ECS765P - Big Data Processing

Coursework Report: NYC Rideshare Analysis

Table of Contents

Task 1 3
APIs used: 3
Steps: 3
Output 4
Challenges encountered in this task: 4
Knowledge / Insight attained: 5
Task 2 5
APIs used: 5
Steps: 5
Output 6
Challenges encountered in this task 8
Knowledge / Insight attained 8
Task 3 8
APIs used: 8
Steps: 9
Output 10
Challenges encountered in this task 13
Knowledge / Insight attained 13
Task 4 13
APIs used: 13
Steps: 13
Output 14
Challenges encountered in this task 15
Knowledge / Insight attained 15
Task 5 15
APIs used: 15
Steps: 15

Output 16
Challenges encountered in this task 17
Knowledge / Insight attained 17
Task 6 17
APIs used: 17
Steps: 17
Output 18
Challenges encountered in this task 19
Knowledge / Insight attained 19
Task 7 19
APIs used: 19
Steps: 20
Output 20
Challenges encountered in this task 21
Knowledge / Insight attained 21
Task 8 21
References 23

Task 1

APIs used:

1. **pyspark.sql.SparkSession:** This PySpark API is used to create Spark sessions. It provides an entry point to work with structured data (DataFrames) in Spark SQL.
2. **pyspark.sql.functions:** This API provides several built-in standard functions to work with DataFrame and SQL queries. We use functions like from_unixtime and date_format in this task.

Steps:

1. **Loading the Datasets:**

   spark.read.option("header",True).csv(): This method is used to load the 'rideshare_data.csv' and 'taxi_zone_lookup.csv' CSV files into DataFrames and the first row is set as column names by specifying option("header",True).

2. **Performing the First Join Operation:**

   join(): This function is used to perform the join operation on the two DataFrames: 'rideshare_data' and 'taxi_zone_lookup_df' based on 'pickup_Location' and 'LocationID' as common key.

   withColumnRenamed(): This function is used to rename the columns.

   drop(): This function is used to drop the 'LocationID' column after the join operation.

3. **Performing the Second Join Operation:**

   Similar to the first join, we perform another join operation between the DataFrame obtained from the first join and taxi_zone_lookup_df followed by renaming the added columns and dropping 'LocationID'.

4. **Converting the UNIX Timestamp to "yyyy-MM-dd" format:**

   withColumn(): This function is used to perform operations on a column such as adding new columns, changing values of existing columns, changing datatypes of columns (changing the datatype of the "date" column in this case). It returns a new spark DataFrame

   from_unixtime(): This function converts a Unix timestamp (number of seconds since January 1, 1970) to a string representing timestamp of that moment.

   date_format(): This function is used to convert the date to a string format

5. **Printing First 10 Rows of the Merged DataFrame:**

   show(n): This function is used to display the first n rows of the DataFrame. We use it to display first 10 rows.

6. **Printing Number of Rows and Schema of the Merged DataFrame:**

   count(): This function returns the number of rows in the DataFrame.

   printSchema(): This function prints the schema of the DataFrame, which includes the data types and nullable properties of each column.

Output

Screenshot of No. of rows and Schema

```
Number of Rows: 69725864
Schema:
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)
```

Challenges encountered in this task:

- Performing the join operation twice with proper syntax and renaming the added columns

- Converting the ‘date’ column from UNIX timestamp to the string “yyyy-MM-dd” format
- Waiting for several minutes for the spark job to execute involving multiple partitions.

I overcame these challenges by looking up Spark documentation and tutorials to figure out the right API to use in each step with some trial and error of different functions and methods.

Knowledge / Insight attained:

- Merging DataFrames with different types of join operations to allow efficient analysis of tables.
- Different timestamp conversion formats.
- Applying transformations to columns such as changing data types, and operations like renaming and dropping.
- Analysing the dataset Schema

Task 2

APIs used:

1. **pyspark.sql.SparkSession**
2. **pyspark.sql.functions:** In this task, we use the following built-in functions of this API: col(), sum()

Steps:

1. **Loading the Datasets:**

   spark.read.option("header",True).csv(): This method is used to load the ‘rideshare_data.csv’ file into DataFrame and the first row is set as column names by specifying option(“header”,True).

