing duration rides so that it is later compared to the mean duration of the route itself. Another information from the first ride that needs to be passed through the rows are the longitude and latitude, so that we can present the pickup location to the user of the query.
==========

```
def updateFunction(newValues, runningObj):
if runningObj is None:
runningObj = (newValues[0][0], 0, 0, newValues[0][1], newValues[0][2], newValues[0][0])
for v in newValues:
# counter - counts the number of sequential rides
counter = runningObj[1]
# prev_dur - previous rows duration
prev_dur = runningObj[0]
# dur - target's row duration
dur = v[0]
# long - target's row long
long = runningObj[2]
# lat - target's row lat
lat = runningObj[3]
# first_dur - first ride's duration of the sequence of rides
first_dur = runningObj[4]
# if current duration is inferior to the previous one, reset counter
if dur <= prev_dur:
counter = 0
else: counter += 1
# if reset happens, also reset some variables to persist from first ride
if counter == 0:
long = v[1]
lat = v[2]
first_dur = v[0]
runningObj = (dur, counter, long, lat, first_dur)
return runningObj
```

==========

As we already have the number of sequential rides per route, we just want those that had a chain equal or bigger than 2, so we apply a filter.

```
# Only get the rows that had a sequential ride number equal or above to 2
# Example: ('315-325:379-372', (560.0, 1, 0, '-73.97625', '40.748528', 234,1))
counters_filtered = counters.filter(lambda line: line[1][1] >= 2)
```

==========

The next thing to do is compute, in a parallel way, the mean duration for every route. Thus, we begin by getting just the desired columns for this case, arranging the same way as earlier, where the key is the route, for further joining of both tables.
We then apply the function that computes the summed durations and occurrences through time for rows with the same key. Afterwards, we divide the summed values and the occurrences, leaving us with the mean.
==========

```
# Get desired columns for mean computation
```

```
desired_columns_means = lines_areas.map(lambda line: (str(line[17]) + ":" + str(line[18]), (float( -
line[4]), line[2], line[3])))
# Get the mean of duration per route
means = desired_columns_means.updateStateByKey(updateFunctionMean) \
.map(lambda line: (line[0], float(line[1][0]) / float(line[1][1]), line[1][2], line[1][3]))
```

==========

For every new input (that has the same key) we summed the duration to the bulk being passed, and increase the occurrences by one. This function also needed the pickup and dropoff datetimes to check if the means are already computing for 4 hours or more. If this is the case, then we would like to reset the values so that our computations do not get outdated information. This is done with the subtraction time wisely of the first pickup date and the new values dropoff date.

==========

```
def updateFunctionMean(newValues, runningCount):
if runningCount is None:
runningCount = (0.0, 0.0, newValues[0][1], newValues[0][2])
for v in newValues:
runningCount = (v[0] + runningCount[0], runningCount[1]+1, runningCount[2], v[2])
if datetime.strptime(v[2], "%Y-%m-%d %H:%M:%S") - datetime.strptime(runningCount[2], "%Y-%m -
-%d %H:%M:%S") > timedelta(hours=4):
runningCount = (v[0], 1, v[1], v[2])
return runningCount
```

==========

The joining of the tables, the one with the chain of rides and the one that holds the means for the routes, is the next step. We join them by the key value (routes) and we use another map to flatten the resulting rows, so that we do not end up with nested tuples.

==========

```
# Joins the two above together and flattens, took summed duration
joined = counters_filtered.join(means) \
.map(lambda line: (line[0], line[1][0][1], line[1][0][2], line[1][0][3], line[1][0][4],  -
line[1][1]))
```

==========

The next thing to be done is to compare the first duration of the ride chains with the mean for the corresponding route. We want to get only the ones with a higher first ride duration, so we filter the results with the appropriate columns.
Lastly, only the useful values for the user are meant to be shown. We decided to include the number

of rides that were chained during the computations because the user might get additional information from this, an alert with a high chain number might be very congested, this gives us a sense of intensity.

==========

```
# Per position meanings: (area, nr_rides, first_long, first_lat, first_dur, area_mean_dur)
# Filter out rows that have a first ride duration equal or less than the mean of the same route
peak_filter = joined.filter(lambda line: float(line[4]) > float(line[5]))
# Get only useful information to show
peak_filter = peak_filter.map(lambda line: (line[0], line[1], line[2], line[3]))
peak_filter.pprint()
```

==========

Below, we show an example of the results. This query is quite useful to the taxi drivers, because they can avoid certain areas that are the origin of congested routes.

Query 5 - Most Pleasant Taxi Driver

The goal of this query was to show the most pleasant taxi driver in a given day. We solved this problem using Structured Spark Streaming.

