BIG DATA PROCESSING
NYC Rideshare Data Analysis

Below are the common APIs used for the implementation of spark scripts:

- **os** - This module provides a way of accessing Operating System dependent functionalities. In our scripts, this module is used for setting up the configurations to access the files stored in the amazon S3 bucket, so that hardcoding of the credentials is not needed.

- **sys** - This module will give access to the variables and functions that will interact with the python interpreter. This module will be used to manipulate the python's runtime environment.

- **SparkSession** - This SparkSession class is the crucial component of the Apache Spark's core API, it will enable us in working with the structured data and executing processing of data tasks in a distributed manner, In our below spark scripts this class is performing several spark operations, namely reading data from sources, processing , transformation of the data and also writing data to storage.

- **from_unixtime** - This function is imported from Spark SQL, it is used for the purpose of converting Unix timestamp values into date-time format, so that it is more comprehensible to us. In our spark scripts this function is used in converting the unix timestamp value present in the column 'date' of the joined_DataFrame into a date value, which will be easier for the processing of date-related information.

- **col** - This function is also imported from Spark SQL, this is used for the purpose of referencing existing columns or even in creating new columns within DataFrames. It will also enable us to manipulate and transform the columns of the DataFrames for efficient data processing.

- **month** - The month function is part of the Spark SQL in Apache Spark, This will provide us with a range of built-in functions that will facilitate manipulation and transformation operations on the DataFrames. In our spark scripts this function is used to extract the month component from the converted 'date' column.

- **concat** - This concat function is part of the Spark SQL, this function is used for concatenation or combining strings from multiple columns into a single string column. In our spark scripts, this function is used to merge the newly created 'month' column with the existing 'business' column.

- **FloatType** - This data type is provided by the pyspark.sql.types module. It will represent the floating-point numbers, that are numerical values with the decimal points. In our spark scripts, this data type is used to convert the data type of certain columns from string to floating-point numbers to perform numerical operations.

Task-1: Merging Datasets

1. **For question 1**, I have loaded both 'rideshare_data' and taxi_zone_lookup' csv files from the shared S3 bucket using SparkSession.
2. **For question 2**, I have applied the 'join' function on ride_share data and taxi_zone_lookup tables based on the field **pickup_location,** and the joined new columns were renamed. Then I joined the intermediate result with a taxi_zone_lookup table based on **dropoff_location,** and again the joined columns were renamed according to the task.
3. **For question 3**, I have converted the date format in the 'date' column from UNIX timestamp to the "yyyy-MM-dd" format using the '**from_unixtime'** function. Then the **'col'** function is used for referencing the 'date' column.
4. **For question 4**, I am printing the schema using the **'printSchema'** function and I am printing the number of rows using the **'count'** function.

Challenges faced in Task 1:

- For me, grasping the knowledge of applying Spark operations on DataFrames is a bit challenging and it involves a steep learning curve. But when I look into the Spark operations and the API documentation, I am able to understand the Spark functionalities and I learnt where to use them.

Knowledge gained in Task 1:

- I understood the Spark operations, functionalities and their APIs

Question-1

Loading the datasets:

Implementation of the logic :

```
    # Loading the csv files
    data_rideshare =
spark.read.format("csv").option("header",
"true").load("s3a://" + s3_data_repository_bucket +
"/ECS765/rideshare_2023/rideshare_data.csv")
    data_taxi_zone_lookup =
spark.read.format("csv").option("header",
"true").load("s3a://" + s3_data_repository_bucket +
"/ECS765/rideshare_2023/taxi_zone_lookup.csv")
```

Question-2
Implementing Join Function:

Implementation of the logic :

```
# loading taxi_zone_lookup table and renaming the columns
    data_taxi_zone_lookup =
spark.read.format("csv").option("header",
"true").load("s3a://" + s3_data_repository_bucket +
"/ECS765/rideshare_2023/taxi_zone_lookup.csv") \
        .withColumnRenamed("LocationID",
"pickup_LocationID") \
        .withColumnRenamed("Borough", "Pickup_Borough") \
        .withColumnRenamed("Zone", "Pickup_Zone") \
        .withColumnRenamed("service_zone",
"Pickup_service_zone")

# first join operation based on pickup_location
    data_join =
data_rideshare.join(data_taxi_zone_lookup,
data_rideshare["pickup_location"] ==
data_taxi_zone_lookup["pickup_LocationID"], "inner") \
        .drop("pickup_LocationID")
```

```
# loading taxi_zone_lookup again for performing second
join opearion and renaming columns
    dropoff_taxi_zone_lookup =
spark.read.format("csv").option("header",
"true").load("s3a://" + s3_data_repository_bucket +
"/ECS765/rideshare_2023/taxi_zone_lookup.csv") \
        .withColumnRenamed("LocationID",
"dropoff_LocationID") \
        .withColumnRenamed("Borough", "Dropoff_Borough") \
        .withColumnRenamed("Zone", "Dropoff_Zone") \
        .withColumnRenamed("service_zone",
"Dropoff_service_zone")

# second join opeartion for dropoff location
    data_join = data_join.join(dropoff_taxi_zone_lookup,
data_join["dropoff_location"] ==
dropoff_taxi_zone_lookup["dropoff_LocationID"], "inner") \
        .drop("dropoff_LocationID")
```

Output visualization:

```
Number of rows: 69725864
root
 |-- business: string (nullable = true)
 |-- pickup_location: string (nullable = true)
 |-- dropoff_location: string (nullable = true)
 |-- trip_length: string (nullable = true)
 |-- request_to_pickup: string (nullable = true)
 |-- total_ride_time: string (nullable = true)
 |-- on_scene_to_pickup: string (nullable = true)
 |-- on_scene_to_dropoff: string (nullable = true)
 |-- time_of_day: string (nullable = true)
 |-- date: string (nullable = true)
 |-- passenger_fare: string (nullable = true)
 |-- driver_total_pay: string (nullable = true)
 |-- rideshare_profit: string (nullable = true)
 |-- hourly_rate: string (nullable = true)
 |-- dollars_per_mile: string (nullable = true)
 |-- Pickup_Borough: string (nullable = true)
 |-- Pickup_Zone: string (nullable = true)
 |-- Pickup_service_zone: string (nullable = true)
 |-- Dropoff_Borough: string (nullable = true)
 |-- Dropoff_Zone: string (nullable = true)
 |-- Dropoff_service_zone: string (nullable = true)
```

Question: 3
Converting UNIX timestamp to “yyyy-MM-dd” format

Implementation of the logic:

```
# converting unix timestamp to "yyyy-mm-dd" format
    data_join = data_join.withColumn("date",
from_unixtime(col("date"), "yyyy-MM-dd"))
```

Output:

```
ost is considered as scanning 4194304 bytes.
root
 |-- business: string (nullable = true)
 |-- pickup_location: string (nullable = true)
 |-- dropoff_location: string (nullable = true)
 |-- trip_length: string (nullable = true)
 |-- request_to_pickup: string (nullable = true)
 |-- total_ride_time: string (nullable = true)
 |-- on_scene_to_pickup: string (nullable = true)
 |-- on_scene_to_dropoff: string (nullable = true)
 |-- time_of_day: string (nullable = true)
 |-- date: string (nullable = true)
 |-- passenger_fare: string (nullable = true)
 |-- driver_total_pay: string (nullable = true)
 |-- rideshare_profit: string (nullable = true)
 |-- hourly_rate: string (nullable = true)
 |-- dollars_per_mile: string (nullable = true)
 |-- Pickup_Borough: string (nullable = true)
 |-- Pickup_Zone: string (nullable = true)
 |-- Pickup_service_zone: string (nullable = true)
 |-- Dropoff_Borough: string (nullable = true)
 |-- Dropoff_Zone: string (nullable = true)
 |-- Dropoff_service_zone: string (nullable = true)
```

Schema output:

```
+--------+---------------+----------------+-----------+-----------------+---------------+------------------+-------------------+----------+----------+--------------+----------------+----------------+-----------+----------+
|business|pickup_location|dropoff_location|trip_length|request_to_pickup|total_ride_time|on_scene_to_pickup|on_scene_to_dropoff|time_of_day|      date|passenger_fare|driver_total_pay|rideshare_profit|hourly_rate|dollars_per_mile|Pickup_Borough|        Pickup_Zone|Pickup_service_zone|Dropoff_Borough|        Dropoff_Zone|Dropoff_service_zone|
+--------+---------------+----------------+-----------+-----------------+---------------+------------------+-------------------+----------+----------+--------------+----------------+----------------+-----------+----------+
|    Uber|            151|             244|       4.98|            226.0|          761.0|              19.0|              780.0|   morning|2023-05-22|         22.82|           13.69|            9.13|      63.18|      2.75|     Manhattan|     Manhattan Valley|    Yellow Zone|     Manhattan|Washington Height...|           Boro Zone|
|    Uber|            244|              78|       4.35|            197.0|         1423.0|             120.0|             1543.0|   morning|2023-05-22|         24.27|            19.1|            5.17|      44.56|      4.39|     Manhattan|Washington Height...|       Boro Zone|         Bronx|       East Tremont|           Boro Zone|
|    Uber|            151|             138|       8.82|            171.0|         1527.0|              12.0|             1539.0|   morning|2023-05-22|         47.67|           25.94|           21.73|      60.68|      2.94|     Manhattan|     Manhattan Valley|    Yellow Zone|        Queens|  LaGuardia Airport|            Airports|
|    Uber|            138|             151|       8.72|            260.0|         1761.0|              44.0|             1805.0|   morning|2023-05-22|         45.67|           28.01|           17.66|      55.86|      3.21|        Queens|  LaGuardia Airport|       Airports|     Manhattan|     Manhattan Valley|       Yellow Zone|
|    Uber|             36|             129|       5.05|            208.0|         1762.0|              37.0|             1799.0|   morning|2023-05-22|         33.49|           26.47|            7.02|      52.97|      5.24|      Brooklyn|      Bushwick North|       Boro Zone|        Queens|     Jackson Heights|           Boro Zone|
+--------+---------------+----------------+-----------+-----------------+---------------+------------------+-------------------+----------+----------+--------------+----------------+----------------+-----------+----------+--------------+--------------------+---------------+--------------+--------------------+--------------------+
only showing top 5 rows
```

