Big Data Processing (ECS76P)

COURSEWORK REPORT

Queen Mary
University of London

Submitted By:
Name: Aman Naredi
Student Id: 230290873

TABLE OF CONTENTS

TASK 1. MERGING DATASETS

1.1 Loading Datasets **2**
1.2 Joining Datasets **4**
1.3 Converting Date Format **5**
1.4 Verifying Datarame Structure **6**

TASK 2 AGGREGATION OF DATA

2.1 Counting trips per buisness months **8**
2.2 Calculating Platform Profits per Business per Month **12**
2.3 Calculating Driver Earnings per Business per Month **15**
2.4 Insights **19**

TASK 3 TOP K PROCESSING

3.1 Top 5 Pickup Boroughs per Month **20**
3.2 Top 5 Dropoff Boroughs per Month **23**
3.3 Top 30 Earnest Routes **24**
3.4 Insights **26**

TASK 4 AVERAGE DATA

4.1 Average Driver Total Pay by Time of Day **27**
4.2 Average Trip Length by Time of Day **29**
4.3 Average Earning per Mile by Time of Day **30**
4.4 Insights **31**

TASK 5 FINDING ANOMALIES

5.1 Average Waiting Time per Day in January **32**
5.2 Days with High Average Waiting Time **36**
5.3 Insights **36**

TABLE OF CONTENTS

TASK 6 FILTERING DATA

6.1 Trip Counts with Conditions 37
6.2 Evening Trip Counts by Pickup Borough 38
6.3 Trips from Brooklyn to Staten Island 39

TASK 7 ROUTE ANALYSIS

7.1 Top 10 Popular Routes Analysis 41

TASK 8 GRAPH PROCESSING

8.1 Defining structype of vertex and edge schema 43
8.2 Construct Edges, Vertices DataFrames and Graph 44
8.3 Creating a graph using vertices and edges 46
8.4 Counting connected vertices with the same Borough and same service_zone 47
8.5 Performing page ranking on the graph dataframe 49

Task 1 Merging Datasets

1.1 LOADING DATASETS

Steps and APIs used:

1. *SparkSession Creation:*
   - Started the coursework by importing **`SparkSession`** from **`pyspark.sql`** and used `SparkSession.builder.appName("NYC").getOrCreate()` to create a `SparkSession` named "NYC" for interacting with Spark.
2. *Defining helper functions:*
   - To validate datasets, defined two helper functions: `good_ride_line` and `good_taxi_line`.
     - good_ride_line: This function takes a `line` of text (a row from rideshare_data.csv) and validates if it has 15 comma-separated fields. It returns True if valid, False otherwise.
     - good_taxi_line: Similar to good_ride_line, it validates if a line from taxi_zone_lookup.csv has 4 comma-separated fields and returns True if valid.
3. *Accessing S3 data:*
   - With reference to the starterkit.py, retrieved environment variables for S3 bucket details like DATA_REPOSITORY_BUCKET, S3_ENDPOINT_URL, BUCKET_PORT, AWS_ACCESS_KEY_ID, and AWS_SECRET_ACCESS_KEY.
   - Using this, configured Spark's Hadoop configuration to access the S3 bucket using the retrieved credentials and endpoint details.
4. *Loading rideshare data:*
   - Read the "rideshare_data.csv" file from S3 using **`spark.sparkContext.textFile`**.
   - **`filter`**`(good_ride_line)` applies the `good_ride_line` function for filtering.
   - The data is transformed into an **RDD** (Resilient Distributed Dataset), which consists of tuples of “,” separated values (rows) using **`map`**.

- The header row, that is the first row of is identified and saved as `header_ride` using **`first`** and removed from the RDD using **`filter`**.
- The RDD is converted into a DataFrame named ride_df with column names specified in column_names using **`toDF`**.

5. *Taxi zone data (taxi_zone_lookup.csv):*
   - Similar steps are followed as for rideshare data to load "taxi_zone_lookup.csv from S3 and create a DataFrame named `taxi_df` with column names in `column_names_taxi`.
6. *Cleaning taxi zone data: API's used [**`regex_replace`**]*
   - To clean taxi_daf, iterated through each column name in column_names_taxi.
   - Used **`regexp_replace`** for each column to remove double quotes (") from the corresponding column in taxi_df.

7. *API's Used:*
   - sparkContext.textFile
   - filter
   - map
   - toDF.
   - withColumn
   - regexp_replace

Challenges encountered:

- Removing double quotes from Taxi zone data

Knowledge gained:

- The task showcases techniques for loading data from CSV files, handling headers, transforming data into DataFrames, and exploring their structure.
- Learned cleaning data by removing unwanted characters (double quotes) from a column.

1.2 JOINING DATASETS

Steps and APIs used:

1. *Joining on pickup location:*
    - Used the **`join`** function from Spark DataFrames to join the ride_df and taxi_df DataFrames.
    - The join condition is specified as `ride_df.pickup_location == taxi_df.LocationID`. This matches rows in the rideshare data where the pickup location matches the LocationID in the taxi zone data.
    - The resulting DataFrame is named joined_df_pickup.
2. *Renaming columns (pickup):*
    - Used a series of **`withColumnRenamed`** functions to rename the relevant columns obtained from the taxi zone data:
        - "Borough" -> "Pickup_Borough"
        - "Zone" -> "Pickup_Zone"
        - "service_zone" -> "Pickup_service_zone"
    - Also used **`drop`**`("LocationID")` to remove the redundant "LocationID" column from the joined data.
    - The resulting DataFrame is named joined_df_Renamed.
3. *Joining on drop-off location:*
    - Performed a second **`join`** using the same approach as previous step, but this time joining `joined_df_Renamed` (which now includes pickup zone information) with `taxi_df` based on the dropoff_location field in the rideshare data.
    - The resulting DataFrame is named joined_df_dropoff.
4. *Renaming columns (drop-off):*
    - Similar to step 2, renamed the columns obtained from the taxi zone data for the drop-off location with the prefix "Dropoff_".
    - We again remove the redundant "LocationID" column.
    - The final enriched DataFrame is named nyc_df.
    - We use **`printSchema()`** to display the schema of the final DataFrame.
5. *API'S used:*
    - join()
    - withColumnRenamed()
    - drop()

Knowledge gained:

- Learned how to perform multi-step joins on DataFrames using the join function with appropriate join conditions.
- It showcases techniques for renaming columns and manipulating DataFrame structures.

1.3 CONVERTING DATE FORMAT

Steps and APIs used:

1. *Accessing the date column:*
   - In this task the focus is on the "date" column in the nyc_df DataFrame, accessed date column using **`nyc_df.date`**
2. *Converting Unix time stamp to date and its data type from string to date, thus creating a new column named date:*
   - Use the **`from_unixtime`** function from **`pyspark.sql.functions`** with the **`withColumn`** function with alias date to create a new column in nyc_df named "date" to convert the UNIX timestamp store in the "date" column.
   - Specified the format string **"**`yyyy-MM-dd`**"** to indicate the desired output format.
   - **`date_format`** takes the previously converted temporary column and the desired output format string ("yyyy-MM-dd") to ensure the final column has the correct format.
   - Converted the data type of this new "date" column using **`to_date`** on nyc_df date column from **string** to **date** inside **`withColumn`** to make changes in the column.
3. *API's used:*
   - from_unixtime
   - withColumn
   - date_format
   - to_date

Challenges encountered:

1. Ensuring the conversion of data type of date column from string to date after converting its format.

Knowledge gained:

- This task demonstrates working with date formats in Spark DataFrames.
- It highlights the use of from_unixtime, date_format and to_date functions for date and its data type conversion.