2. **Converting the UNIX Timestamp to "yyyy-MM" format**
3. **Getting aggregated DataFrame of trips count of each business in each month**

   groupby(): This function is used to group the DataFrame based on one or more attributes and perform aggregated functions on it such as count in this case

   coalesce(): This function is used to specify number of output partitions (1 in this case)

   write.option("header", "true").csv(): This function writes the DataFrame to a CSV file in the bucket and assigns the first row as column header

4. **Getting aggregated DataFrame of Platform Profit of each business in each month**

   withColumn(): To change datatype of rideshare_profit column

   Column.cast(): Function to change the column type from string data type to float

   groupby().agg(): Function to calculate one or more aggregates

pyspark.sql.functions.sum(): Returns the sum of values in each group

column.alias(): Function to rename the column

coalesce(): specify number of output partitions (1 in this case)

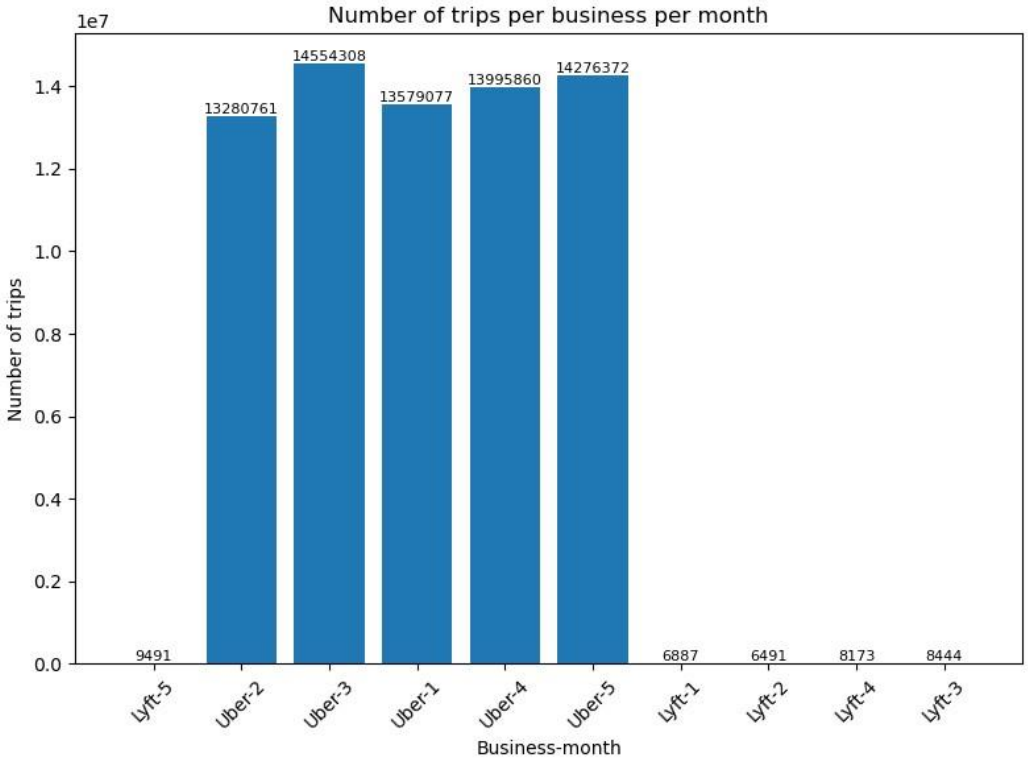
write.option("header", "true").csv(): write the DataFrame to a CSV file in the bucket and assign the first row as column header

5. **Getting aggregated DataFrame of Driver Total Pay of each business in each month**