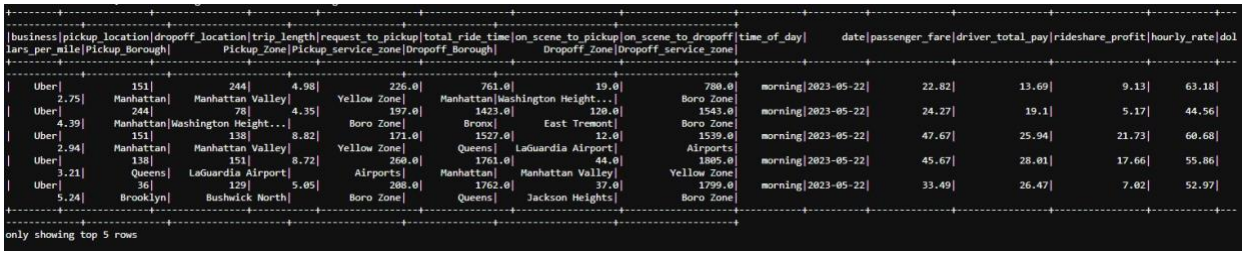
Question:4
Printing the number of rows

Implementation of the logic:

```
# printing the number of rows
    print("Number of rows:", data_join.count())
    data_join.printSchema()
    data_join.show(5)
```

Outputs:

```
2024-03-28 11:41:27,297 INFO scheduler.DAGScheduler: Job 5 finished: count at NativeMethodAccessorImpl.java:0, took 147.160168 s
Number of rows: 69725864
root
```



```
|business|pickup_location|dropoff_location|trip_length|request_to_pickup|total_ride_time|on_scene_to_pickup|on_scene_to_dropoff|time_of_day|      date|passenger_fare|driver_total_pay|rideshare_profit|hourly_rate|dollars_per_mile|Pickup_Borough|         Pickup_Zone|Pickup_service_zone|Dropoff_Borough|        Dropoff_Zone|Dropoff_service_zone|
+--------+---------------+----------------+-----------+-----------------+---------------+------------------+-------------------+-----------+----------+--------------+----------------+----------------+-----------+----------------+--------------+--------------------+-------------------+---------------+--------------------+--------------------+
|    Uber|            151|             244|       4.98|            226.0|          761.0|              19.0|              780.0|    morning|2023-05-22|         22.82|           13.69|            9.13|      63.18|            2.75|     Manhattan|     Manhattan Valley|        Yellow Zone|      Manhattan|Washington Height...|           Boro Zone|
|    Uber|            244|              78|       4.35|            197.0|         1423.0|             120.0|             1543.0|    morning|2023-05-22|         24.27|            19.1|            5.17|      44.56|            4.39|     Manhattan|Washington Height...|          Boro Zone|          Bronx|        East Tremont|           Boro Zone|
|    Uber|            151|             138|       8.82|            171.0|         1527.0|              12.0|             1539.0|    morning|2023-05-22|         47.67|           25.94|           21.73|      60.68|            2.94|     Manhattan|     Manhattan Valley|        Yellow Zone|         Queens|   LaGuardia Airport|            Airports|
|    Uber|            138|             151|       8.72|            260.0|         1761.0|              44.0|             1805.0|    morning|2023-05-22|         45.67|           28.01|           17.66|      55.86|            3.21|        Queens|   LaGuardia Airport|           Airports|      Manhattan|    Manhattan Valley|        Yellow Zone|
|    Uber|             36|             129|       5.05|            208.0|         1762.0|              37.0|             1799.0|    morning|2023-05-22|         33.49|           26.47|            7.02|      52.97|            5.24|      Brooklyn|      Bushwick North|          Boro Zone|         Queens|     Jackson Heights|           Boro Zone|
+--------+---------------+----------------+-----------+-----------------+---------------+------------------+-------------------+-----------+----------+--------------+----------------+----------------+-----------+----------------+--------------+--------------------+-------------------+---------------+--------------------+--------------------+
only showing top 5 rows
```

```
ost is considered as scanning 4194304 bytes.
root
 |-- business: string (nullable = true)
 |-- pickup_location: string (nullable = true)
 |-- dropoff_location: string (nullable = true)
 |-- trip_length: string (nullable = true)
 |-- request_to_pickup: string (nullable = true)
 |-- total_ride_time: string (nullable = true)
 |-- on_scene_to_pickup: string (nullable = true)
 |-- on_scene_to_dropoff: string (nullable = true)
 |-- time_of_day: string (nullable = true)
 |-- date: string (nullable = true)
 |-- passenger_fare: string (nullable = true)
 |-- driver_total_pay: string (nullable = true)
 |-- rideshare_profit: string (nullable = true)
 |-- hourly_rate: string (nullable = true)
 |-- dollars_per_mile: string (nullable = true)
 |-- Pickup_Borough: string (nullable = true)
 |-- Pickup_Zone: string (nullable = true)
 |-- Pickup_service_zone: string (nullable = true)
 |-- Dropoff_Borough: string (nullable = true)
 |-- Dropoff_Zone: string (nullable = true)
 |-- Dropoff_service_zone: string (nullable = true)
```

Task2 : Aggregation of data

APIs that are used in this task

- **count** - This function is provided by pyspark.sql.functions and it is an aggregate function. It will count all the non-null values in the DataFrame. Here in the spark implementation we used this function in Task 2, Question 1 to count the trip_count for each business in each month.
- **lit** - This function is provided by the Spark SQL functions. This function is used to create a new column with a constant value.
- **StringType** - This class is provided by pyspark.sql.types. It will be useful for specifying a column datatype as string.
- **StructType**: This function is provided by the pyspark.sql.types. This function will allow us to define the overall structure of a DataFrame.
- **StructField**: This function is provided by the pyspark.sql.types. This function is used to specify the individual columns and their properties such as data type and name.
- **repartition** - repartition() function here, it is used to optimize the data distribution and reduce the number of small files generated during certain operations. This function will repartition the no_of_trips Data Frame into a single partition.

Detailed Explanation

- **Question 1:**
  - ➔ From the **'date'** column, extracting the month values to a new column named **'month'**.
  - ➔ I have selected only needed columns, that is **'business'** column and **'month'** column for this part for the simplification process and to reduce the memory usage.
  - ➔ I am performing a concatenation operation with **concat** function on the column business and month and creating a new column named **'business month'.**
  - ➔ Counting the number of trips for each business in each month using the count function.
  - ➔ Repartitioning the resulting Data Frame into a single file and writing it as a csv file for the plotting.
  - ➔ I used pandas and matplotlib libraries to plot the results as a histogram.

- **Question 2:**
  - ➔ From the **'date'** column, extracting the month values to a new column named **'month'**.
  - ➔ I have selected only needed columns, that is **'business'** column and **'month'** column for this part for the simplification process and to reduce the memory usage.
  - ➔ I am performing a concatenation operation with **concat** function on the column business and month and creating a new column named **'business month'.**
  - ➔ Casting rideshare_profit field to **float-type** for the further analysis.
  - ➔ Calculating the platform's profits (rideshare-profit) by applying **groupBy()** function to the 'business month' column and then using the **sum()** function to calculate the profit for each business in each month.
  - ➔ Repartitioning the resulting Data Frame into a single file and writing it as a csv file for the plotting.
  - ➔ I used pandas and matplotlib libraries to plot the results as a histogram.

- **Question 3:**
  - ➔ From the **'date'** column, extracting the month values to a new column named **'month'**.
  - ➔ I have selected only needed columns, that is **'business'** column and **'month'** column for this part for the simplification process and to reduce the memory usage.
  - ➔ I am performing a concatenation operation with **concat** function on the column business and month and creating a new column named **'business month'.**
  - ➔ Calculating the driver's earnings (driver_total_pay) by applying **groupby()** function to the 'business month' column and then using the sum**()** function to calculate earnings for each business in each month.
  - ➔ Repartitioning the resulting Data Frame into a single file and writing it as a csv file for the plotting.

- ➔ I used pandas and matplotlib libraries to plot the results as a histogram.

- **Question 4:**

  Insights from the trip count:
  - ➔ In general I can notice that Uber has higher trip counts than Lyft.
  - ➔ The trend of the trip count for both the businesses are increasing overall.
  - ➔ May month has the highest trip count value.
  - ➔ From the obtained results, as a driver I can notice that the month of may is presenting an opportunity to maximize earnings because the demand is potentially increasing.
  - ➔ From the perspective of the company's CEO, I can make strategic decisions like implementing targeted marketing campaigns during the month of may to attract more drivers and to expand the market share as well as i can optimize the resource allocation by ensuring sufficient supply to meet the surge in demand.

  Insights from the platform profits:
  - ➔ In overall comparison, Uber has the higher platform profits than Lyft.
  - ➔ The platform profits for both the businesses are increasing towards may.
  - ➔ Except in May, Lyft is showing negative profit for all months.
  - ➔ From the analytical results, as a stockbroker I can leverage the insights to guide investment decisions by investing into uber.
  - ➔ From the perspective of the company's CEO, by looking into the company's finance statement, I can be able to make decisions for better cost management and better pricing strategies.