Output:

```
2024-03-28 16:46:48,770 INFO codegen.CodeGenerator: Code generated in 29.759298 ms
+--------+---------------+----------------+-----------+-----------------+---------------+------------------+-------------------+-----------+---
-------+--------------+----------------+----------------+-----------+----------------+--------------+-----------+------------------+----------
-----+------------+--------------------+
|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_Bo
rough|Dropoff_Zone|Dropoff_service_zone|
+--------+---------------+----------------+-----------+-----------------+---------------+------------------+-------------------+-----------+---
-------+--------------+----------------+----------------+-----------+----------------+--------------+-----------+------------------+----------
-----+------------+--------------------+
|    Uber|             25|             125|        9.8|            299.0|         2107.0|              45.0|             2152.0|      night|202
3-04-09|         51.28|           36.02|           15.26|      60.26|            3.68|      Brooklyn|Boerum Hill|          Boro Zone|       Manh
attan|   Hudson Sq|         Yellow Zone|
|    Uber|             25|             125|       3.32|            221.0|          945.0|             121.0|             1066.0|    morning|202
3-04-09|         25.24|           13.72|           11.52|      46.33|            4.13|      Brooklyn|Boerum Hill|          Boro Zone|       Manh
attan|   Hudson Sq|         Yellow Zone|
|    Uber|             25|             125|       3.98|            321.0|         1669.0|              47.0|             1716.0|  afternoon|202
3-04-09|         30.23|           20.92|            9.31|      43.89|            5.26|      Brooklyn|Boerum Hill|          Boro Zone|       Manh
attan|   Hudson Sq|         Yellow Zone|
|    Uber|             25|             125|       3.68|            376.0|         2681.0|              20.0|             2701.0|  afternoon|202
3-04-09|         33.59|           31.07|            2.52|      41.41|            8.44|      Brooklyn|Boerum Hill|          Boro Zone|       Manh
attan|   Hudson Sq|         Yellow Zone|
|    Uber|             25|             125|        3.6|             54.0|         1081.0|              29.0|             1110.0|    morning|202
3-05-08|         22.69|           18.89|             7.8|      61.26|            5.25|      Brooklyn|Boerum Hill|          Boro Zone|       Manh
attan|   Hudson Sq|         Yellow Zone|
|    Uber|             25|             125|        3.6|             92.0|         1003.0|              12.0|             1015.0|  afternoon|202
3-05-08|         34.83|            18.8|           20.67|      66.68|            5.22|      Brooklyn|Boerum Hill|          Boro Zone|       Manh
attan|   Hudson Sq|         Yellow Zone|
|    Uber|             25|             125|        3.7|           1052.0|         1363.0|             111.0|             1474.0|  afternoon|202
3-05-08|         21.26|           20.28|            0.98|      49.53|            5.48|      Brooklyn|Boerum Hill|          Boro Zone|       Manh
attan|   Hudson Sq|         Yellow Zone|
|    Uber|             25|             125|       3.42|            431.0|         1015.0|              16.0|             1031.0|      night|202
3-05-08|         21.42|           14.03|            7.39|      48.99|             4.1|      Brooklyn|Boerum Hill|          Boro Zone|       Manh
attan|   Hudson Sq|         Yellow Zone|
|    Uber|             25|             125|       3.87|            187.0|         1022.0|               7.0|             1029.0|    morning|202
3-05-09|         47.48|            23.9|           23.58|      83.62|            6.18|      Brooklyn|Boerum Hill|          Boro Zone|       Manh
attan|   Hudson Sq|         Yellow Zone|
|    Uber|             25|             125|        3.8|            683.0|         1223.0|              59.0|             1282.0|    morning|202
3-05-09|         47.22|           24.27|           29.24|      68.15|            6.39|      Brooklyn|Boerum Hill|          Boro Zone|       Manh
attan|   Hudson Sq|         Yellow Zone|
+--------+---------------+----------------+-----------+-----------------+---------------+------------------+-------------------+-----------+---
-------+--------------+----------------+----------------+-----------+----------------+--------------+-----------+------------------+----------
-----+------------+--------------------+
only showing top 10 rows
```

1.4 VERIFYING DATAFRAME STRUCTURE

Steps and APIs used:

1. *Counting rows:*
   - Used **`count`** function from Spark DataFrames to get the total number of rows in nyc_df.
   - The result is printed to the console using an f-string to include informative text.
2. *Printing schema:*
   - Printed schema (column names and data types) using **`printSchema`** function of nyc_df.
3. *API's used:*
   - count
   - printSchema

Knowledge gained:

- This task demonstrates using count and printSchema functions to verify DataFrame structure.
- It highlights the importance of checking data quality and consistency after data processing steps.

Output:

```
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: date (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)
```

Task 2 Aggregation of Data

2.1 COUNTING TRIPS PER BUSINESS PER MONTH

Steps and APIs used:

1. *Data Loading and Preparation (Covered in Task 1.1 Report)*

   - Created `SparkSession`.
   - Rideshare data and taxi zone lookup data are loaded from S3 using textFile and converted into DataFrames.
   - Cleaned taxi_df using **`regexp_replace`**.
   - DataFrames are joined based on pickup and dropoff locations using join and columns are renamed.
   - The date column is formatted to "yyyy-MM-dd" format using date_format.

2. *Data Aggregation and Visualization:*

   - **Converting Data Types:**
     - The date column is converted to a date type using **`to_date`**.
     - rideshare_profit and driver_total_pay columns are cast to float type using **`cast("float")`**.
   - **Extracting Month:**
     - A new column named month is added using `month` to represent the month from the date.
   - **Counting Trips:**
     - **`groupBy`** is used on business and month to create groups.
     - **`agg`** with **`count`**`("*")`**`.alias`**`("trip_count")` aggregates the number of trips in each group and assigns an `alias (trip_count)` to the result.
     - trips_per_business_month DataFrame stores the aggregated data.

3. *Saving to S3 Bucket:*

   - **S3 Resource Object:**
     - Created a resource object for my S3 bucket using the boto3 library. This object allowed me to interact with S3 for storing and retrieving data.
   - **Current Date and Time:**
     - It retrieves the current date and time using **`datetime.now()`** and formats it using **`strftime`**. This timestamp is used to create a unique filename for the output file.
   - **Coalescing DataFrame:**

- The **`coalesce(1)`** function reduces the number of output files written to S3 by merging smaller files into a single file.

- **Output Path:**
  - Defined the S3 path to save the DataFrame. It combines the bucket name, a folder structure (`aman_<date_time>`), and the filename (trips_per_business_month.csv) with the .csv extension.
- **Saving as CSV:**
  - The **`write.csv`** method of the DataFrame is used to specify:
    - path: The S3 path constructed earlier.
    - mode: "overwrite" to replace existing files with the same name.
    - header: True to include the column names in the CSV file.

4. *Copying to Local Storage:*

- Copied trips_per_business_per_month.csv from S3 bucket to my local file system in output folder using command :
```
ccc method bucket cp -r bkt:aman_28-03-2024_17:51:57/trips_per_business_month.csv/ output
```

5. *Histogram Generation (Python Script):*

- **Data Loading:**
  - `pandas` is used to read the trips_per_business_month.csv file into a pandas DataFrame named task2_1.
- **Filtering Data:**
  - Separate DataFrames for Uber (`task2_1_uber`) and Lyft (`task2_1_lyft`) trips are created by filtering task2_1 based on the business column.
- **Histogram Creation with Matplotlib:**
  - Generated two histograms(barplots) using plt.bar.
  - Each plot focuses on either Uber or Lyft trips with bars representing trip counts for each month.
  - Added titles, labels, and customization (e.g., colors) for clarity.

6. *API’s used (same in all 3 parts):*

- month()
- cast(“float”)
- groupBy()
- agg()
- count(“*”)
- sum()

- orderBy()
- coalesce()
- datetime.now()
- strftime("%d-%m-%Y_%H:%M:%S")
- boto3.resource

Challenges encountered:

- Faced problems in saving the files in bucket. Learned about coalesce functionality and merging files.

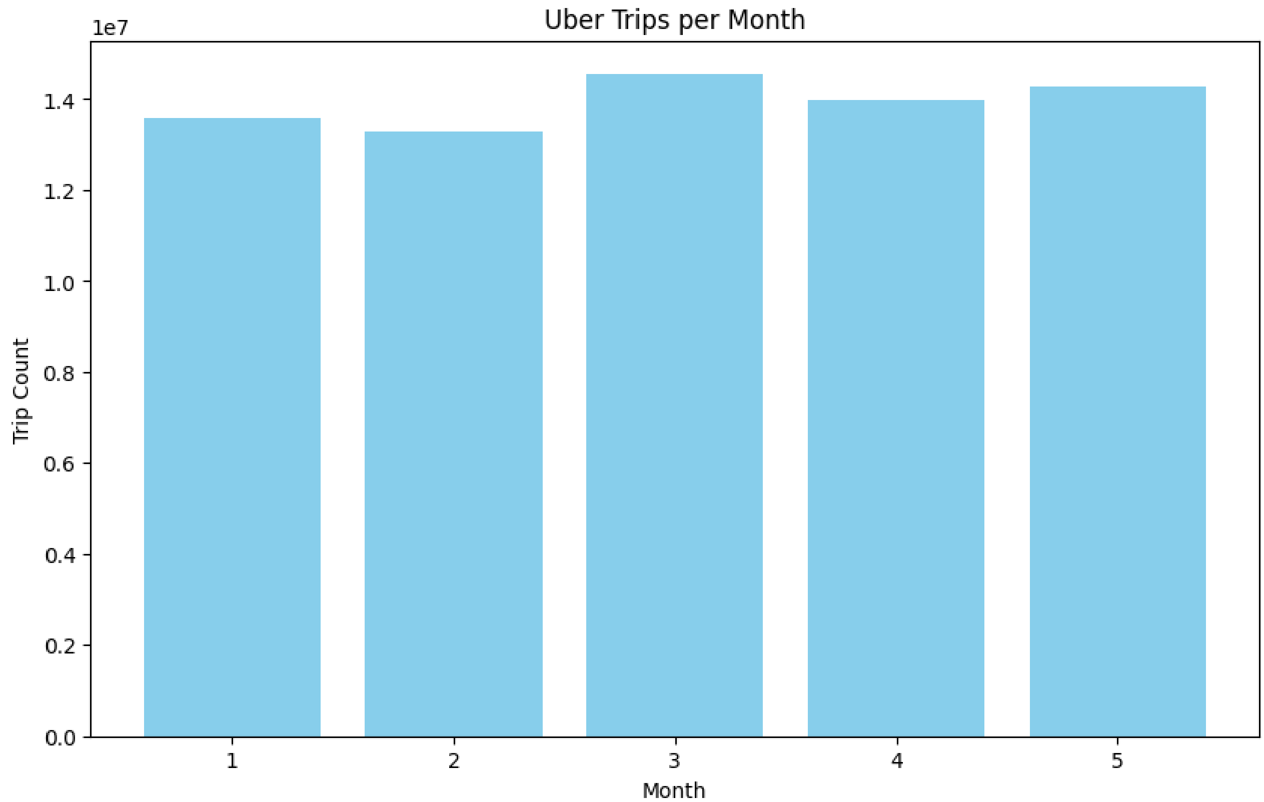
Knowledge gained:

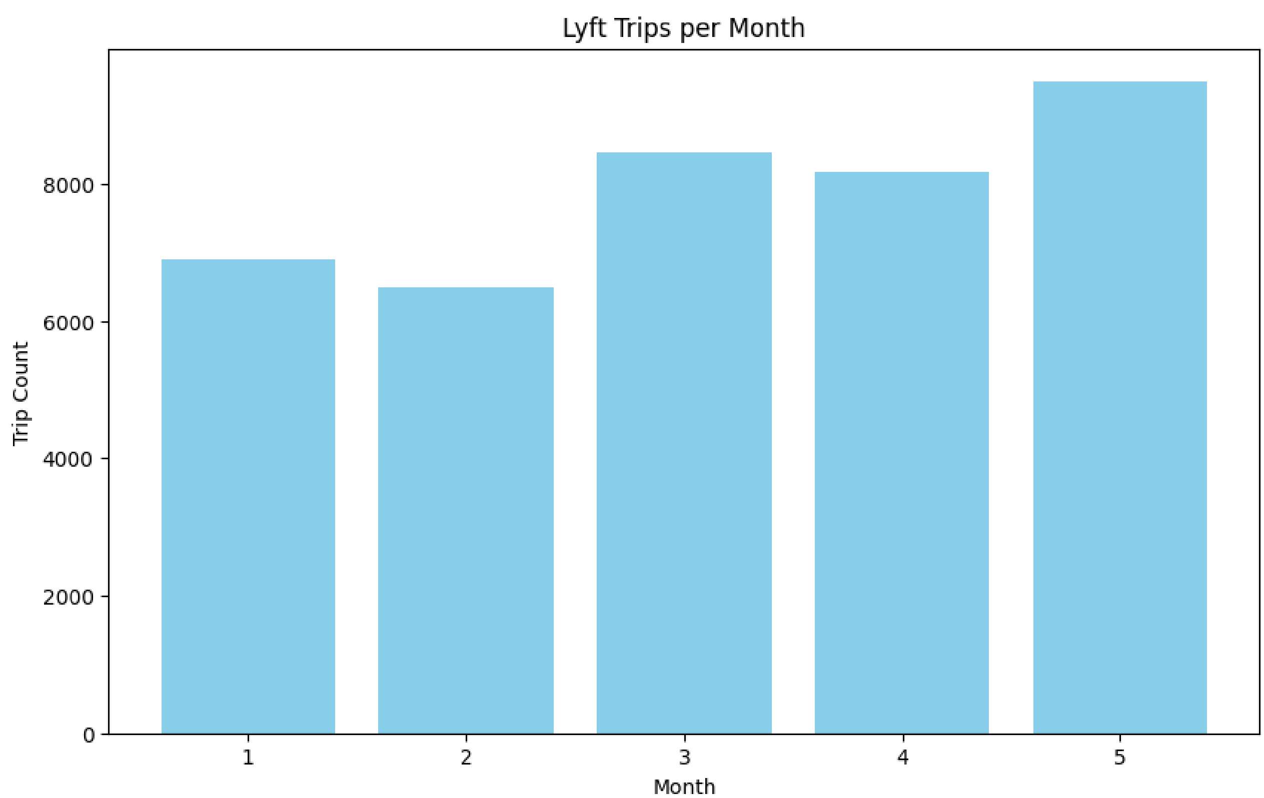
- It highlights the use of aggregation functions like groupBy and count to calculate group-wise statistics.
- Learned the functionality of coalesce API and saving dataframes as csv files into bucket and then how to save them into our local system.

Output:

```
2024-03-28 17:51:57,123 INFO codegen.CodeGenerator: Code generated in 10.898244 ms
+--------+-----+----------+
|business|month|trip_count|
+--------+-----+----------+
|    Lyft|    4|      8173|
|    Uber|    5|  14276372|
|    Uber|    4|  13995860|
|    Lyft|    3|      8444|
|    Uber|    2|  13280761|
|    Lyft|    1|      6887|
|    Uber|    3|  14554308|
|    Uber|    1|  13579077|
|    Lyft|    2|      6491|
|    Lyft|    5|      9491|
+--------+-----+----------+

2024-03-28 17:52:01,481 WARN commit.AbstractS3ACommitterFactory: Using standard FileOutputCommitter to commit work. This is slow and potentiall
```


Uber Trips per Month
1e7
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Trip Count
1
2
3
4
5
Month


Lyft Trips per Month
8000
6000
4000
2000
0
Trip Count
1
2
3
4
5
Month

2.2 CALCULATING PLATFORM PROFITS PER BUSINESS PER MONTH

Steps and APIs Used:

1. Data Loading and Preparation (Done in part 2.1)

2. Calculating Platform Profits:

1. ***Converting Data Type:***
   - The rideshare_profit column is casted to an float type using **`cast(“float”)`** for numerical operations.
2. ***Profit Aggregation:***
   - **`groupBy`** is used on business and month to create groups.
   - **`agg`** with sum("rideshare_profit").alias("Platform Profit") calculates the total platform profit for each group (business-month) and assigns an alias (Platform Profit) to the result.
   - trips_per_business_month_profit DataFrame stores the aggregated data.

3. Saving to S3 Bucket:(similar to previous part)

- **S3 Resource Object:**
  - Created a resource object for my S3 bucket using the boto3 library. This object allowed me to interact with S3 for storing and retrieving data.
- **Current Date and Time:**
  - It retrieves the current date and time using **`datetime.now()`** and formats it using **`strftime`**. This timestamp is used to create a unique filename for the output file.
- **Coalescing DataFrame:**
  - The **`coalesce(1)`** function reduces the number of output files written to S3 by merging smaller files into a single file.
- **Output Path:**
  - Defined the S3 path to save the DataFrame. It combines the bucket name, a folder structure (`aman_<date_time>`), and the filename (trips_per_business_month_profit.csv) with the .csv extension.
- **Saving as CSV:**
  - The **`write.csv`** method of the DataFrame is used to specify:
    - path: The S3 path constructed earlier.
    - mode: "overwrite" to replace existing files with the same name.
    - header: True to include the column names in the CSV file.

4. *Copying to Local Storage:*

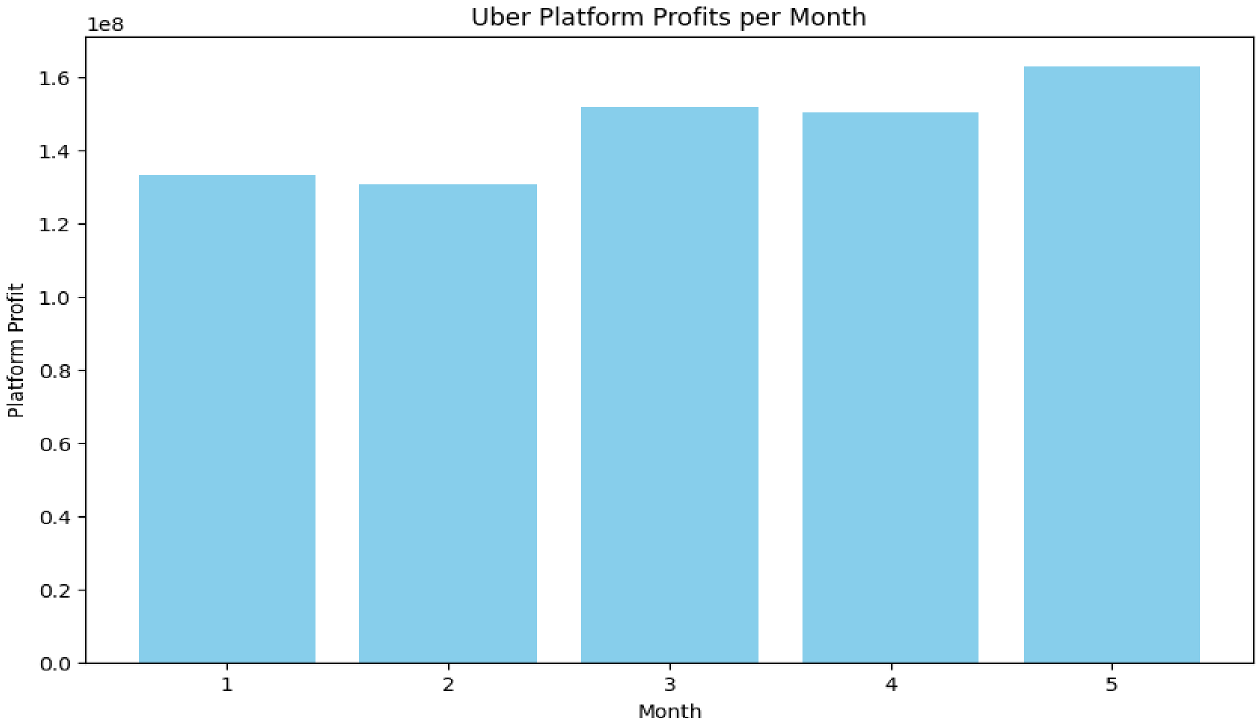
   - Copied trips_per_business_per_month.csv from S3 bucket to my local file system in output folder using command :
     `ccc method bucket cp -r bkt:aman_03-04-2024_13:05:20/trips_per_business_month_profit.csv output`