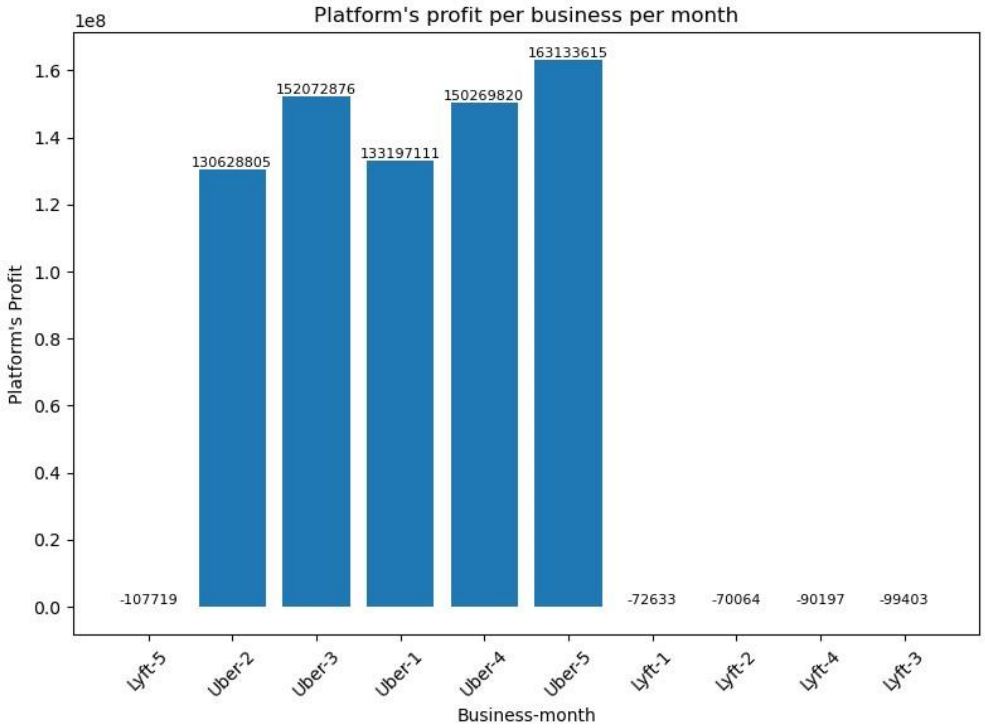
   The functions used for this sub-task are same as sub-task 4

Output

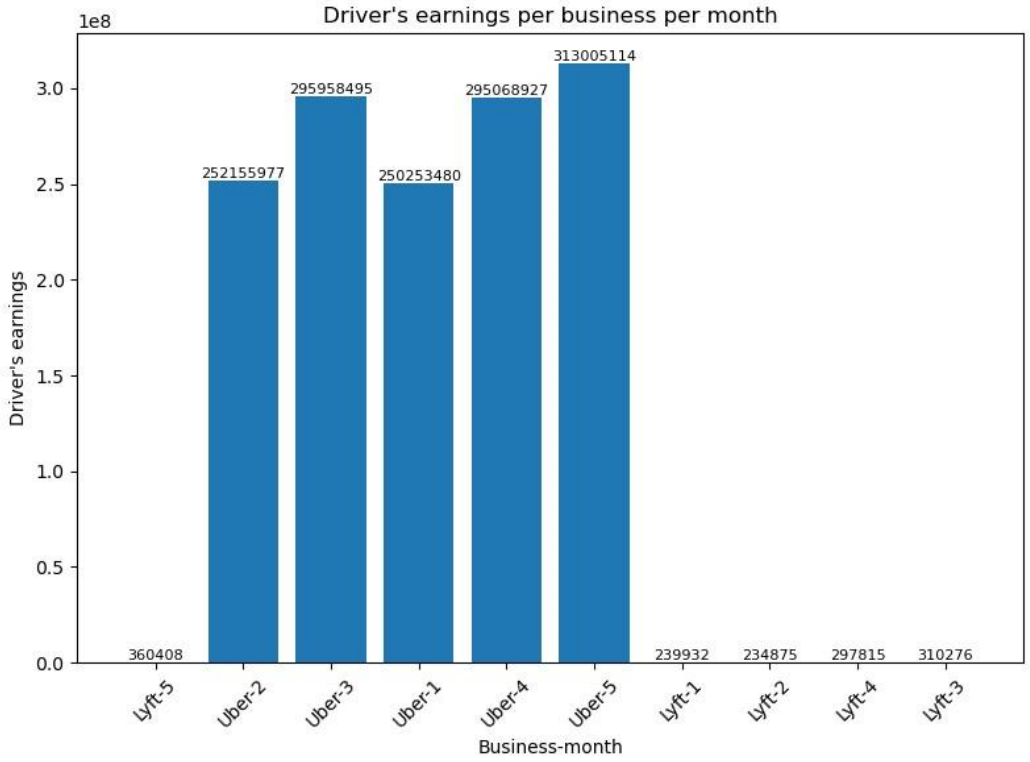
1) Histogram of number of trips for each business in each month