  Insights from the drivers earnings:
  - ➔ In general, by looking at the results, I can notice that Uber drivers are earning higher compared to that of Lyft.
  - ➔ Increase of driver's earnings for both the businesses is happening towards may month.
  - ➔ May month is considered to be the profitable month for the drivers in both the businesses.
  - ➔ As a driver, I can apply for a job in Uber, so that my earnings will be higher when compared to Lyft.
  - ➔ As a CEO of the company, I should monitor the drivers earnings, so that I can be able to provide insights on the driver's satisfaction and retention rates.

Challenges faced in Task2:

- When I was creating a histogram for visualizing the distribution of values, I noticed a significant discrepancy between the small and large values, which finally resulted in an inadequate representation of the entire value range. The histogram is not able to show all the values properly. For this problem, I used logarithmic scaling. When I applied logarithmic transformation to the data, the discrepancy between the large and small values were significantly compressed and I got a better visualization for better analysation and graph interpretation.

Knowledge gained in Task2:

- I understood how to compute the sum and total count using the spark operations.
- Now I can write the results in a csv file.
- Learned how to scale extremely large values for better visualization.
- Learned aggregation of data.
- Learned how to create new Dataframe from the schema

Question:1

Counting the number of trips for each business in each month.

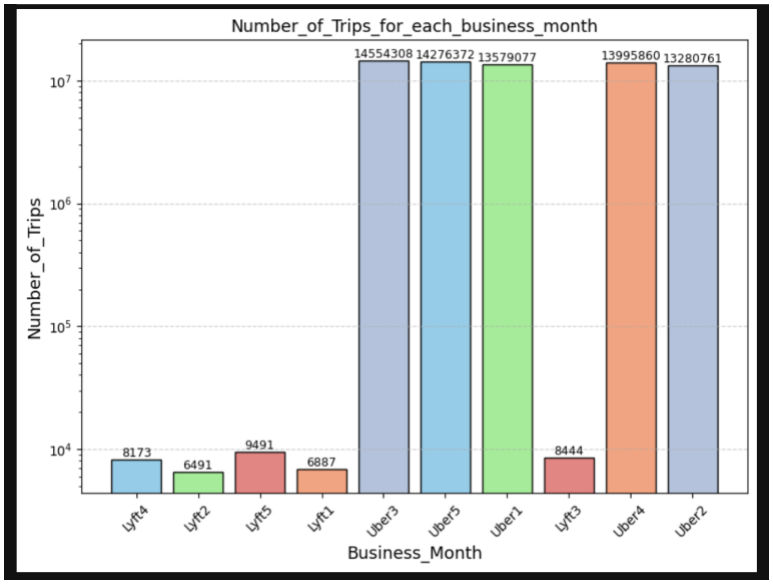
Implementation of the logic

```
    # extracting month from "date" column
    data_join = data_join.withColumn("month",
month(col("date")))

    # performing concatenation for "business" and "month"
column to create "business_month" column
    data_join = data_join.withColumn("business_month",
concat(col("business"), col("month")))

    # counting the no.of.trips based on grouping by
"business_month"
    no_of_trips =
data_join.groupBy("business_month").count().withColumnRena
med("count", "no_of_trips")

    # repartitioning the datadframe to avoid multiple
data files
    no_of_trips = no_of_trips.repartition(1)

    # writing the resulting dataframe into a single csv
for plotting
    output_path_file = "s3a://" +
s3_data_repository_bucket +
"/ECS765/rideshare_2023/number_of_trips.csv"
    no_of_trips.write.csv(output_path_file,
mode="overwrite", header=True)
```

Output:

```
jovyan@jupyter-ec23543:~/teaching_material/ECS765P/coursework$ ccc method bucket ls //data-repository-bkt/ECS765/rideshare
_2023/
                           PRE average_waiting_time/
                           PRE average_waiting_times/
                           PRE avg_waiting_time/
                           PRE number_of_trips.csv/
                           PRE total_trip_count_part_1/
                           PRE total_trip_count_part_2/
2024-02-02 12:22:50 6119240191 rideshare_data.csv
2024-03-06 10:48:21     260375 sample_data.csv
2024-02-02 12:24:26      12322 taxi_zone_lookup.csv
jovyan@jupyter-ec23543:~/teaching_material/ECS765P/coursework$ ccc method bucket ls //data-repository-bkt/ECS765/rideshare
_2023/number_of_trips.csv/
2024-03-30 23:19:46          0 _SUCCESS
2024-03-30 23:19:45        157 part-00000-ee97b8ab-08e8-42ef-a3e8-a59580596190-c000.csv
jovyan@jupyter-ec23543:~/teaching_material/ECS765P/coursework$ ccc method bucket cp bkt://data-repository-bkt/ECS765/r
hare_2023/number_of_trips.csv/part-00000-ee97b8ab-08e8-42ef-a3e8-a59580596190-c000.csv ./
download: s3://data-repository-bkt/ECS765/rideshare_2023/number_of_trips.csv/part-00000-ee97b8ab-08e8-42ef-a3e8-595805961
90-c000.csv to ./part-00000-ee97b8ab-08e8-42ef-a3e8-a59580596190-c000.csv
```

Histogram:



Question:2

Calculation of platform's profit for each business in each month

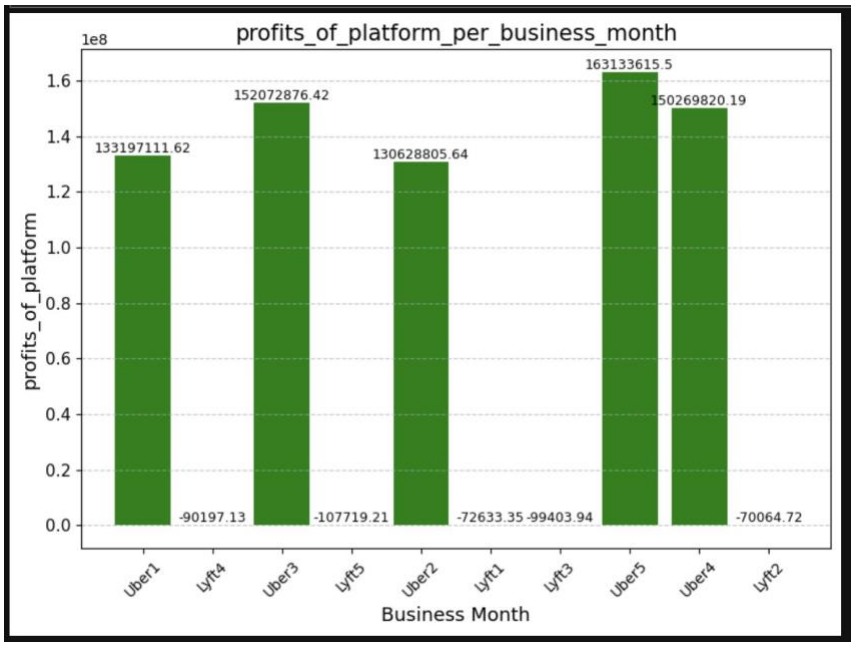
Implementation of the logic:

```
    # Grouping by 'business_month' and adding the
platform's profits
    profit_permonth=
data_join.groupBy("business_month").sum("rideshare_profit"
).withColumnRenamed("sum(rideshare_profit)",
"platform_profits")

    # repartitioning the datadframe to avoid multiple
data files
    profit_permonth = profit_permonth.repartition(1)

    # writing the resulting dataframe into a single csv
for plotting
    output_path_file = "s3a://" +
s3_data_repository_bucket +
"/ECS765/rideshare_2023/profits.csv"
    profit_permonth.write.csv(output_path_file,
mode="overwrite", header=True)
```

Output:

```
jovyan@jupyter-ec23543:~/teaching_material/ECS765P/coursework$ ccc method bucket ls //data-repository-bkt/ECS765/rideshare
_2023/
                           PRE average_waiting_time/
                           PRE average_waiting_times/
                           PRE avg_waiting_time/
                           PRE profits.csv/
                           PRE total_trip_count_part_1/
                           PRE total_trip_count_part_2/
2024-02-02 12:22:50 611924019 rideshare_data.csv
2024-03-06 10:48:21     260375 sample_data.csv
2024-02-02 12:24:26      12322 taxi_zone_lookup.csv
jovyan@jupyter-ec23543:~/teaching_material/ECS765P/coursework$ ccc method bucket ls //data-repository-bkt/ECS765/rideshare
_2023/profits.csv/
2024-03-30 23:31:12          0 _SUCCESS
2024-03-30 23:31:12        291 part-00000-f6244104-d929-4994-8595-0bf52175fd16-c000.csv
jovyan@jupyter-ec23543:~/teaching_material/ECS765P/coursework$ ccc method bucket cp bkt://data-repository-bkt/ECS765/rides
hare_2023/number_of_trips.csv/part-00000-f6244104-d929-4994-8595-0bf52175fd16-c000.csv ./
fatal error: An error occurred (404) when calling the HeadObject operation: Key "ECS765/rideshare_2023/number_of_trips.csv
/part-00000-f6244104-d929-4994-8595-0bf52175fd16-c000.csv" does not exist
jovyan@jupyter-ec23543:~/teaching_material/ECS765P/coursework$ ccc method bucket cp bkt://data-repository-bkt/ECS765/rides
hare_2023/profits.csv/part-00000-f6244104-d929-4994-8595-0bf52175fd16-c000.csv ./
download: s3://data-repository-bkt/ECS765/rideshare_2023/profits.csv/part-00000-f6244104-d929-4994-8595-0bf52175fd16-c000.
csv to ./part-00000-f6244104-d929-4994-8595-0bf52175fd16-c000.csv
```

Histogram:



Question:3

Calculation of driver's earnings for each business in each month

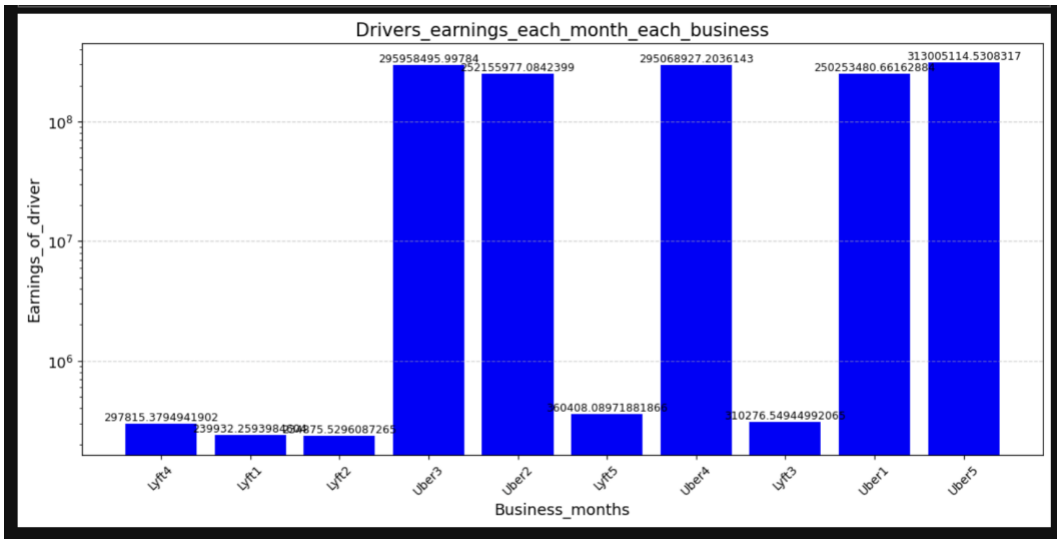
Implementation of the logic:

```
    # Create 'business_month' column
    data_join = data_join.withColumn("business_month",
concat(col("business"), col("month")))

    # grouping by 'business_month' and adding the
driver's earnings
    permonth_earnings =
data_join.groupBy("business_month").sum("driver_total_pay"
).withColumnRenamed("driver_total_pay", "driver_earnings")

    # repartitioning the datadframe to avoid multiple
data files
    permonth_earnings = permonth_earnings.repartition(1)

    # writing the resulting dataframe into a single csv
for plotting
    output_path_file = "s3a://" +
s3_data_repository_bucket +
"/ECS765/rideshare_2023/driver_earnings.csv"
    permonth_earnings.write.csv(output_path_file,
mode="overwrite", header=True)
```

Output

```
jovyan@jupyter-ec23543:~/teaching_material/ECS765P/coursework$ ccc method bucket ls //data-repository-bkt/ECS765/rideshare
_2023/
                           PRE average_waiting_time/
                           PRE average_waiting_times/
                           PRE avg_waiting_time/
                           PRE driver_earnings.csv/
                           PRE profits.csv/
                           PRE total_trip_count_part_1/
                           PRE total_trip_count_part_2/
2024-02-02 12:22:50 6119240191 rideshare_data.csv
2024-03-06 10:48:21     260375 sample_data.csv
2024-02-02 12:24:26      12322 taxi_zone_lookup.csv
jovyan@jupyter-ec23543:~/teaching_material/ECS765P/coursework$ ccc method bucket ls //data-repository-bkt/ECS765/rideshare
_2023/driver_earnings.csv/
2024-03-30 23:56:18          0 _SUCCESS
2024-03-30 23:56:17        289 part-00000-c5caccca-ee3f-46d5-b9f7-3cc028743200-c000.csv
jovyan@jupyter-ec23543:~/teaching_material/ECS765P/coursework$ ccc method bucket cp bkt://data-repository-bkt/ECS765/rides
hare_2023/driver_earnings.csv/part-00000-c5caccca-ee3f-46d5-b9f7-3cc028743200-c000.csv ./
download: s3://data-repository-bkt/ECS765/rideshare_2023/driver_earnings.csv/part-00000-c5caccca-ee3f-46d5-b9f7-3cc0287432
00-c000.csv to ./part-00000-c5caccca-ee3f-46d5-b9f7-3cc028743200-c000.csv
```

Histogram:



Task: 3 Top-K Processing

APIs used in this Task:

- **lit:** This function is provided by the Spark SQL functions. This function is used to create a new column with a constant value.
- **Count:** This function is provided by pyspark.sql.functions and it is an aggregate function. It will count all the non-null values in the DataFrame.
- **StructType:** This function is provided by the pyspark.sql.types. This function will allow us to define the overall structure of a DataFrame.
- **StructField:** This function is provided by the pyspark.sql.types. This function is used to specify the individual columns and their properties such as data type and name.
- **Sum:** This function is provided by pyspark.sql.functions. It will aggregate the values in a dataframe column or within the group.
- **StringType:** This class is provided by pyspark.sql.types. It will be useful for specifying a column datatype as string.
- **window:** This function will be performing ranking over partitions of data.

Detailed Explanation:

- **Question 1:**
  - ➔ Used only selected columns for the simplified process and to reduce memory usage.
  - ➔ To identify the top 5 pickup_borough each month, I used the **groupby()** function that groups the DataFrame by the 'pickup_borough' column and the month extracted from the 'date' column.
  - ➔ I used window specification, which will be used for performing the window functions (ranking) over partitions of data. It will partition the data by 'month'

column and it will order the rows within each partition by the ‘trip count’ column in descending order.

- ➔ I am filtering the top 5 pickup boroughs for each month based on ‘trip count’. The **filter()** will keep only the rows where the rank column is less than or equal to 5, then the **drop()** function will drop the ‘rank’ column, as it is no longer needed.

- **Question 2:**
  - ➔ Used only selected columns for the simplified process and to reduce memory usage.
  - ➔ To identify the top 5 dropoff_borough each month, I used the **groupby()** function that groups the DataFrame by the ‘dropoff_borough’ column and the month extracted from the ‘date’ column.
  - ➔ I used window specification, which will be used for performing the window functions (ranking) over partitions of data. It will partition the data by ‘month’ column and it will order the rows within each partition by the ‘trip count’ column in descending order.
  - ➔ I am filtering the top 5 dropoff boroughs for each month based on ‘trip count’. The **filter()** will keep only the rows where the rank column is less than or equal to 5, then the **drop()** function will drop the ‘rank’ column, as it is no longer needed.

- **Question 3:**
  - ➔ Used only selected columns for the simplified process and to reduce memory usage.
  - ➔ I have created a new column ‘Route’ by concatenating the ‘Pickup_Borough_Name’ column, a literal string ‘to’ and the ‘Dropoff_Borough_Name’ using the **concat()** function.
  - ➔ It will group the data_join DataFrame by the column ‘Route’ using **groupby()** function.
  - ➔ Within each route, it will aggregate the ‘driver_total_pay’ column using the **sum()** function and rename the result as ‘total profit’.
  - ➔ I have ordered the ‘profit’ DataFrame by ‘total_profit’ column in descending order using the orderBy**()** function and printed the top 30 earnest routes.

- **Question 4:**
  - ➔ Manhattan has the highest trip count for both pickup and dropoff boroughs overall, that indicates a high demand for transportation services in that area.
  - ➔ The “Manhattan to Manhattan” route has the highest total profit while compared to the other routes.
  - ➔ As a driver, these insights suggest that aligning most of their trips to the Manhattan route would be a good move.
  - ➔ From the CEO's perspective, with these insights I can make some strategic decisions like resource allocation to manhattan route to meet the demands and also amplify the market efforts into this route.

Challenges faced in Task 3:

- No challenges were faced in this Task.

Knowledge gained in Task 3:

- I learned how to do Top K-processing using spark
- Now I am able to concatenate the values of 2 columns into a new column

Question:1
Identifying Top 5 popular pickup boroughs each month

Implementation of the logic:

```
    # Grouping by Pickup_Borough and Month, count the
number of trips
    count_of_pickup_borough =
data_join.groupBy("Pickup_Borough",
month("date").alias("Month")).count().withColumnRenamed("c
ount", "trip_count")

    # Defining a window specification to rank the
boroughs within each month based on trip_count
    window_specification =
Window.partitionBy("Month").orderBy(col("trip_count").desc
())

    # Adding a rank column based on trip_count within
each month
    count_of_pickup_borough =
count_of_pickup_borough.withColumn("rank",
rank().over(window_specification))

    # Filtering the top 5 pickup boroughs for each month
    pickup_boroughs_top_5 =
count_of_pickup_borough.filter(col("rank") <=
5).drop("rank")

    # displaying top 5 pickup boroughs for each month
    pickup_boroughs_top_5.orderBy("Month",
col("trip_count").desc()).show()
```

Output

```
+-------------+-----+----------+
|Pickup_Borough|Month|trip_count|
+-------------+-----+----------+
|     Manhattan|    1|   5854818|
|      Brooklyn|    1|   3360373|
|        Queens|    1|   2589034|
|         Bronx|    1|   1607789|
| Staten Island|    1|    173354|
|     Manhattan|    2|   5808244|
|      Brooklyn|    2|   3283003|
|        Queens|    2|   2447213|
|         Bronx|    2|   1581889|
| Staten Island|    2|    166328|
|     Manhattan|    3|   6194298|
|      Brooklyn|    3|   3632776|
|        Queens|    3|   2757895|
|         Bronx|    3|   1785166|
| Staten Island|    3|    191935|
|     Manhattan|    4|   6002714|
|      Brooklyn|    4|   3481220|
|        Queens|    4|   2666671|
|         Bronx|    4|   1677435|
| Staten Island|    4|    175356|
|     Manhattan|    5|   5965594|
|      Brooklyn|    5|   3586009|
|        Queens|    5|   2826599|
|         Bronx|    5|   1717137|
| Staten Island|    5|    189924|
+-------------+-----+----------+
```