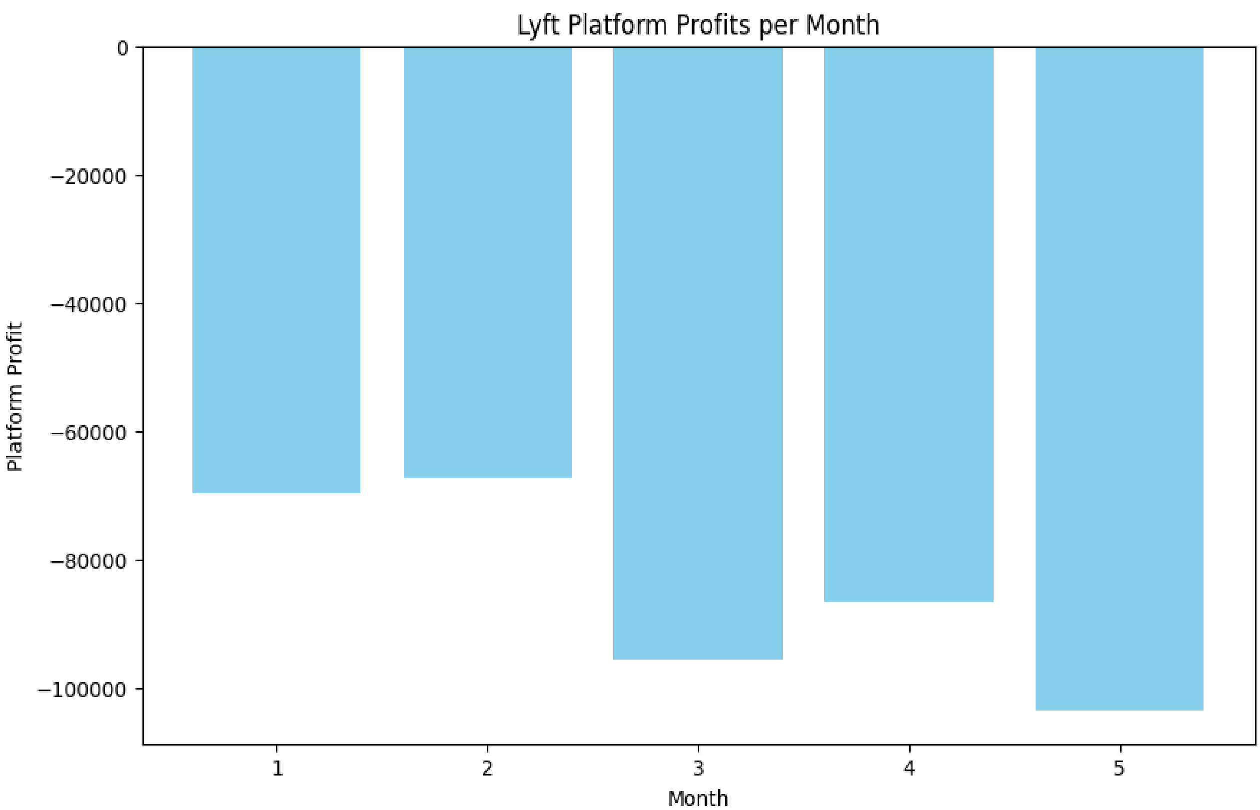
5. *Visualization using Python Script(Similar to previous task):*

- *Libraries:*
  - Using **pandas** for data manipulation and **matplotlib.pyplot** for creating histograms.
- *Reading CSV:*
  - Read the downloaded CSV file(`trips_per_business_month_profit.csv`) into a pandas DataFrame (`task2_2`).
- *Filtering Data:*
  - Created separate Dataframes for Uber (`task2_2_uber`) and Lyft (`task2_2_lyft`) trips by filtering `task2_2` based on the business column.
- *Histogram Creation:*
  - The script uses Matplotlib to create separate histograms for Uber and Lyft platform profits, similar to the approach used for trip counts.

Output:

```
2024-04-03 02:12:21,555 INFO scheduler.DAGScheduler: Job 8 finished: showString at NativeMethodAccessorImpl.java:0, took 0.099491 s
2024-04-03 02:12:21,593 INFO codegen.CodeGenerator: Code generated in 14.838169 ms
+--------+-----+--------------------+
|business|month|     Platform Profit|
+--------+-----+--------------------+
|    Lyft|    4|   -90197.13001759537|
|    Uber|    5|1.6313361550055724E8|
|    Uber|    4| 1.502698201941709E8|
|    Lyft|    3|   -99403.93998675235|
|    Uber|    2|1.3062880563633618E8|
|    Lyft|    1|    -72633.3500049822|
|    Uber|    3| 1.520728764191219E8|
|    Uber|    1|1.3319711162465689E8|
|    Lyft|    2|   -70064.72000297531|
|    Lyft|    5|  -107719.21000343934|
+--------+-----+--------------------+
```


Uber Platform Profits per Month
1e8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Platform Profit
1
2
3
4
5
Month


Lyft Platform Profits per Month
0
−20000
−40000
−60000
−80000
−100000
Platform Profit
1
2
3
4
5
Month

2.3 CALCULATING DRIVER EARNINGS PER BUSINESS PER MONTH

Steps and APIs used:

1. *Data Loading and Preparation (Done in part 2.1)*

2. *Calculating Driver Earnings:*

- **Converting Data Type:**
  - The driver_total_pay column is casted to an float type using **`cast("float")`** for numerical operations.
- **Earnings Aggregation:**
  - **`groupBy`** is used on business and month to create groups.
  - **`agg`** with sum("driver_total_pay").alias("Driver Earnings") calculates the total driver earnings for each group (business-month) and assigns an alias (Driver Earnings) to the result.
  - trips_per_business_month_driver_pay DataFrame stores the aggregated data.

3. *Saving to S3 Bucket:(similar to previous part)*

- **S3 Resource Object:**
  - Created a resource object for my S3 bucket using the boto3 library. This object allowed me to interact with S3 for storing and retrieving data.
- **Current Date and Time:**
  - It retrieves the current date and time using **`datetime.now()`** and formats it using **`strftime`**. This timestamp is used to create a unique filename for the output file.
- **Coalescing DataFrame:**
  - The **`coalesce(1)`** function reduces the number of output files written to S3 by merging smaller files into a single file.
- **Output Path:**
  - Defined the S3 path to save the DataFrame. It combines the bucket name, a folder structure (`aman_<date_time>`), and the filename (trips_per_business_month_profit.csv) with the .csv extension.
- **Saving as CSV:**
  - The **`write.csv`** method of the DataFrame is used to specify:
    - path: The S3 path constructed earlier.
    - mode: "overwrite" to replace existing files with the same name.
    - header: True to include the column names in the CSV file.

4. *Copying to Local Storage:*

   - Copied trips_per_business_per_month.csv from S3 bucket to my local file system in output folder using command : `ccc method bucket cp -r bkt:aman_03-04-2024_15:03:52/trips_per_business_month_driver_pay.csv output`

5. *Visualization using Python Script(Similar to previous task):*

- *Libraries:*
  - Using **`pandas`** for data manipulation and **`matplotlib.pyplot`** for creating histograms.
- *Reading CSV:*
  - Read the downloaded CSV file(`trips_per_business_month_driver_pay.csv`) into a pandas DataFrame (`task2_2`).
- *Filtering Data:*
  - Separate Dataframes for Uber (`task2_3_uber`) and Lyft (`task2_3_lyft`) trips are created by filtering `task2_3` based on the business column.
- *Histogram Creation:*
  - The script uses Matplotlib to create separate histograms for Uber and Lyft driver earnings, similar to the approach used for trip counts and platform profits.

Challenges encountered:

- *Data type consistency:* Similar to previous tasks, ensuring "driver_total_pay" is an integer type is essential for accurate sum calculation.