2) Histogram of platform's profits for each business in each month



3) Histogram of driver's earnings for each business in each month



4) **Insights extracted from the above results:**

   It is inferred from the above results that Uber makes significantly higher earnings than Lyft in any given month. Uber is a much more renowned and used service than Lyft, as evident from the number of monthly trips illustrated by the first histogram. The next 2 results are a direct outcome of this. Given this, if I were a stockbroker, I would recommend clients to invest more heavily in Uber. If I were a CEO of the business, I would build strategies to optimize revenues targeting those sources which maximize revenue. I would also invest in schemes that would benefit the drivers, which would attract more drivers.

Challenges encountered in this task

- Discovering how to use groupby and aggregate functions
- A major challenge was to write the DataFrame to a single CSV file in the bucket. Initially, I forgot to include coalesce() due to which multiple partitions were being generated, one for each row.
- Another challenge I faced was to convert the datatype of numeric columns from string to floating points.
- Spark job execution took several minutes.

I overcame most of the challenges by looking up spark documentations and tutorials on the internet. I also tried different syntax combinations to use aggregate functions with alias() to rename the columns appropriately after performing aggregate. Finally, I overcame the lengthy execution time by using dynamic number of executors to run the spark job using the command ‘ccc create spark task2.py -d -s’

Knowledge / Insight attained

- Learned how to use the ‘groupby’ function with different aggregate functions to perform aggregate operations such as ‘count’ and ‘sum’ on the DataFrame.
- Learned how to merge all output file partitions to one using ‘coalesce’ function
- Learned how to convert a column datatype using the ‘cast’ function of class ‘Column’
- Using the ‘alias’ function with aggregate functions to rename columns

Task 3

APIs used:

1. **pyspark.sql.SparkSession**
2. **pyspark.sql.window:** This API introduces the Window class to create window frames/partitions of rows on which window functions like rank() operate. It uses Window.partitionBy() to create a WindowSpec
3. **pyspark.sql.functions:** In this task, we use the following new built-in functions of this API: rank(), concat(), lit()

Steps:

1. **Loading the Datasets**
2. **Joining the datasets (Refer to task 1)**
3. **Converting the UNIX Timestamp to "MM" format and casting datatype to integer**
4. **Identifying the top 5 popular pickup boroughs each month**

   groupby().count(): Grouping DataFrame on Month and Pickup_Borough and getting aggregated count on each group

   Window.partitionBy(): This function of the Window class creates a WindowSpec with the defined partition

   rank(): This is a window function that returns the rank of rows within a window partition. Ranking is done to sort rows in a specified order

   filter(): Function to filter rows based on given condition. Here we filter rows of rank 1-5

   orderBy(): Function to sort the DataFrame on given columns

5. **Identifying the top 5 popular dropoff boroughs each month**

   Same functions as step 4

6. **Identifying top 30 earnest routes**

   concat(): Function to concatenate 2 or more columns. Here we add a new column ‘Route’ by concatenating Pickup_Borough and Dropoff_Borough columns

   lit(): Function to add a constant value to a column. Here we add the literal “to” while concatenating ‘Pickup_Borough’ and ‘Dropoff_Borough’ columns

   groupby().agg(): calculate one or more aggregates

   sum(): calculate sum of values in each group

   sort(): Function to sort DataFrame on given column in either ascending or descending order

   limit(n): DataFrame transformation to return top n rows (30 in this case)