Question: 2
Identifying the top 5 popular dropoff boroughs each month

Implementation of the logic:

```
    # Grouping by Dropoff_Borough and Month, count the
number of trips
    count_of_dropoff_boroughs =
data_join.groupBy("Dropoff_Borough",
month("date").alias("Month")).count().withColumnRenamed("c
ount", "trip_count")

    # Defining a window specification to rank the
boroughs within each month based on trip_count
    window_specification =
Window.partitionBy("Month").orderBy(col("trip_count").desc
())

    # Adding a rank column based on trip_count within
each month
    count_of_dropoff_boroughs =
count_of_dropoff_boroughs.withColumn("rank",
rank().over(window_specification))

    # Filtering the top 5 dropoff boroughs for each month
    dropoff_boroughs_top_5 =
count_of_dropoff_boroughs.filter(col("rank") <=
5).drop("rank")

    # displaying the top 5 dropoff boroughs for each month
    dropoff_boroughs_top_5.orderBy("Month",
col("trip_count").desc()).show(50)
```

Output

```
2024-03-28 12:57:13,581 INFO codege
+---------------+-----+----------+
|Dropoff_Borough|Month|trip_count|
+---------------+-----+----------+
|      Manhattan|    1|   5444345|
|       Brooklyn|    1|   3337415|
|         Queens|    1|   2480080|
|          Bronx|    1|   1525137|
|        Unknown|    1|    535610|
|      Manhattan|    2|   5381696|
|       Brooklyn|    2|   3251795|
|         Queens|    2|   2390783|
|          Bronx|    2|   1511014|
|        Unknown|    2|    497525|
|      Manhattan|    3|   5671301|
|       Brooklyn|    3|   3608960|
|         Queens|    3|   2713748|
|          Bronx|    3|   1706802|
|        Unknown|    3|    566798|
|      Manhattan|    4|   5530417|
|       Brooklyn|    4|   3448225|
|         Queens|    4|   2605086|
|          Bronx|    4|   1596505|
|        Unknown|    4|    551857|
|      Manhattan|    5|   5428986|
|       Brooklyn|    5|   3560322|
|         Queens|    5|   2780011|
|          Bronx|    5|   1639180|
|        Unknown|    5|    578549|
+---------------+-----+----------+
```

Question: 3
Identifying the top 30 earnest routes

Implementation of the logic:

```
    # Selecting relevant columns and calculating
total_profit
    profit =
data_join.groupBy(concat(col("Pickup_Borough_name"),
lit(" to "), col("Dropoff_Borough_name")).alias("Route"))
\
.agg(sum(col("driver_total_pay")).alias("total_profit"))

    # Ordering by total_profit in descending order
    routes_top_30 =
profit.orderBy(col("total_profit").desc()).limit(30)

    # Show the top 30 earnest routes
    routes_top_30.show(50)
```

Output

```
2024-03-28 12:48:27,651 INFO codegen.CodeGene
+--------------------+--------------------+
|               Route|         total_profit|
+--------------------+--------------------+
|Manhattan to Manh...|3.3385772555002284E8|
|Brooklyn to Brooklyn|1.7394472147999206E8|
|    Queens to Queens|1.1470684719998911E8|
| Manhattan to Queens|1.0173842820999998E8|
| Queens to Manhattan| 8.603540026000004E7|
|Manhattan to Unknown| 8.010710241999994E7|
|      Bronx to Bronx| 7.414622575999327E7|
|Manhattan to Broo...|       6.799047559E7|
|Brooklyn to Manha...| 6.317616104999997E7|
|  Brooklyn to Queens| 5.045416243000009E7|
|  Queens to Brooklyn|4.729286536000156E7|
|   Queens to Unknown| 4.629299990000003E7|
|  Bronx to Manhattan| 3.248632517000009E7|
|  Manhattan to Bronx|3.1978763450000055E7|
|    Manhattan to EWR|2.3750888619999994E7|
| Brooklyn to Unknown|       1.084882757E7|
|     Bronx to Unknown|1.0464800209999993E7|
|      Bronx to Queens|1.0292266499999998E7|
|     Queens to Bronx|1.0182898729999993E7|
|Staten Island to ...|   9686862.450000016|
|   Brooklyn to Bronx|          5848822.56|
|   Bronx to Brooklyn|          5629874.41|
|     Brooklyn to EWR|   3292761.710000001|
|Brooklyn to State...|  2417853.8199999994|
|Staten Island to ...|          2265856.46|
|Manhattan to Stat...|  2223727.3700000006|
|Staten Island to ...|  1612227.7199999995|
|       Queens to EWR|          1192758.66|
|Staten Island to ...|   891285.8100000004|
|Queens to Staten ...|           865603.38|
+--------------------+--------------------+
```

Task:4 Average of data

APIs used in this task

- **avg** - average function is provided by pyspark.sql.functions. This function will return us the average (mean) of the values in a DataFrame column.

Detailed Explanation:

- **Question 1:**
  - ➔ Used only selected columns for the simplified process and to reduce memory usage.
  - ➔ To calculate the average 'driver_total_pay' during different 'time_of_day', I am grouping the DataFrame by the "time_of_day" column and calculating the average of the "driver_total_pay" column using **avg()** function and renaming the new column as "average_driver_total_pay".
  - ➔ As a next step, I am sorting the DataFrame based on the newly created column "average_driver_total_pay" in descending order, according to the task.
  - ➔ Then, I am printing the sorted 'average_pay' DataFrame, which will display the average driver total pay for each "time_of_day", in descending order. (Highest value shown first).

- **Question 2:**
  - ➔ Used only selected columns for the simplified process and to reduce memory usage.
  - ➔ To calculate the average 'trip_length' during different 'time_of_day' period, I am grouping the DataFrame by the "time_of_day" column and calculating the average "trip_length" column using **avg()** function and renaming the new column as "average_trip_length".
  - ➔ Next, I am sorting the DataFrame based on the currently created column "average_trip_length" in descending order, as mentioned in the task requirements.
  - ➔ Finally, I am displaying the sorted 'average_trip_length' DataFrame, which will show the acreage trip length for each "time_of_day", in descending order. (Highest value shown first).

- **Question 3:**
  - ➔ Used only selected columns for the simplified process and to reduce memory usage.
  - ➔ For calculating 'average_earned_per_mile' for each "time_of_day" period, i took the last two above results, I joined the two DataFrame ('average_driver_total_pay' and 'average_trip_length') on the "time_of_day" column, by creating a new DataFrame called "average_earning" that will have the average_trip_length and average_driver_total_pay for each time of the day.
  - ➔ I have added a new column called "average_earning_per-mile" to the newly created DataFrame "average_earning", this column has the calculation of dividing

"average_driver-total_pay" / "average_trip_length", which results in the average earnings per mile.

➔ Now I have selected only the required columns, that is "time_of_day" and "average_earning_per_mile" from the 'average_earning' DataFrame and dropped the rest of the columns.

➔ Displaying the results ("average_earning").

- **Question 4:**
  - ➔ Based on the results of trip length and average earnings data, I can infer that the afternoon trips have the maximum 'average_earnings_per_mile', that is indicating that the afternoon time is the most profitable for drivers in case of earnings.
  - ➔ Night trips have the longest 'trip_length', that is indicating that there is a high demand at this time for transportation.
  - ➔ As a driver, I can plan my strategic move by aligning my schedule on afternoon trips to maximize my income by prioritizing my trips during this period.
  - ➔ As a CEO, I will ensure the allocation of resources, namely the availability of drivers and vehicles, are adequately allocated to meet the afternoon trip demands and I will optimize my pricing strategies during the night hours, to maximize the revenue potential.

Challenges faced in Task4:

- I did not face any challenge in this Task.

Knowledge gained in Task4:

- I learnt how to use spark functions to join two DataFrames and to create a new column.
- Now I am able to provide average summaries based on the requirements.

Question:1
Calculating the average 'driver_total_pay'

Implementation of the logic:

```
    # Calculating the average 'driver_total_pay' during
different 'time_of_day' periods
    average_pay =
data_join.groupBy("time_of_day").agg(avg("driver_total_pay
").alias("average_drive_total_pay"))

    # Sorting the results by average 'driver_total_pay'
in descending order
    average_pay =
average_pay.orderBy(col("average_drive_total_pay").desc())

    # displaying the results
    average_pay.show()
```

Output

```
2024-03-28 13:02:09,105 INFO codegen.Co
+-----------+-----------------------+
|time_of_day|average_drive_total_pay|
+-----------+-----------------------+
|  afternoon|     21.212428756593535|
|      night|     20.087438003592712|
|    evening|     19.777427702398384|
|    morning|      19.63333279394483|
+-----------+-----------------------+
```

Question:2
Calculating the average ‘trip length’

Implementation of the logic

```
    # Calculating the average 'trip_length' during
different 'time_of_day' periods
    average_trip_length =
data_join.groupBy("time_of_day").agg(avg("trip_length").al
ias("average_trip_length"))

    # Sorting the results by average 'trip_length' in
descending order
    average_trip_length =
average_trip_length.orderBy(col("average_trip_length").des
c())

    # displaying the results
    average_trip_length.show()
```

Output

```
+-----------+-------------------+
|time_of_day|average_trip_length|
+-----------+-------------------+
|      night|  5.323984801961738|
|    morning|  4.927371866442787|
|  afternoon|  4.861410525661209|
|    evening| 4.4847503674475195|
+-----------+-------------------+
```

Question:3
Calculating the average earned per mile

Implementation of the logic:

```
    # Calculating the average 'trip_length' during
different 'time_of_day' periods
    average_trip_length =
data_join.groupBy("time_of_day").agg(avg("trip_length").al
ias("average_trip_length"))

    # Calculating the average 'driver_total_pay' during
different 'time_of_day' periods
    average_driver_total_pay =
data_join.groupBy("time_of_day").agg(avg("driver_total_pay
").alias("average_driver_total_pay"))

    # Joining the two DataFrames
    average_earning = average_driver_total_pay.join(
        average_trip_length,
        "time_of_day"
    )

    # Calculating the average earning per mile
    average_earning = average_earning.withColumn(
        "average_earning_per_mile",
        col("average_driver_total_pay") /
col("average_trip_length")
    )

    # Selecting only the required columns
    average_earning = average_earning.select(
        "time_of_day",
        "average_earning_per_mile"
    )

    # Show the results
    average_earning.show()
```

Output

```
+-----------+-----------------------+
|time_of_day|average_earning_per_mile|
+-----------+-----------------------+
|  afternoon|      4.363430869420021|
|      night|     3.7730081416068373|
|    morning|     3.9845445657664005|
|    evening|      4.409928330894958|
+-----------+-----------------------+
```

Task:5 Finding anomalies

APIs used in the Task 5:

- **avg** - average function is provided by pyspark.sql.functions. This function will return us the average (mean) of the values in a DataFrame column.
- **dayofmonth** - This function is provided by the pyspark.sql.functions. This will extract the day of the month from the columns that have Timestamp Type or DataType.
- **FloatType** - This data type is provided by the pyspark.sql.types module. It will represent the floating-point numbers, that are numerical values with the decimal points.