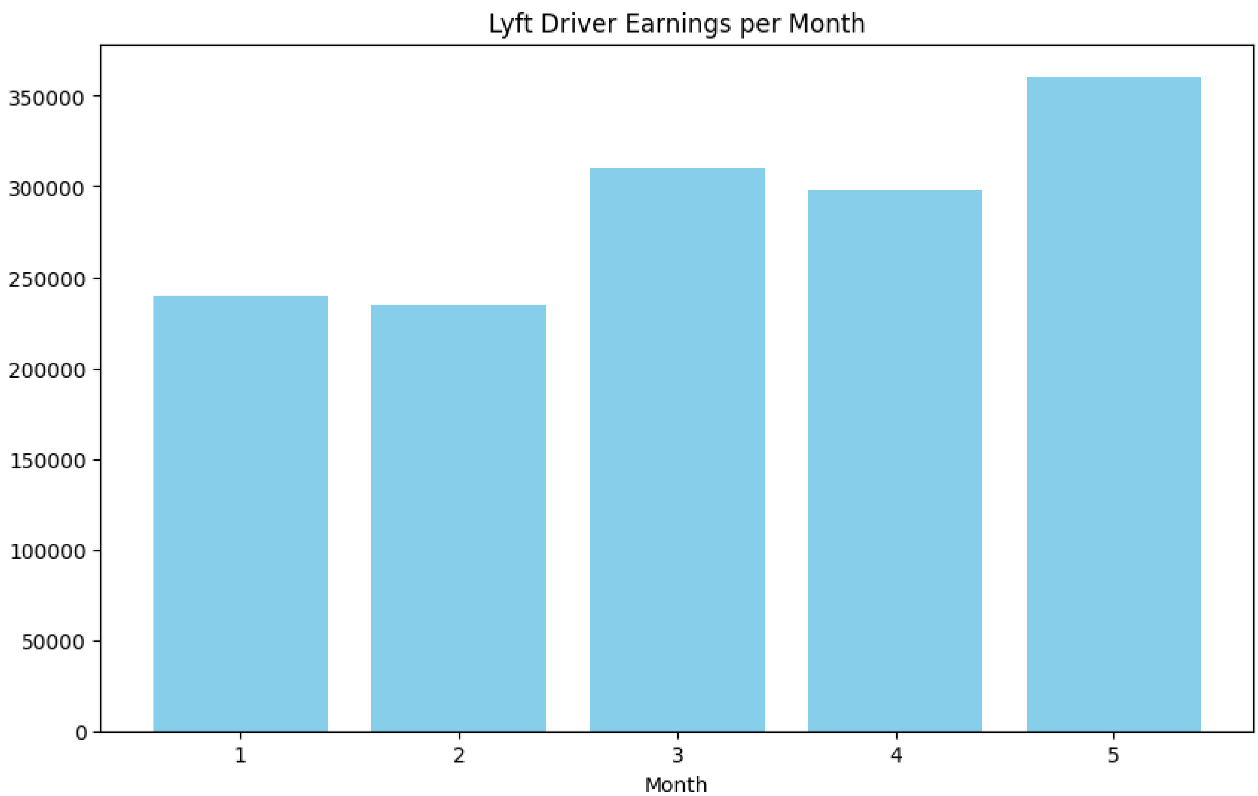
Knowledge gained:

- This task reiterates the use of groupBy for group-wise aggregations.
- It highlights the sum function for calculating total values within a group.

Output:

```
2024-04-04 11:02:58,942 INFO codegen.CodeGenerator: Code generated in 14.635607 ms
+--------+-----+--------------------+
|business|month|     driver_earnings|
+--------+-----+--------------------+
|    Lyft|    4|   297815.3799999999|
|    Uber|    5|3.1300511454999596E8|
|    Uber|    4|2.9506892721999764E8|
|    Lyft|    3|   310276.5499999997|
|    Uber|    2|2.5215597709000513E8|
|    Lyft|    1|   239932.2599999999|
|    Uber|    3|2.9595849601000154E8|
|    Uber|    1|2.5025348066999513E8|
|    Lyft|    2|  234875.53000000003|
|    Lyft|    5|           360408.09|
+--------+-----+--------------------+
```


Uber Driver Earnings per Month
1e8
3.0
2.5
2.0
1.5
1.0
0.5
0.0
1
2
3
4
5
Month


Lyft Driver Earnings per Month
350000
300000
250000
200000
150000
100000
50000
0
1
2
3
4
5
Month

2.4 WHEN WE ARE ANALYZING DATA, IT'S NOT JUST ABOUT GETTING RESULTS, BUT ALSO ABOUT EXTRACTING INSIGHTS TO MAKE DECISIONS OR UNDERSTAND THE MARKET. SUPPOSE YOU WERE ONE OF THE STAKEHOLDERS, FOR EXAMPLE, THE DRIVER, CEO OF THE BUSINESS, STOCKBROKER, ETC, WHAT DO YOU FIND FROM THE THREE RESULTS? HOW DO THE FINDINGS HELP YOU MAKE STRATEGIES OR MAKE DECISIONS?

Considering **trip counts** of both the businesses, **Uber** seems to have **stable ridership**, while **Lyft** experiences a **decline**. This presents a considerable **market capture** by **Uber** than Lyft.

Uber boasts significant and **increasing platform profits**, while **Lyft** faces consistent **losses**. This highlights Uber's current financial strength compared to Lyft.

Similarly to platform profits, **driver montly earnings** for **Uber** seems to be **increasing** although for **Lyft** it is **fluctuating**. This suggests a need for **Lyft** to consider strategies that **incentivize** driving during peak hours of high-demand periods to retain drivers.

These findings can inform strategic decisions for both companies:

- **Uber:**
    - Analyze Lyft's declining market share and explore targeted campaigns to attract those customers.
    - Continue focusing on strategies that are driving the increase in platform profits.
- **Lyft:**
    - Address the decline in ridership through targeted promotions or adjustments to pricing strategy.
    - Implement cost-cutting measures or explore revenue-generating strategies to address negative profits.

Task 3 Top-K Processing

3.1 TOP 5 PICKUP BOROUGHS PER MONTH

Steps and APIs used:

1. Data Loading and Preparation (Covered in Task 1 Report)

- Created `SparkSession`.
- Rideshare data and taxi zone lookup data are loaded from S3 using textFile and converted into DataFrames.
- Cleaned taxi_df using **`regexp_replace`**.
- DataFrames are joined based on pickup and dropoff locations using join and columns are renamed.
- The date column is formatted to "yyyy-MM-dd" format using date_format.

2. Windowing and ranking:

- Defined a **`window`** specification (windowSpec) that partitions data by "month" and sorts them by "trip_count" in descending order.
- Used **`row_number`** function with windowSpec to assign a row number to each record within its month, ranking them by trip count (highest gets 1).
- **`filter`** the DataFrame to keep only rows with "row_num" less than or equal to 5 (top 5 for each month).
- Finally, **`drop`** the "row_num" column as it's no longer needed.

3. Sorting:

- Used **`orderBy`** to sort the resulting DataFrame (top5_borough_monthly_pickup_trips) first by "month" (ascending) and then by "trip_count" (descending).
- This ensures the output shows the top 5 boroughs with the highest trip counts for each month.

4. Viewing results :

- Used **`top5_borough_monthly_trips.show(25)`** to display the results.

5. *API's used* ***(same in all the parts):***

- window.partitionBy(): It defines how the data is partitioned before applying a window function. In this scenario, data is partitioned by month, allowing for separate rankings within each month.Used in both the top 5 pickup and dropoff borough analysis, enabling month-specific ranking.
- concat_ws(): Used to either add a new column or replace an existing one. In this task, it's specifically used for adding the Route column and the row_num column post-window function application
- row_number()
- cast("float")
- groupBy()
- agg()
- count("*")
- sum()
- orderBy()
- coalesce()
- datetime.now()
- strftime("%d-%m-%Y_%H:%M:%S")
- boto3.resource

Challenges encountered:

- Faced challenges in working with window functions and defining other utilities.

Knowledge gained:

- Learned about window functions and using them for ranking data within groups.
- This task highlights techniques for top-k retrievals using row numbering and filtering.

Output:

```
2024-03-28 19:25:52,197 INFO storage.BlockManagerInfo: Removed broadcast_7_piece0 on 10.133.32.138:37843 in memory (size: 11.0 KiB, free: 2.1 G
iB)
+--------------+-----+----------+
|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|
+--------------+-----+----------+
```

3.2 TOP 5 DROPOFF BOROUGHS PER MONTH

Steps and APIs used:

1. *Data Loading and Preparation (Covered in 3.1)*

2. *Window function:*

   - Similar to task 3.1, defined a window specification (windowSpec) using the **`window`** function.
   - This window partitions the data by "month" and **sorts** them by "trip_count" in descending order.

3. *Top 5 calculation:*

   - Used **`row_number`** with the defined window to assign a row number to each entry within each month (partition).
   - **`filter`** the DataFrame to keep only rows with a row number less than or equal to 5 (top 5).
   - The result is stored in **`top5_borough_monthly_dropoff_trips`**.