Output

1) Table of top 5 popular pickup boroughs each month

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

2) Table of top 5 popular dropoff boroughs each month

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

3) Table of top 30 earnest routes

Route	total_profit
Manhattan to Manhattan	3.3385772555002296E8
Brooklyn to Brooklyn	1.7394472147999212E8
Queens to Queens	1.1470684719998913E8
Manhattan to Queens	1.0173842820999996E8
Queens to Manhattan	8.603540026000005E7
Manhattan to Unknown	8.0107102419999967
Bronx to Bronx	7.414622575999323E7
Manhattan to Brooklyn	6.799047559E7
Brooklyn to Manhattan	6.3176161049999975E7
Brooklyn to Queens	5.045416243000008E7
Queens to Brooklyn	4.729286536000015E7
Queens to Unknown	4.629299990000005E7
Bronx to Manhattan	3.248632517000008E7
Manhattan to Bronx	3.197876345000006E7
Manhattan to EWR	2.3750888619999986E7
Brooklyn to Unknown	1.0848827569999998E7
Bronx to Unknown	1.0464800209999992E7
Bronx to Queens	1.0292266499999996E7
Queens to Bronx	1.0182898729999997E7
Staten Island to Staten Island	9686862.450000014
Brooklyn to Bronx	5848822.559999999
Bronx to Brooklyn	5629874.41
Brooklyn to EWR	3292761.710000001
Brooklyn to Staten Island	2417853.82
Staten Island to Brooklyn	2265856.46
Manhattan to Staten Island	2223727.37
Staten Island to Manhattan	1612227.72
Queens to EWR	1192758.66
Staten Island to Unknown	891285.8100000002
Queens to Staten Island	865603.3800000001

4) Insights extracted from above results

Manhattan is the most popular borough accounting for the highest number of trips in all months, followed by Brooklyn and Queens. Bronx and Staten Island have comparatively lesser trips. Trips within the same borough in the top 3 boroughs (Manhattan, Brooklyn, Queens) generate maximum profits. Routes involving Staten Island are least profitable. As a stakeholder, I would implement dynamic pricing algorithms to earn maximum profits during peak hours in the popular boroughs. I would consider improving services in routes involving Staten Island to increase operations in that area.

Challenges encountered in this task

- Creating the window partitions on Month and ranking rows within each partition
- Concatenating the pickup and dropoff columns to a single column
- Discovering the limit() function to return top n rows

I overcame these challenges by looking up spark documentations and tutorials on the internet. It was also helpful to look at output of each step by running the spark job after writing each step, to understand syntax of new APIs.

Knowledge / Insight attained

- Learned how to create partitions on a DataFrame on a given attribute and use Window functions to perform operations on the range of rows within each partition
- Learned how to use the filter() function to extract rows satisfying a given condition
- Learned how to concatenate 2 or more columns to a single column using the concat() and lit() functions
- Learned the limit() function to return top n rows of the DataFrame

Task 4

APIs used:

1. **pyspark.sql.SparkSession**
2. **pyspark.sql.functions:** In this task, we use the following built-in functions of this API: avg(), col()

Steps:

1. **Loading the rideshare_data.csv dataset**
2. **Calculating average driver total pay at different time of day**

   groupby(): grouping DataFrame by different times of day

   avg(): Function to return average of all values of the driver_total_pay column for each time_of_day group

   sort(): Function to sort the DataFrame on the given column in either ascending or descending order

3. **Calculating average trip length at different time of day**

   Same functions as step 2

4. **Calculating average earning per mile at different time of day**

   join(): Joining avg_driver_total_pay with avg_trip_length on time_of_day column

   withColumn(): Adding a new column average_earning_per_mile obtained from dividing average_driver_total_pay by average_trip_length

   drop(): Dropping average_driver_total_pay and average_trip_length columns and also dropping duplicate time_of_day column due to join operation