Detailed Explanation:

- **Question 1:**
  - ➔ Used only selected columns for the simplified process and to reduce memory usage.
  - ➔ Filtering the January month data in the date column.
  - ➔ Created a new DataFrame that extracts the day of the month from the “date” column.
  - ➔ For performing numerical operations, the “request_to_pickup” column is converted from default type to FloatType.
  - ➔ Calculated the average of the “request_to_pickup” column using **avg()** function. **orderBy()** will order the results by the “day” column, and will rename the resulting column to “average_waiting_time”.
  - ➔ Repartitioning the resulting Data Frame into a single file and writing it as a csv file for the plotting.
  - ➔ I used pandas and matplotlib libraries to plot the results as a histogram.

- **Question 2:**
  - ➔ The average waiting time exceeds 300 seconds on Day 1.

- **Question 3:**
  - ➔ The average waiting time was longer on this day compared to other days, maybe because it is the new year day or due to New Year’s celebration.

Challenges faced in Task 5:

- I did not face much challenge in this task, figuring out the logic took some time.

Knowledge gained in Task 5:

- I got to detect the anomalies among the dataset using spark functions and to visualize it.

Question1:

Implementation of the logic:

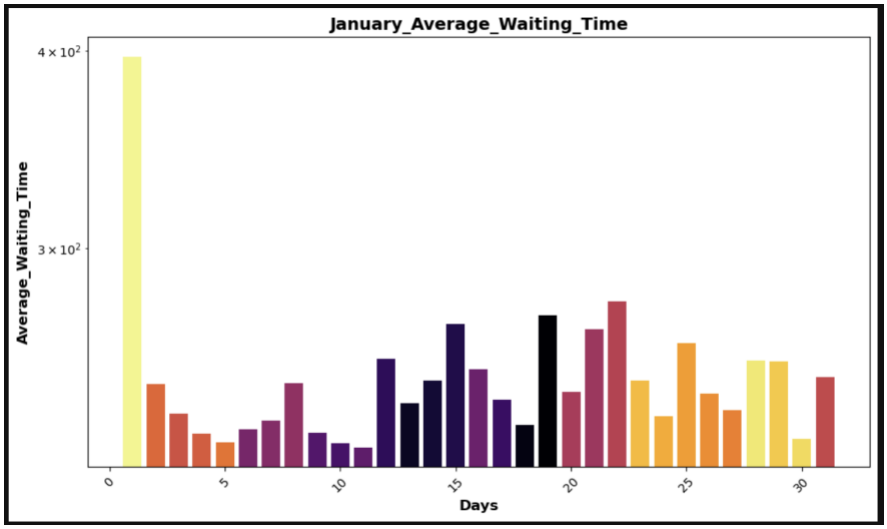
```
    # Converting the UNIX timestamp to "yyyy-MM-dd"
format in the date column
    rideshare_data = rideshare_data.withColumn("date",
from_unixtime(col("date"), "yyyy-MM-dd"))

    # Filtering January data
    january_data =
rideshare_data.filter(col("date").startswith("2023-01"))

    # Extracting day from date
    january_data = january_data.withColumn("day",
dayofmonth("date"))

    # Converting 'request_to_pickup' column to FloatType
    january_data =
january_data.withColumn("request_to_pickup",
january_data["request_to_pickup"].cast(FloatType()))

    # Calculating average waiting time for each day
    average_waiting_time =
january_data.groupBy("day").avg("request_to_pickup").order
By("day").withColumnRenamed("avg(request_to_pickup)",
"average_waiting_time")
    average_waiting_time.show(20)

    # Repartitioning the DataFrame to avoid writing
multiple small files
    average_waiting_time =
average_waiting_time.repartition(1)

    # Writing to CSV
    output_path_file = "s3a://" +
s3_data_repository_bucket +
"/ECS765/rideshare_2023/january_average_waiting_time.csv"
    average_waiting_time.write.csv(output_path_file,
mode="overwrite", header=True)
```

Output

```
+---+--------------------+
|day|average_waiting_time|
+---+--------------------+
|  1|   396.5318744409635|
|  2|  246.05148716456986|
|  3|  235.68026834234155|
|  4|  228.85434668408274|
|  5|  226.08877381422872|
|  6|  230.35306927438575|
|  7|  233.25699185710533|
|  8|  246.41358687741243|
|  9|    229.265944341545|
| 10|  225.65276195086662|
| 11|  224.40468798627612|
| 12|  255.17599322195403|
| 13|  239.22308233638282|
| 14|  247.49345781069232|
| 15|   268.5346481777792|
| 16|  251.55102299494047|
| 17|   240.5772885527869|
| 18|  231.90770494488552|
| 19|  272.02203820618143|
| 20|  243.43761253646377|
+---+--------------------+
only showing top 20 rows
```

```
jovyan@jupyter-ec23543:~/teaching_material/ECS765P/coursework$ ccc method bucket ls //data-repository-bkt/ECS765/rideshare
_2023/
                           PRE average_waiting_time/
                           PRE average_waiting_times/
                           PRE avg_waiting_time/
                           PRE january_average_waiting_time.csv/
                           PRE total_trip_count_part_1/
                           PRE total_trip_count_part_2/
2024-02-02 12:22:50 6119240191 rideshare_data.csv
2024-03-06 10:48:21     260375 sample_data.csv
2024-02-02 12:24:26      12322 taxi_zone_lookup.csv
jovyan@jupyter-ec23543:~/teaching_material/ECS765P/coursework$ ccc method bucket ls //data-repository-bkt/ECS765/rideshare
_2023/january_average_waiting_time.csv/
2024-03-31 10:36:45          0 _SUCCESS
2024-03-31 10:36:45        689 part-00000-1e2e9492-7e97-4d75-b9b7-4c5b97856aef-c000.csv
jovyan@jupyter-ec23543:~/teaching_material/ECS765P/coursework$ ccc method bucket cp bkt://data-repository-bkt/ECS765/r
hare_2023/january_average_waiting_time.csv/part-00000-1e2e9492-7e97-4d75-b9b7-4c5b97856aef-c000.csv ./
download: s3://data-repository-bkt/ECS765/rideshare_2023/january_average_waiting_time.csv/part-00000-1e2e9492-7e97-4d75-b9
b7-4c5b97856aef-c000.csv to ./part-00000-1e2e9492-7e97-4d75-b9b7-4c5b97856aef-c000.csv
```

Histogram



Question:2
Finding the average waiting time that exceeds 300 seconds

Implementation of the logic:

```python
import pandas as pd

# loading the data
dataframe = pd.read_csv('average_waiting_time.csv')

# filtering data
waiting_time_exceeds_300 = dataframe[dataframe['average_waiting_time'] > 300]

if not waiting_time_exceeds_300.empty:
    print("Days_Where_Average_Waiting_Time_Exceeds_300_Seconds")
    print(waiting_time_exceeds_300)

    # Calculate and display basic statistics
    mean_of_time = waiting_time_exceeds_300['average_waiting_time'].mean()
    max_of_time = waiting_time_exceeds_300['average_waiting_time'].max()
    min_of_time = waiting_time_exceeds_300['average_waiting_time'].min()

    print("Summary_of_Average_waiting_Time_Data")
    print(f"Waiting_Time_Mean: {mean_time:.2f} seconds")
    print(f"Waiting_Time_Maximum: {max_time} seconds")
    print(f"Waiting_Time_Minimum: {min_time} seconds")
else:
    print("None of the days exceeded 300 seconds average waiting time")
```

Output

```
Days_Where_Average_Waiting_Time_Exceeds_300_Seconds
    day  average_waiting_time
30    1            396.531874
Summary_of_Average_waiting_Time_Data
Waiting_Time_Mean: 396.53 seconds
Waiting_Time_Maximum: 396.5318744409635 seconds
Waiting_Time_Minimum: 396.5318744409635 seconds
```

Task:6 Filtering data

APIs used in the Task 6:

- **lit** - This function is provided by the **Spark SQL functions**. This function is used to create a new column with a constant value.
- **count:** This function is provided by **pyspark.sql.functions** and it is an aggregate function. It will count all the non-null values in the DataFrame.