4. *Sorting:*

   - Used **`orderBy`** to sort the final results first by "month" (ascending) and then by "trip_count" (descending).

Output:

```
iB)
2024-03-28 19:47:24,716 INFO storage.BlockManagerInfo: Removed broadcast_9_piece0 on 10.134.100.247:42961 in memory (size: 25.9 KiB, free: 2.1
GiB)
+---------------+-----+----------+
|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|
+---------------+-----+----------+
```

3.3 TOP 30 EARNEST ROUTES

Steps and APIs used:

1. *Creating a route column:*
    - Used the **`concat_ws`** function to create a new column named "Route" that combines the "Pickup_Borough" and "Dropoff_Borough" columns separated by the string "to ".
    - This creates a **unique identifier** for each route.
2. *Grouping and aggregation:*
    - Used **`groupBy`** on the "Route" column to group trips for each unique route.
    - Calculated the sum of "driver_total_pay" within each group using **`sum`** to represent the total driver earnings for that route.
    - The result is stored in a new DataFrame named **`route_profits`** with columns "Route" and "total_profit".
3. *Sorting and filtering:*
    - Used **`orderBy`** to sort the **`route_profits`** DataFrame by the "total_profit" column in descending order (highest profits first).
    - Used the **`limit`** function to select the **top 30** most profitable routes and store the result in **`top30_routes`**.
4. *Displaying results :*
    - Used **`show(30, truncate=False)`** to display the top 30 routes and their total profits without truncating route names.

Knowledge gained:

- This task demonstrates creating new columns using string manipulation functions.
- It highlights grouping, aggregation, and sorting operations for route-based profitability analysis.

Output:

```
2024-03-28 19:10:35,977 INFO codegen.CodeGenerator: Code generated in 10.925538 ms
+------------------------------+--------------------+
|Route                         |total_profit        |
+------------------------------+--------------------+
|Manhattan to Manhattan        |3.3385772555001795E8|
|Brooklyn to Brooklyn          |1.739447214800117E8 |
|Queens to Queens              |1.1470684719999422E8|
|Manhattan to Queens           |1.0173842820999901E8|
|Queens to Manhattan           |8.603540026000012E7 |
|Manhattan to Unknown          |8.010710242000188E7 |
|Bronx to Bronx                |7.414622575999315E7 |
|Manhattan to Brooklyn         |6.79904755900002E7  |
|Brooklyn to Manhattan         |6.3176161049999915E7|
|Brooklyn to Queens            |5.045416242999964E7 |
|Queens to Brooklyn            |4.729286535999981E7 |
|Queens to Unknown             |4.629299990000313E7 |
|Bronx to Manhattan            |3.2486325169999994E7|
|Manhattan to Bronx            |3.1978763449999884E7|
|Manhattan to EWR              |2.3750888619999688E7|
|Brooklyn to Unknown           |1.0848827569999939E7|
|Bronx to Unknown              |1.0464800209999718E7|
|Bronx to Queens               |1.0292266500000013E7|
|Queens to Bronx               |1.0182898729999995E7|
|Staten Island to Staten Island|9686862.450000165   |
|Brooklyn to Bronx             |5848822.56          |
|Bronx to Brooklyn             |5629874.410000006   |
|Brooklyn to EWR               |3292761.7099999995  |
|Brooklyn to Staten Island     |2417853.819999999   |
|Staten Island to Brooklyn     |2265856.4599999976  |
|Manhattan to Staten Island    |2223727.370000001   |
|Staten Island to Manhattan    |1612227.72          |
|Queens to EWR                 |1192758.6600000001  |
|Staten Island to Unknown      |891285.8100000016   |
|Queens to Staten Island       |865603.3800000001   |
+------------------------------+--------------------+
```

3.4 SUPPOSE YOU WERE ONE OF THE STAKEHOLDERS, FOR EXAMPLE, EITHER THE DRIVER, CEO OF THE BUSINESS, OR STOCKBROKER, ETC, WHAT DO YOU FIND (I.E., INSIGHTS) FROM THE PREVIOUS THREE RESULTS? HOW DO THE FINDINGS HELP YOU MAKE STRATEGIES OR MAKE DECISIONS?

From above three results it is observable that Manhattan, Brooklyn and Queens are the most famous pickup and drop-off boroughs. Even trips within these boroughs that is Manhattan to Manhattan, Brooklyn to Brooklyn and Queens to Queens are the most profitable trips.

Companies might consider increasing the supply of drivers in these **three** boroughs, to utilize the opportunity of the **high demand**.

The businesses should develop marketing campaigns, offers on fairs or reduced fairs in boroughs like Bronkx, Statens Island and EWR. The businesses could expand their services in these boroughs to meet its specific needs, which may have different peak hours or trip purposes.

To ensure an adequate supply of drivers during peak times, the companies could offer incentives or bonuses to drivers who work during the busiest hours or in the most in-demand locations.

Task 4 Average of data

4.1 Average Driver Total Pay by Time of Day

Steps and APIs used:

1. *Data Loading and Preparation (Covered in Task 1 Report)*

   - Created `SparkSession`.
   - Rideshare data and taxi zone lookup data are loaded from S3 using textFile and converted into DataFrames.
   - Cleaned taxi_df using **`regexp_replace`**.
   - DataFrames are joined based on pickup and dropoff locations using join and columns are renamed.
   - The date column is formatted to "yyyy-MM-dd" format using date_format.

2. *Casting data types:*

   - Changed the "trip_length" and "driver_total_pay" columns to float types using **`cast("float")`** to ensure accurate computation for average calculation.

3. *Grouping and aggregation:*

   - Used **`groupBy`** on the "time_of_day" column to group trips by their time period.
   - Calculated the average of "driver_total_pay" using **`avg`** within each group and stored the result in a new column named "**`average_drive_total_pay`**".
   - The result is stored in **`avg_drive_pay_time_day`** with columns "time_of_day" and "average_drive_total_pay".

4. *Sorting:*

   - Used **`orderBy`** to sort **`avg_drive_pay_time_day`** by the "average_drive_total_pay" column in **descending** order, showing the time periods with the highest average earnings first.

5. *API's used (**same in all the parts**):*

   - orderBy ()

- agg()
- avg()
- join()
- withColumn()
- cast(“float”)
- groupBy()

Challenges encountered:

- Ensuring consistent data types (e.g., float for numerical values) for aggregation operations like calculating averages.

Knowledge gained:

- Learned changing datatypes of columns using cast.
- It highlights using avg for calculating average values within groups.

Output:

```
2024-04-03 13:34:36,123 INFO codegen.CodeGenerator: Code generated in 10.831996 ms
+-----------+----------------------+
|time_of_day|average_drive_total_pay|
+-----------+----------------------+
|  afternoon|     21.212428755696347|
|      night|      20.08743800270718|
|    evening|     19.777427701749236|
|    morning|      19.63333279274821|
+-----------+----------------------+
```

4.2 AVERAGE TRIP LENGTH BY TIME OF DAY

This task calculates the average trip length for each time of day period to identify the period with the highest average trip distance.

Steps and APIs used:

1. *Grouping and aggregation:*
   - Used **`groupBy`** on the "time_of_day" column to group trips by their time period.
   - Computed the average "trip_length" using **`avg`** within each group and stored the result in a new column named "average_trip_length".
   - The result is stored in **`avg_trip_length_time_day`** with columns "time_of_day" and "average_trip_length".
2. *Sorting:*
   - Used **`orderBy`** to sort **`avg_trip_length_time_day`** by the "average_trip_length" column in **descending** order, showing the time periods with the highest average trip lengths first.

Output:

```
2024-04-03 13:41:15,470 INFO codegen.CodeGenerator: Code generated in 9.733817 ms
+-----------+-------------------+
|time_of_day|average_trip_length|
+-----------+-------------------+
|      night|  5.323984802300155|
|    morning|  4.927371866627282|
|  afternoon|  4.861410525884578|
|    evening|  4.484750367647451|
+-----------+-------------------+
```

4.3 AVERAGE EARNING PER MILE BY TIME OF DAY

Steps and APIs used:

1. *Joining DataFrames:*
   - Used the **`join`** function to combine the **`avg_drive_pay_time_day`** and **`avg_trip_length_time_day`** DataFrames based on the matching "time_of_day" column.
   - This creates a new DataFrame named joined_df that includes columns from both DataFrames.
2. *Calculating average earning per mile:*
   - Added a new column named **`"average_earning_per_mile"`** to joined_df using **`withColumn()`** API.
   - The value in this column is calculated by **dividing** the **`"average_drive_total_pay"`** by **`"average_trip_length".`** This provides the average earnings per unit of distance traveled during each time period.
3. *Selecting and displaying results:*
   - Used the **`select`** function to choose only the "time_of_day" and "average_earning_per_mile" columns for the final result.
   - Displayed the results using **`show(truncate=False)`** to display output without truncating the time-of-day values.