Output

1) Table of Average driver total pay during different times of a day

time_of_day	average_driver_total_pay
afternoon	21.21242875659354
night	20.087438003592712
evening	19.777427702398395
morning	19.633332793944838

2) Table of Average trip length during different times of a day

time_of_day	average_trip_length
night	5.32398480196174
morning	4.927371866442788
afternoon	4.861410525661209
evening	4.484750367447519

3) Table of Average earning per mile during different times of a day

time_of_day	average_earning_per_mile
afternoon	4.363430869420023
night	3.773008141606838
morning	3.984544565766397
evening	4.4099283308949575

4) **Insights**

   From the tables, it is inferred that afternoon is the peak time during which the fare is maximum. Night fares are second highest which can be due to limited options available at that time. The average trip length at night is also the highest which may be due to non-availability of public transport services at that time for long distance travels causing people to opt for taxis. As a stakeholder, I would deploy more services during the afternoon to meet increasing demands and maximize profits. I would also make more long-distance vehicles available at night.

Challenges encountered in this task

- Obtaining the average_earning_per_mile column from dividing average_driver_total_pay by average_trip_length

I overcame these challenges by looking up spark documentations and tutorials to learn how to perform aggregate functions arithmetic operations on columns.

Knowledge / Insight attained

- Learned how to use the avg() aggregate function to calculate average of columns values in each group
- Learned how to perform arithmetic division on 2 columns to get a new column

Task 5

APIs used:

1. **pyspark.sql.SparkSession**
2. **pyspark.sql.functions:** In this task, we use the following new built-in functions of this API: month()

Steps:

1. **Loading the rideshare_data.csv dataset**
2. **Calculating average waiting time on each day in January**

   where(): Function to filter rows based on the given condition. Here we filter rows of only January days by specifying condition if month = 1

   month(): Function to extract the month as an integer from given date/timestamp

   from_unixtime(): Function to convert a Unix timestamp (number of seconds since January 1, 1970) to a string representing timestamp of that moment.

   withColumn(): Make changes to the ‘date’ column

date_format(): Change value of column ‘date’ from UNIX time to string ‘dd’ format