Detailed Explanation:

- **Question 1:**
  - ➔ Used only selected columns for the simplified process and to reduce memory usage.
  - ➔ Grouped the DataFrame by the “pickup_borough” and “time_of_day” columns using **groupBy()** functions, and counted the number of rows for all combination of boroughs and time_of_day, and renamed the **“count”** column to **“trip_count”**.
  - ➔ Filtered the “count_of_pickup_borough” DataFrame using **filter()** function, to include only the rows where the “trip_count” is greater than 0 and lesser than 1000. This is done to remove anomalies in the data.

- ➔ Used **orderBy()** function, to order the DataFrame by “pickup_borough” and “time_of_day” columns.
- ➔ Displayed the result.

- **Question 2:**
  - ➔ Used only selected columns for the simplified process and to reduce memory usage.
  - ➔ Grouped the DataFrame by “pickup_borough” and “time_of_day” and counted the number of trips.
  - ➔ Filtered trips count only for the “evening” time.
  - ➔ Printed the resulting DataFrame.

- **Question 3:**
  - ➔ Used only selected columns for the simplified process and to reduce memory usage.
  - ➔ Filtered the DataFrame where the “pickup_borough” is Brooklyn and “dropoff_borough” is Staten Island.
  - ➔ Calculated the total number of trips between these two boroughs by counting the rows.
  - ➔ Printed the resulting DataFrame.

Challenges faced in Task 6:

- I did not face any challenge during this task.

Knowledge gained in Task 6:

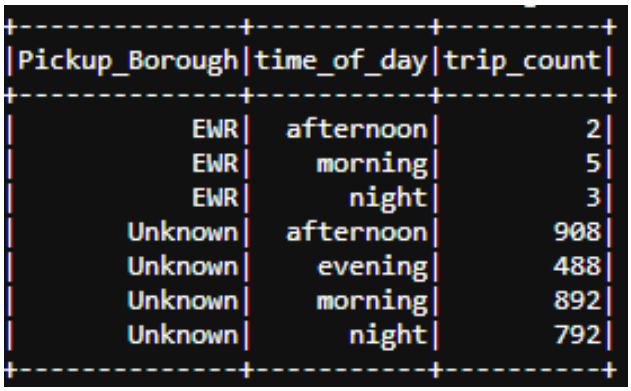
- I learned how to filter the DataFrame using one or more conditions according to the requirements.

Question:1
Finding trip counts greater than 0 and less than 1000

Implementation of the logic

```
    # Grouping by Pickup_Borough, time_of_day, and count
the number of trips
    count_of_pickup_borough =
data_join.groupBy("Pickup_Borough",
"time_of_day").count().withColumnRenamed("count",
"trip_count")

    # Filtering trip counts greater than 0 and less than
1000
    data_filter =
count_of_pickup_borough.filter((col("trip_count") > 0) &
(col("trip_count") < 1000))

    # Displaying the result
    data_filter.orderBy("Pickup_Borough",
"time_of_day").show()
```

Output



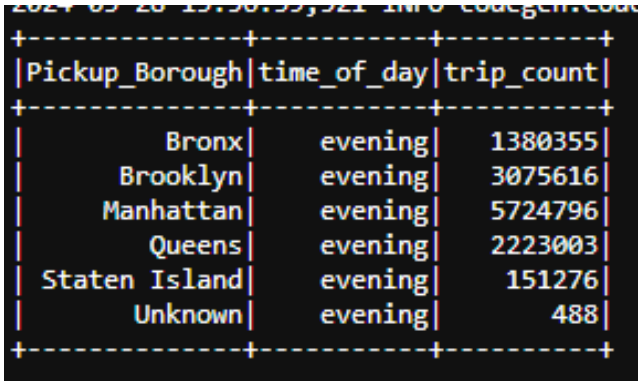
Pickup_Borough	time_of_day	trip_count
EWR	afternoon	2
EWR	morning	5
EWR	night	3
Unknown	afternoon	908
Unknown	evening	488
Unknown	morning	892
Unknown	night	792

Question:2

Implementation of the logic:

```
    # Grouping by Pickup_Borough and time_of_day, and
count the number of trips
    count_of_pickup_borough =
data_join.groupBy("Pickup_Borough",
"time_of_day").count().withColumnRenamed("count",
"trip_count")

    # Filtering trip counts only for 'evening' time
    count_of_evening_pickup =
count_of_pickup_borough.filter(col("time_of_day") ==
"evening")

    # Displaying the result

count_of_evening_pickup.orderBy("Pickup_Borough").show()
```

Output



Pickup_Borough	time_of_day	trip_count
Bronx	evening	1380355
Brooklyn	evening	3075616
Manhattan	evening	5724796
Queens	evening	2223003
Staten Island	evening	151276
Unknown	evening	488

Question: 3
Calculating the number of trips that started in Brooklyn and ended in staten island

Implementation of the logic

```
    # Filtering data for trips from Brooklyn to Staten
Island
    brooklyn_staten_trips =
data_join.filter((col("Pickup_Borough") == "Brooklyn") &
(col("Dropoff_Borough") == "Staten Island"))

    # Selecting required columns
    new_data =
brooklyn_staten_trips.select("Pickup_Borough",
"Dropoff_Borough", "Pickup_Zone")

    # Displaying 10 samples without truncating Pickup_Zone
    new_data.show(10, truncate=False)

    # Calculating the number of trips
    number_of_trips = brooklyn_staten_trips.count()
    print("Number of trips from Brooklyn to Staten
Island:", number_of_trips)
```

Output

```
2024-03-28 16:41:12,004 INFO scheduler.DAGScheduler: Job 8 finished: count at NativeMethodAccessorImpl.java:0, took 2012.363311 s
Number of trips from Brooklyn to Staten Island: 69437
```

```
+--------------+---------------+-------------------------+
|Pickup_Borough|Dropoff_Borough|Pickup_Zone              |
+--------------+---------------+-------------------------+
|Brooklyn      |Staten Island  |DUMBO/Vinegar Hill       |
|Brooklyn      |Staten Island  |Dyker Heights            |
|Brooklyn      |Staten Island  |Bensonhurst East         |
|Brooklyn      |Staten Island  |Williamsburg (South Side)|
|Brooklyn      |Staten Island  |Bay Ridge                |
|Brooklyn      |Staten Island  |Bay Ridge                |
|Brooklyn      |Staten Island  |Flatbush/Ditmas Park     |
|Brooklyn      |Staten Island  |Bay Ridge                |
|Brooklyn      |Staten Island  |Bath Beach               |
|Brooklyn      |Staten Island  |Bay Ridge                |
+--------------+---------------+-------------------------+
only showing top 10 rows
```

Task:7 Routes analysis

APIs used in the Task 7:

- Lit - This function is provided by the **Spark SQL functions**. This function is used to create a new column with a constant value.
- Count: This function is provided by **pyspark.sql.functions** and it is an aggregate function. It will count all the non-null values in the DataFrame.

Detailed Explanation:

- ➔ Used only selected columns for the simplified process and to reduce memory usage.
- ➔ Created a new column named “route” by concatenating the columns “pickup_zone” and “drop_off_zone” with a literal string “to” in between.
- ➔ Calculated the “total_count” of trips for each route from both Lyft and Uber.
- ➔ Pivoted the data for separating Uber and Lyft.
- ➔ Calculated the total count for each route.
- ➔ Reordered the columns in the DataFrame to display the columns in certain order.
- ➔ Got the top ten routes by the “total_count”.
- ➔ Printed the resulting DataFrame.

Challenges faced in Task 7:

- I faced a challenge in using pivot for creating columns for Lyft and Uber. I thought of finding a method to create columns for both the businesses without additional steps. So I was supposed to analyze what pivots do here.

Knowledge gained in Task 7:

- I learnt how to use pivot for creating columns.

Question:1

Implementation of the logic

```
    # Creating a new column 'Route' by concatenating
Pickup_Zone and Dropoff_Zone
    data_join = data_join.withColumn("Route",
concat(col("Pickup_Zone"), lit(" to "),
col("Dropoff_Zone")))

    # Calculating the total count of trips for each
unique route from both Uber and Lyft
    count_of_routes = data_join.groupBy("Route",
"business").count()

    # Pivoting the data to separate Uber and Lyft counts
    counts_pivot =
count_of_routes.groupBy("Route").pivot("business").sum("co
unt")

    # Calculating the total count for each route
    total_routes_count =
counts_pivot.withColumnRenamed("Uber", "uber_count") \

.withColumnRenamed("Lyft", "lyft_count") \

.withColumn("total_count",

col("uber_count").cast("int") +
col("lyft_count").cast("int"))

    # Reordering the columns
    total_routes_count =
total_routes_count.select("Route", "uber_count",
"lyft_count", "total_count")

    # Getting the top 10 routes by total count
    top_10_routes =
total_routes_count.orderBy(col("total_count").desc()).limi
t(10)

    # Displaying the result
    top_10_routes.show()
```

Output

```
+--------------------+----------+----------+-----------+
|               Route|uber_count|lyft_count|total_count|
+--------------------+----------+----------+-----------+
|   JFK Airport to NA|    253211|        46|     253257|
|East New York to ...|    202719|       184|     202903|
|Borough Park to B...|    155803|        78|     155881|
|LaGuardia Airport...|    151521|        41|     151562|
|Canarsie to Canarsie|    126253|        26|     126279|
|South Ozone Park ...|    107392|      1770|     109162|
|Crown Heights Nor...|     98591|       100|      98691|
|Bay Ridge to Bay ...|     98274|       300|      98574|
|  Astoria to Astoria|     90692|        75|      90767|
|Jackson Heights t...|     89652|        19|      89671|
+--------------------+----------+----------+-----------+
```

Task:8 Graph Processing

APIs used in the Task 8:

- **StringType** - This class is provided by pyspark.sql.types. It will be useful for specifying a column datatype as string.
- **StructType**: This function is provided by the pyspark.sql.types. This function will allow us to define the overall structure of a DataFrame.
- **StructField**: This function is provided by the pyspark.sql.types. This function is used to specify the individual columns and their properties such as data type and name.
- **Graph**: This function is provided by the graphframes. This function is used for graph representation, manipulation that are made from DataFrames.
- **SQLContext**: This function is provided by the pyspark.sll.functions. This function allows us to work with structured data and execute SQL queries in spark.