Knowledge gained:

- Learned joining DataFrames based on a shared column.
- It highlights creating new columns with calculations involving existing columns.

Output:

```
2024-04-03 13:52:07,242 INFO codegen.CodeGenerator: Code generated in 11.431698 ms
+-----------+------------------------+
|time_of_day|average_earning_per_mile|
+-----------+------------------------+
|afternoon  |4.3634308690349854      |
|night      |3.7730081412006795      |
|morning    |3.9845445653743488      |
|evening    |4.409928330553616       |
+-----------+------------------------+
```

4.4 WHAT DO YOU FIND (I.E., INSIGHTS) FROM THE THREE RESULTS? HOW DO THE FINDINGS HELP YOU MAKE STRATEGIES OR MAKE DECISIONS?

From the three results it is observable that:

- The **'afternoon'** period recorded the **second highest** "**average driver total pay**" compared to other times of the day and accounts **second least "average trip length"** thereby resulting into **highest** "**average earning per mile**".

- From this we can infer that **afternoon** period is a **high peak fare time** and **more profitable** period for the **company and drivers** as for **short trip length** the **pay is high**.

- Alternatively, **another** reason for **high average driver total pay** and **less average trip length** for **afternoon** period can be the number of rides in **afternoon period** being **less**, might be because of **high fares**. So here, **companies** can adjust the fares to **increase** trips in afternoon.

- The **'night'** time has the "**highest average trip length'**. But accounts to **least "average earning per mile"**. Thus this is the **period** where the **companies** should **increase** the **fares** to make **profit**.

Task 5 Finding anomalies

5.1 AVERAGE WAITING TIME PER DAY IN JANUARY

Steps and APIs used:

1. *Data Loading and Preparation (Covered in Task 1 Report)*

   - Created `SparkSession`.
   - Rideshare data and taxi zone lookup data are loaded from S3 using textFile and converted into DataFrames.
   - Cleaned taxi_df using **`regexp_replace`**.
   - DataFrames are joined based on pickup and dropoff locations using join and columns are renamed.
   - The date column is formatted to "yyyy-MM-dd" format using date_format

2. *Filtering data:*

   - The **`filter`** function with **`month("date") == 1`** selects only those rides that occurred in January.
   - This creates a new DataFrame **`jan_df`** containing January data.

3. *Grouping and aggregation:*

   - **`groupBy`** is used on the **`dayofmonth("date")`** column, which extracts the **day of the month** from the "date" column and assigns it an alias "day".
   - Calculated the average "**request_to_pickup**" time using **`avg`** within each day group and stored the result in a new column named "**`average_waiting_time`**".
   - The result is stored in **`avg_waiting_time_per_day`**, showing average waiting time for each day in **January**.

4. *Sorting and Displaying first 20 rows:*

   - **`orderBy("day")`** to sort **`avg_waiting_time_per_day`** by the "day" column, ensuring results are presented chronologically.
   - Printed first 20 rows of resulting dataframe in output using **`avg_waiting_time_jan.show(20).`**

5. *Saving to S3 Bucket:(similar to Task 2)*

- **S3 Resource Object:**
  - Created a resource object for my S3 bucket using the boto3 library. This object allowed me to interact with S3 for storing and retrieving data.
- **Current Date and Time:**
  - It retrieves the current date and time using **`datetime.now()`** and formats it using **`strftime`**. This timestamp is used to create a unique filename for the output file.
- **Coalescing DataFrame:**
  - The **`coalesce(1)`** function reduces the number of output files written to S3 by merging smaller files into a single file.
- **Output Path:**
  - Defined the S3 path to save the DataFrame. It combines the bucket name, a folder structure (`aman_<date_time>`)**,** and the filename (trips_per_business_month_profit.csv) with the .csv extension.
- **Saving as CSV:**
  - The **`write.csv`** method of the DataFrame is used to specify:
    - path: The S3 path constructed earlier.
    - mode: "overwrite" to replace existing files with the same name.
    - header: True to include the column names in the CSV file.

6. *Copying to Local Storage:*

- Copied avg_waiting_time_jan.csv from S3 bucket to my local file system in output folder using command :
  `ccc method bucket cp -r bkt: aman_29-03-2024_02:14:32/avg_waiting_time_jan.csv/ output`

7. Visualization using Python Script(Similar to previous task):

- *Libraries:*
  - Using **`pandas`** for data manipulation and **`matplotlib.pyplot`** for creating histograms.
- *Reading CSV:*
  - Downloaded and read the csv file(`avg_waiting_time_jan.csv`) into a pandas DataFrame (`task5`).
- *Histogram Creation:*
  - Genearted a Histogram (barplot) for visulaising average waiting time of January using Matplotlib library function **`plt.bar()`** with “day” on **x_axis** and **average waiting time** on “y-axis”.

8. API's used (same in all parts):

- filter()
- month()
- groupBy()
- agg()
- avg()
- dayofmonth()
- orderBy()
- filter()
- write.csv()

Challenges encountered:

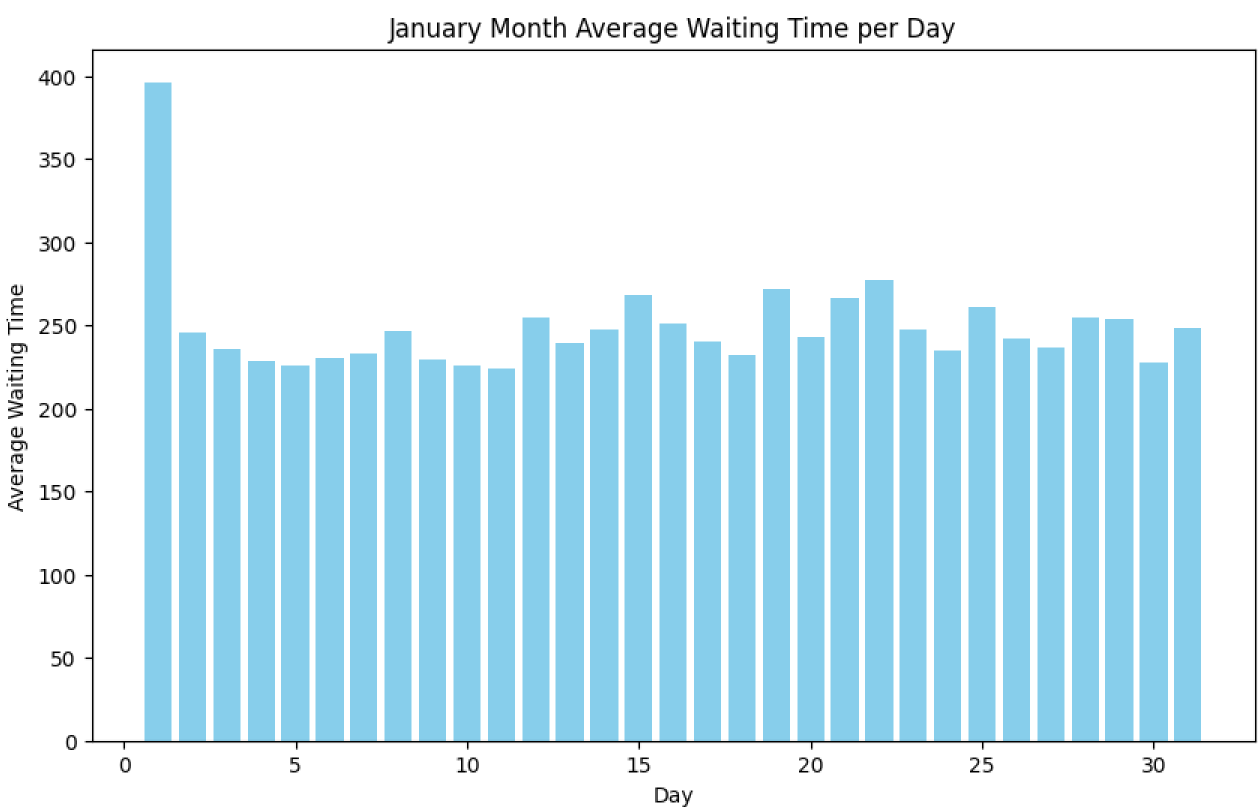
The challenge of saving the datafrme as a csv file into our system, which was solved while completing task2**.**

Knowledge gained:

- Learned data filtering methodology based on specific conditions (month).
- It highlights using dayofmonth to extract the day of the month from a date column.

Output:

```
2024-03-29 02:14:32,502 INFO codegen.CodeGenerator: Code generated in 8.100081 ms
+---+--------------------+
|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
```



5.2 Days with High Average Waiting Time

Steps and APIs used:

1. *Filtering results:*
   - Applied **`filter`** function on **`avg_waiting_time_jan`** dataframe (created in Task 5.1) to select only days where the "average_waiting_time" is greater than 300 seconds.
   - This filtered DataFrame, named **`day_greater_300_secs`**, highlights days with potentially high waiting times.
2. *Displaying results:*
   - Displayed the content of **day_greater_300_secs**, which includes the result of filter operation, "day" and "average_waiting_time" columns using **`day_greater_300_secs.show()`**.