groupby(): Group by ‘date’ column

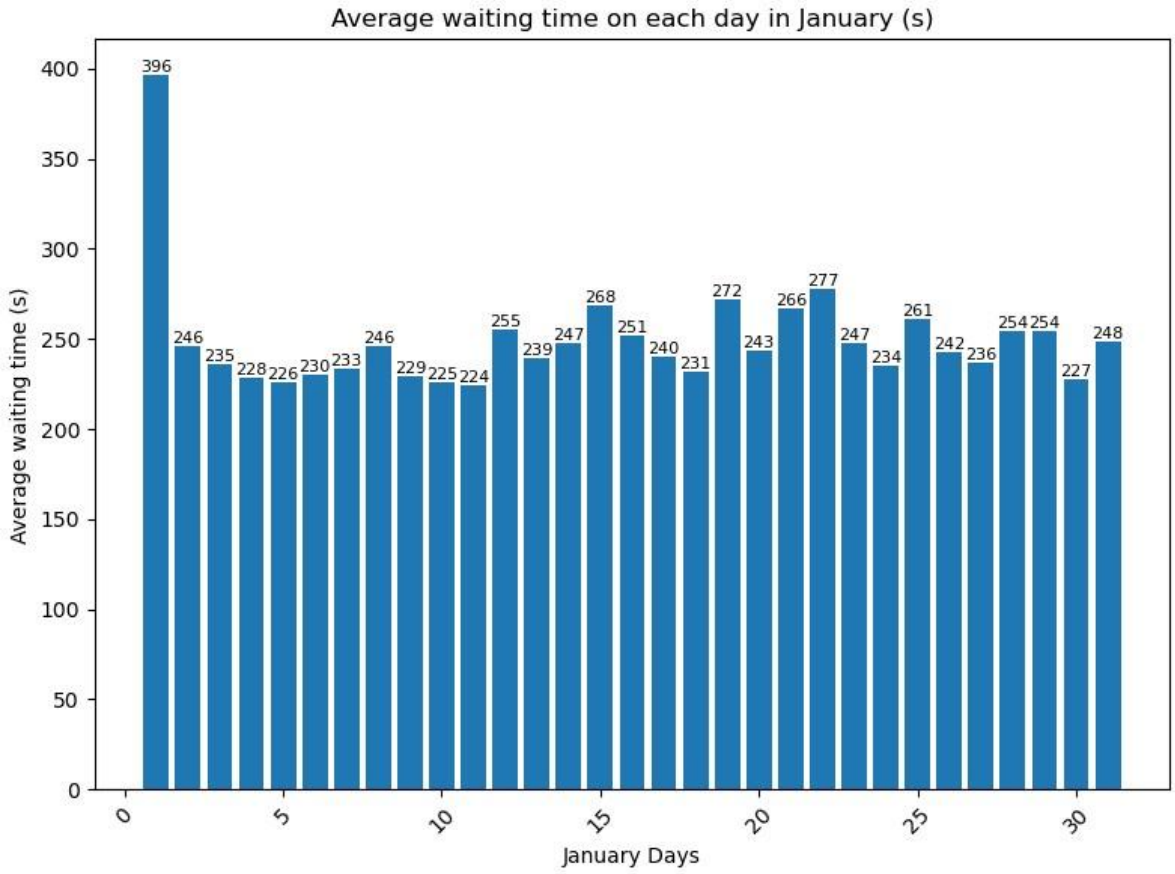
agg(): Perform aggregate function

avg(): Calculate average request to pickup time on each day

sort(): sort DataFrame in ascending order of the days

Output

1) Histogram of average waiting time on each day in January



2) From the plot, it is observed that the average waiting time on **January 1** was 396 seconds, i.e., exceeding 300 seconds

3) January 1st is New Year’s day, which is a public holiday. There might be an increased demand for taxi services due to people going out to celebrate, attend events, or travel to visit friends and family. Increased demand coupled with limited availability of drivers could lead to longer wait times.

Challenges encountered in this task

- Filtering the rows containing only records of January days
- Using the month() function to extract the month integer along with from_unixtime()

I overcame these challenges by trying out different SQL date and timestamp functions

Knowledge / Insight attained

- Learned how to use where() to filter rows of a DataFrame based on a given condition
- Figured out how to use month() in extracting the month number as an integer
- Learned how to use date_format() to express the date in 'dd' format

Task 6

APIs used:

3. **pyspark.sql.SparkSession**
4. **pyspark.sql.functions**

Steps:

1. **Loading the rideshare_data.csv and taxi_zone_lookup.csv datasets**
2. **Calculating Trip counts greater than 0 and less than 1000 for different Pickup Borough at different time of day**

   join(): Joining the 2 DataFrames

   groupby(): Grouping DataFrame by Pickup_Borough and time_of_day and performing aggregated count

   where(): Used for Filtering the records where trip_count>0 and trip_count<1000
3. **Calculating Trip counts for each Pickup Borough in the evening time**

   join(): Same as step 2

   groupby(): Same as step 2

   where(): Used for Filtering the records where time_of_day is 'evening'
4. **Calculating the number of trips that started in Brooklyn and ended in Staten Island**

   join(): Joining DataFrames twice to include all columns

select(): Function used to select multiple columns from the DataFrame by passing column names

where(): Used for Filtering the records where Pickup_Borough is ‘Brooklyn’ and Dropoff_Borough is ‘Staten Island’

count(): Count the number of rows of the resulting DataFrame

Output

1) Trip counts greater than 0 and less than 1000 for different Pickup Borough at different time of day

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

2) Trip counts for each Pickup Borough in the evening time

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

3) Number of trips that started in Brooklyn and ended in Staten Island (10 samples)

```
+--------------+---------------+-------------------------+
|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                |
+--------------+---------------+-------------------------+

Number of trips that started in Brooklyn and ended in Staten Island: 69437
```