If a taxi driver is pleasant, then it makes sense that the passengers give a higher tip. However, the tip also depends on the total price of a ride. Therefore, we computed the tip percentage, i.e., how much of the total amount paid in a ride corresponds to the tip given by the passenger. It is best to compute this percentage to compare between taxi drivers, because there can be taxi drivers that had longer and more expensive rides and so the total tip is higher than taxi drivers that had shorter and less expensive rides, even if the former are not as nice as the latter.

After adding a new column to the dataframe with the tip percentage, we created a window of 1 day based on the dropoff datetime, to get the results for whole days. We grouped by that window and by hack license (i.e., taxi driver) and computed the average of the tip percentage for each taxi driver. The goal of computing the average is to guarantee that taxi drivers that had more rides in a day are not in advantage with respect to taxi drivers that had less rides.

In the end, we ordered the results by the average of the tip percentage in descending order and limited the results to show only the taxi driver with the highest average tip percentage (the most pleasant
taxi driver).

==========

```
# Get tip percentage
lines = lines.withColumn("tip_percentage", lines["tip_amount"]/lines["total_amount"]) \
.drop("value","key","topic","partition","offset","timestamp","timestampType")
# Compute the average tip percentage
avg_tip_percentage = lines.groupBy(window("dropoff_datetime","1 days"),"hack_license") \
.avg("tip_percentage") \
1
.orderBy('avg(tip_percentage)',ascending=False) \
.limit(1)
```

==========

Below, we show an example of the results.

Query 6 - Revenue Discrimination for the Highest Paying Areas

As an optional query we decided to look into how the revenue discrimination looks like for the pickup areas where the taxi drivers make most money, in each 2 hour window. A pickup area may look very lucrative when the total amount received by the drivers is high, but is it really profit or is there a high percentage of surcharge or tolls included, masking the actual revenue?

To do this using Spark Structured Streaming, we first selected the useful variables, namely the pickup datetime and cell, total amout, fare amount, surcharge, tax, tip amount and tolls amount. We grouped the results by cell pickup and by each 2 hour window. Then, for each pickup area in each window, we summed the total received amount and calculated the percentage of each kind of payment in the total received amount. This gives us the revenue discrimination. In the end we ordered by the total amount and limited the results to 5, so that we show for each 2 hour window the revenue discrimination of the top 5 highest paying areas.

==========

```
# Compute the sum of the total amount and what percentage corresponds to the fare amount,
# the surcharge, the mta_tax, the tip_amount and the tolls_amount
statistics = lines.groupBy("cell_pickup", window("pickup_datetime","2 hours")) \
.agg(round(sum("total_amount"),3).alias("sum_total_amount"), \
round(sum("fare_amount")/sum("total_amount"),3).alias("fare"), \
```

```
round(sum("surcharge")/sum("total_amount"),3).alias("surcharge"),\
round(sum("mta_tax")/sum("total_amount"),3).alias("mta_tax"),\
round(sum("tip_amount")/sum("total_amount"),3).alias("tip"),\
round(sum("tolls_amount")/sum("total_amount")).alias("tolls"))\
.orderBy("sum_total_amount", ascending=False) \
.limit(5)
```
==========

Below, we show an example of the results.
The results show that, as expected, most of the received amount corresponds to the actual paid fare. For these areas, the percentage of total amount that corresponds to the tip varies between 4% and 12%. The surcharge percentage does not seem to vary a lot, namely between 1% and 3%. Tolls

seem to not be included in these areas. Mta tax is actually a fixed value per trip of 50 cents, so it is not relevant for this price discrimination.

Query 7 - Routes with Highest Total Amount per Mile

For this optional query we decided to check in which routes the total paid amount per mile is highest in the last hour. This is useful for the taxi drivers, because they can decide in which areas to pickup passengers, based on that routes, since the more a passenger pays per mile, the more advantageous it is for the taxi driver.
We started by creating a window of 1 hour based on the pickup datetime and grouping the results by that window and by route. For each route, we summed the total paid amount and the total distance traveled in all the rides that used that route.
==========

```
# Compute the summed total amount and the summed trip distance
total_per_route = lines.groupBy(window("pickup_datetime", "1 hours"),"route") \
.agg(sum("total_amount").alias("sum_total_amount"),
sum("trip_distance").alias("total_distance"))
```
==========

We then added a new column to the dataframe with the division between the summed total paid amount and the total trip distances. This gives us the total amount per mile traveled. In the end, we only needed to order the results by this total amount per mile and to limit the results to show only the 10 routes with the highest total amount per mile.
==========

```
# Add a new column with the division between the summed total amount and the summed trip distance
profit_per_route = total_per_route.withColumn("amount_per_mile",
total_per_route["sum_total_amount"]/total_per_route["total_distance"])\
.drop("sum_total_amount","total_distance")
# Order the results by the total amount per mile
most_profitable_routes = profit_per_route.orderBy("amount_per_mile", ascending=False) \
.limit(10)
```
Below, we show an example of the results.