Detailed Explanation:

➔ Defined the schema for both edges and vertices in the graph.
➔ Created DataFrames for both vertices and edges.
➔ Using vertices and edges DataFrames, created a graph.
➔ Found the vertices in the graph where the “pickup” and “dropoff” locations lie in the same borough and same service zone and printed the count.
➔ Selected the columns specifically from the vertices and printed 10 rows of data.

Challenges faced in Task 8:

- I did not face much challenge in this task except processing the graph.

Knowledge gained in Task 8:

- I learnt how to create graphs using spark functions and how to traverse it.

Graph Processing
Question:1 and 2

```
    # Creating edges dataframe
    dataframe_edges = data_join.select("pickup_location",
"dropoff_location").withColumnRenamed("pickup_location",
"src").withColumnRenamed("dropoff_location", "dst")

    # Displaying 10 samples of edges dataframe
    print("Edges Dataframe:")
    dataframe_edges.show(10, truncate=False)

    # Displaying 10 samples of vertices dataframe
    print("Vertices Dataframe:")
    dataframe_vertices.show(10, truncate=False)
```

Output

```
+---+-------------+-----------------------+------------+
|id |Borough      |Zone                   |service_zone|
+---+-------------+-----------------------+------------+
|1  |EWR          |Newark Airport         |EWR         |
|2  |Queens       |Jamaica Bay            |Boro Zone   |
|3  |Bronx        |Allerton/Pelham Gardens|Boro Zone   |
|4  |Manhattan    |Alphabet City          |Yellow Zone |
|5  |Staten Island|Arden Heights          |Boro Zone   |
|6  |Staten Island|Arrochar/Fort Wadsworth|Boro Zone   |
|7  |Queens       |Astoria                |Boro Zone   |
|8  |Queens       |Astoria Park           |Boro Zone   |
|9  |Queens       |Auburndale             |Boro Zone   |
|10 |Queens       |Baisley Park           |Boro Zone   |
+---+-------------+-----------------------+------------+
only showing top 10 rows
```

```
in 14.260464 ms
+---+---+
|src|dst|
+---+---+
|151|244|
|244|78 |
|151|138|
|138|151|
|36 |129|
|138|88 |
|200|138|
|182|242|
|248|242|
|242|20 |
+---+---+
only showing top 10 rows
```

Question: 3

Implementation of the logic

```
    # Loading vertices and edges data
    data_vertices = spark.read.csv("s3a://" +
s3_data_repository_bucket +
"/ECS765/rideshare_2023/taxi_zone_lookup.csv",
header=True)
    data_edges = spark.read.csv("s3a://" +
s3_data_repository_bucket +
"/ECS765/rideshare_2023/rideshare_data.csv", header=True,
inferSchema=True)

    # Creating vertices DataFrame with required columns
    data_vertices = data_vertices.selectExpr("LocationID
as id", "Borough", "Zone", "service_zone")

    # Creating edges DataFrame with required columns
    data_edges = data_edges.selectExpr("pickup_location
as src", "dropoff_location as dst")

    # Creating GraphFrame
    dataframe_graph = GraphFrame(data_vertices,
data_edges)

    # Printing 10 samples of the graph DataFrame with
columns 'src', 'edge', and 'dst'
    edges_of_graph = dataframe_graph.find("(src)-[edge]->
(dst)").select("src", "edge", "dst")
    print("Graph DataFrame (Edges):")
    edges_of_graph.show(10, truncate=False)
```

Output

```
+--------------------------------------------------------+---------+-----------------------------------------------------+
|src                                                     |edge     |dst                                                  |
+--------------------------------------------------------+---------+-----------------------------------------------------+
|[151, Manhattan, Manhattan Valley, Yellow Zone]         |[151, 244]|[244, Manhattan, Washington Heights South, Boro Zone]|
|[244, Manhattan, Washington Heights South, Boro Zone]   |[244, 78] |[78, Bronx, East Tremont, Boro Zone]                 |
|[151, Manhattan, Manhattan Valley, Yellow Zone]         |[151, 138]|[138, Queens, LaGuardia Airport, Airports]           |
|[138, Queens, LaGuardia Airport, Airports]              |[138, 151]|[151, Manhattan, Manhattan Valley, Yellow Zone]      |
|[36, Brooklyn, Bushwick North, Boro Zone]               |[36, 129] |[129, Queens, Jackson Heights, Boro Zone]            |
|[138, Queens, LaGuardia Airport, Airports]              |[138, 88] |[88, Manhattan, Financial District South, Yellow Zone]|
|[200, Bronx, Riverdale/North Riverdale/Fieldston, Boro Zone]|[200, 138]|[138, Queens, LaGuardia Airport, Airports]       |
|[182, Bronx, Parkchester, Boro Zone]                    |[182, 242]|[242, Bronx, Van Nest/Morris Park, Boro Zone]        |
|[248, Bronx, West Farms/Bronx River, Boro Zone]         |[248, 242]|[242, Bronx, Van Nest/Morris Park, Boro Zone]        |
|[242, Bronx, Van Nest/Morris Park, Boro Zone]           |[242, 20] |[20, Bronx, Belmont, Boro Zone]                      |
+--------------------------------------------------------+---------+-----------------------------------------------------+
only showing top 10 rows
```

Question:4

Implementation of the logic

```
    # Load vertices and edges data
    data_vertices = spark.read.csv("s3a://" +
s3_data_repository_bucket +
"/ECS765/rideshare_2023/taxi_zone_lookup.csv",
header=True)
    data_edges = spark.read.csv("s3a://" +
s3_data_repository_bucket +
"/ECS765/rideshare_2023/rideshare_data.csv", header=True,
inferSchema=True)

    # Creating vertices DataFrame with required columns
    data_vertices = data_vertices.selectExpr("LocationID
as id", "Borough", "Zone", "service_zone")

    # Creating edges DataFrame with required columns
    data_edges = data_edges.selectExpr("pickup_location
as src", "dropoff_location as dst")

    # Creating GraphFrame
    dataframe_graph = GraphFrame(data_vertices,
data_edges)

    # Counting connected vertices with the same Borough
and service_zone
    vertices_connected = dataframe_graph.find("(src)-[]->
(dst)") \
        .filter("src.Borough == dst.Borough AND
src.service_zone == dst.service_zone") \
        .select("src.id", "dst.id", "src.Borough",
"src.service_zone")

    # Selecting 10 samples
    output = vertices_connected.limit(10)

    # Displaying the result
    print("Sample of Connected Vertices with Same Borough
and Service Zone:")
    output.show(truncate=False)
```

Output

id	id	Borough	service_zone
182	242	Bronx	Boro Zone
248	242	Bronx	Boro Zone
242	20	Bronx	Boro Zone
20	20	Bronx	Boro Zone
236	262	Manhattan	Yellow Zone
262	170	Manhattan	Yellow Zone
239	239	Manhattan	Yellow Zone
94	69	Bronx	Boro Zone
47	247	Bronx	Boro Zone
76	76	Brooklyn	Boro Zone

Question:5

```
   # Loading vertices and edges data
    data_vertices = spark.read.csv("s3a://" +
s3_data_repository_bucket +
"/ECS765/rideshare_2023/taxi_zone_lookup.csv",
header=True)
    data_edges = spark.read.csv("s3a://" +
s3_data_repository_bucket +
"/ECS765/rideshare_2023/rideshare_data.csv", header=True,
inferSchema=True)

    # Creating vertices DataFrame with required columns
    data_vertices = data_vertices.selectExpr("LocationID
as id", "Borough", "Zone", "service_zone")

    # Creating edges DataFrame with required columns
    data_edges = data_edges.selectExpr("pickup_location
as src", "dropoff_location as dst")

    # Creating GraphFrame
    dataframe_graph = GraphFrame(data_vertices,
data_edges)

    # Performing page ranking
    page_ranking =
dataframe_graph.pageRank(resetProbability=0.17, tol=0.01)

    # Getting top 5 samples of page rank results
    top_5_page_rank =
page_ranking.vertices.orderBy("pagerank",
ascending=False).limit(5)

    # Showing the top 5 samples
    print("Top 5 samples of PageRank results:")
    top_5_page_rank.show(truncate=False)
```

Output

```
+---+-------+---------------+------------+------------------+
|id |Borough|Zone           |service_zone|pagerank          |
+---+-------+---------------+------------+------------------+
|1  |EWR    |Newark Airport |EWR         |7.145955745404148 |
|130|Queens |Jamaica        |Boro Zone   |3.179586478748835 |
|95 |Queens |Forest Hills   |Boro Zone   |2.801515055722621 |
|82 |Queens |Elmhurst       |Boro Zone   |2.7909647586062567|
|129|Queens |Jackson Heights|Boro Zone   |2.7422638007398654|
+---+-------+---------------+------------+------------------+
```

```
2024-03-23 12:03:41,818 INFO codegen.CodeGenerator: Code generated in 5.635627 ms
+---+------------------+
|id |pagerank          |
+---+------------------+
|1  |7.1459557454041445|
|130|3.179586478748832 |
|95 |2.80151505572262  |
|82 |2.790964758606256 |
|129|2.7422638007398636|
+---+------------------+
```

```
2024-04-02 08:36:40,544 INFO scheduler.DAGScheduler: Job 8 finished: count at NativeMethodAccessorImpl.java:0, took 202.515731 s
Number of Connected Vertices with Same Borough and Service Zone: 46886992
```