Challenges encountered in this task

- Filtering the rows based on multiple conditions combined using the conditional expression '&' in each sub-task. Took multiple attempts to get the syntax right

I overcame these challenges by looking up spark documentations and tutorials, and trying out different ways to get the syntax right

Knowledge / Insight attained

- Learned how to use the conditional expression '&' to combine multiple conditions for filtering rows of a DataFrame

Task 7

APIs used:

1. **pyspark.sql.SparkSession**
2. **pyspark.sql.functions**

Steps:

1. **Loading the rideshare_data.csv and taxi_zone_lookup.csv datasets**
2. **Calculating Trip counts on each Route for Uber, Lyft and adding both to get total count**

   join(): Joining the 2 DataFrames

   total_count_df DataFrame:

   withColumn(): Adding a new column 'Route'

   concat(): Concatenating Pickup_Zone and Dropoff_Zone columns with literal 'to' to get a single column

   groupby(): Grouping DataFrame by Route and performing aggregated count

   uber_df DataFrame:

   where(): Used for Filtering Uber records

   Followed by withColumn(), concat(), groupby() same as above

   merged_df DataFrame

   join(): Joining total_count_df and uber_df on 'Route' column and adding a new column 'lyft_count' by subtracting the column uber_count from column total_count

Output

Route	uber_count	lyft_count	total_count
JFK AirporttoNA	253211	46	253257
East New YorktoEast New York	202719	184	202903
Borough ParktoBorough Park	155803	78	155881
LaGuardia AirporttoNA	151521	41	151562
CanarsietoCanarsie	126253	26	126279
South Ozone ParktoJFK Airport	107392	1770	109162
Crown Heights NorthtoCrown Heights North	98591	100	98691
Bay RidgetoBay Ridge	98274	300	98574
AstoriatoAstoria	90692	75	90767
Jackson HeightstoJackson Heights	89652	19	89671

Challenges encountered in this task

- Figuring out how to get separate columns for uber_count and lyft_count

I overcame these challenges by first aggregating on entire dataset for total counts and then extracting a dataset of only Uber records and performing aggregated count. Finally, I obtained the Lyft count by subtracting Uber count from total count.

Knowledge / Insight attained

- Performing aggregated count on each attribute value and assigning it as a separate column, for example, separating the Uber only dataset and getting aggregated count on each route and finally merging it with entire dataset's aggregated total count.
- Assigning the difference in total and Uber counts as Lyft count.

Task 8

1. **StructType of vertexSchema and edgeSchema**

```
vertexSchema = StructType([StructField("id", IntegerType(), False),
                           StructField("Borough", StringType(), True),
                           StructField("Zone", StringType(), True),
                           StructField("service_zone", StringType(), True)])

edgeSchema = StructType([StructField("src", IntegerType(), False),
                           StructField("dst", IntegerType(), False)])
```

2. **Vertices DataFrame**

```
+---+-------------+----------------------+------------+
|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   |
+---+-------------+----------------------+------------+
```

Edges DataFrame

src	dst
151	244
244	78
151	138
138	151
36	129
138	88
200	138
182	242
248	242
242	20

3. Graph DataFrame

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]

4. **Connected vertices with the same Borough and same service_zone**

```
+---+---+---------+------------+
|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   |
+---+---+---------+------------+
```

```
Number of connected vertices with same Borough and same service_zone: 46886992
```

5. **Page Ranking**

```
+---+------------------+
|id |pagerank          |
+---+------------------+
|265|11.105433344108407|
|1  |5.471845424920648 |
|132|4.5511325720677815|
|138|3.5683223416560614|
|61 |2.676397365341704 |
+---+------------------+
```

References

https://sparkbyexamples.com/pyspark-tutorial/

https://graphframes.github.io/graphframes/docs/_site/index.html

https://spark.apache.org/docs/latest/api/python/

https://spark.apache.org/docs/3.1.2/api/python/getting_started/quickstart.html#DataFrame-Creation

https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page