Output:

```
2024-03-28 12:00:33,066 INFO codegen.CodeGenerator: Code generated in 10.11413 ms
+---+--------------------+
|day|average_waiting_time|
+---+--------------------+
|  1|   396.5318744409635|
+---+--------------------+
```

5.3 Why was the average waiting time longer on these day(s) compared to other days?

The average wait time exceeds 300 only on 1st January , likely due to New Year celebrations. Since many individuals attend parties to ring in the New Year, there is heightened demand on this day, resulting in longer wait times compared to other days of January, where the average waiting time typically hovers around 270. This presents a reasonable anomaly in comparison to the rest of the month.

Task 6 Filtering Data

6.1 TRIP COUNTS WITH CONDITIONS

Steps and APIs used:

1. Data Loading and Preparation (Covered in Task 1 Report)

- Created `SparkSession`.
- Rideshare data and taxi zone lookup data are loaded from S3 using textFile and converted into DataFrames.
- Cleaned taxi_df using **`regexp_replace`**.
- DataFrames are joined based on pickup and dropoff locations using join and columns are renamed.
- The date column is formatted to "yyyy-MM-dd" format using date_format

2. Grouping and aggregation:

- Used **`groupBy`** on both "Pickup_Borough" and "time_of_day" columns.
- Count the number of trips using **`count`** within each group and store the result in a new column named "trip_count".
- This creates **`borough_time_of_day_trip_counts`**, showing trip counts for each combination of pickup borough and time of day.

3. Filtering:

- Used **`filter`** with a compound condition to **`select`** only rows where "trip_count" is greater than 0 but less than 1,000.
- This ensures the focus on trip counts within the specified range.

4. Displaying results:

- Displayed the filtered DataFrame filtered_trip_counts using **`show(truncate=False)`** to print without truncating column values, providing a clear view of pickup boroughs, times of day, and their corresponding trip counts.

*5. API's used (**same in all parts**):*

- groupBy()
- agg()

- filter()
- select()
- show()
- count()
- col()

Knowledge gained:

- Learned filtering DataFrames based on multiple conditions applied to columns.
- Also about grouping by two columns for the required analysis.

Output:

```
2024-03-28 14:25:48,190 INFO codegen.CodeGenerator: Code generated in 14.860048 ms
+--------------+-----------+----------+
|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       |
+--------------+-----------+----------+
```

6.2 EVENING TRIP COUNTS BY PICKUP BOROUGH

Steps and APIs used:

1. *Filtering data:*
   - Reused the **`borough_time_of_day_trip_counts`** DataFrame created in 6.1.
   - Applied **`filter`** to select only rows where the "time_of_day" column value is "evening".
   - This ensures the focus on trips that occurred in the evening.
2. *Displaying results:*
   - Displayed the filtered Dataframe **`filtered_evening_trip_counts`** using **`show(truncate=False)`** to print dataframe without truncating column values, providing a clear view of pickup boroughs and their corresponding evening trip counts.

Knowledge gained:

- Learned how to filter DataFrames based on a specific value in a column.

Output:

```
2024-03-28 14:34:31,664 INFO codegen.CodeGenerator: Code generated in 15.684413 ms
+--------------+-----------+----------+
|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       |
+--------------+-----------+----------+
```

6.3 TRIPS FROM BROOKLYN TO STATEN ISLAND

Steps and APIs used:

1. *Data filtering:*
    - Performed **`filter`** on **`nyc_df1`** to select trips where the "Pickup_Borough" is "Brooklyn" and the "Dropoff_Borough" is "Staten Island".
    - This ensures the focus on trips that meet the specified origin and destination criteria.
2. *Selecting columns:*
    - Used **`select`** to choose only the "Pickup_Borough", "Dropoff_Borough", and "Pickup_Zone" columns for the results.
3. *Displaying and counting results:*
    - Used **`show(10)`** to display the first 10 rows of the filtered DataFrame **`selected_df_final`**.
    - Used **`count`** to determine the total number of trips that match the criteria and print the result using Python's **`print`** function.

Knowledge gained:

- Learned selecting specific columns using select for the output

Output:

```
2024-04-05 02:39:21,355 INFO scheduler.DAGScheduler: Job 4 finished: count at NativeMethodAccessorImpl.java:0, took 530.074293 s
Count of trips from Brooklyn to Staten Island:  69437
2024-04-05 02:39:21,387 INFO server.AbstractConnector: Stopped Spark@7574f7b1{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
```

```
2024-04-05 02:09:23,464 INFO codegen.CodeGenerator: Code generated in 15.135113 ms
+--------------+---------------+--------------------+
|Pickup_Borough|Dropoff_Borough|         Pickup_Zone|
+--------------+---------------+--------------------+
|      Brooklyn|  Staten Island|     Columbia Street|
|      Brooklyn|  Staten Island|     Columbia Street|
|      Brooklyn|  Staten Island|     Columbia Street|
|      Brooklyn|  Staten Island|     Columbia Street|
|      Brooklyn|  Staten Island|     Columbia Street|
|      Brooklyn|  Staten Island|Marine Park/Mill ...|
|      Brooklyn|  Staten Island|Marine Park/Mill ...|
|      Brooklyn|  Staten Island|Marine Park/Mill ...|
|      Brooklyn|  Staten Island|Marine Park/Mill ...|
|      Brooklyn|  Staten Island|Marine Park/Mill ...|
+--------------+---------------+--------------------+
only showing top 10 rows
```

Task 7 Route Analysis

7.1 TOP 10 POPULAR ROUTES ANALYSIS

Steps and APIs used:

1. Data Loading and Preparation (Covered in Task 1 Report)

- Created `SparkSession`.
- Rideshare data and taxi zone lookup data are loaded from S3 using textFile and converted into DataFrames.
- Cleaned taxi_df using **`regexp_replace`**.
- DataFrames are joined based on pickup and dropoff locations using join and columns are renamed.
- The date column is formatted to "yyyy-MM-dd" format using date_format

2. Creating a new 'Route' column:

- Used **`concat_ws`** to create a new column using **`withColumn`**, named "Route" that **combines** the "**`Pickup_zone`**" and **`"Dropoff_zone"`** columns **separated** by **`"to"`**.
- This creates a unique identifier for each route.

3. Calculating trip counts by route and business:

- **`filter`** to separate trips for Uber and Lyft businesses.
- Within each business group, used **`groupBy`** on "Route" and **`count`** to calculate the number of trips for each route.
- This results in two DataFrames: route_profits_uber (Uber trips) and route_profits_lyft (Lyft trips).

4. Merging DataFrames:

- Used **`join`** on the "Route" column to merge **`route_profits_uber`** and **`route_profits_lyft`** DataFrames.
- This creates a DataFrame **`route_profits_merged`** that includes "Route", "uber_count", and "lyft_count" columns.

5. Calculating total trip count:

- Created **`total_count`** column using **`withColumn`**. This column is a result of the addition of "uber_count", and "lyft_count" columns.
- Sorted the resulting Dataframe **`route_profits_final`** using **`orderBy(by = “total_cont”, ascending = Flase)`** in descending order on the basis of “**`total_count`**” column

6. Displaying results:

- Displayed the first 10 rows of route_profits_merged using **`show(10, truncate=False)`** to print without truncating column names, providing insights into the top 10 routes and their Uber/Lyft trip counts.

7. API's used:

- concat_ws()
- withColumn()
- filter()
- groupBy()
- agg()
- count()
- join()
- orderBy()

Knowledge gained:

- Learned how to create a new column by concatenating existing columns.
- It highlights grouping and aggregation using business type and route.
- Also learned how to merge DataFrames based on a shared column.

Output:

```
2024-04-05 03:22:41,916 INFO codegen.CodeGenerator: Code generated in 12.875848 ms
+----------------------------------------+----------+----------+-----------+
|Route                                   |uber_count|lyft_count|total_count|
+----------------------------------------+----------+----------+-----------+
|JFK Airport to NA                       |253211    |46        |253257     |
|East New York to East New York          |202719    |184       |202903     |
|Borough Park to Borough Park            |155803    |78        |155881     |
|LaGuardia Airport to NA                 |151521    |41        |151562     |
|Canarsie to Canarsie                    |126253    |26        |126279     |
|South Ozone Park to JFK Airport         |107392    |1770      |109162     |
|Crown Heights North to Crown Heights North|98591   |100       |98691      |
|Bay Ridge to Bay Ridge                  |98274     |300       |98574      |
|Astoria to Astoria                      |90692     |75        |90767      |
|Jackson Heights to Jackson Heights      |89652     |19        |89671      |
+----------------------------------------+----------+----------+-----------+
only showing top 10 rows
```

Task 8 Graph Processing

8.1 DEFINING STRUCTYPE OF VERTEX AND EDGE SCHEMA

Steps and API's used:

1. StructType Definition (**Spark SQL Structype API**):
   - `StructType` is used to define the schema for both vertices and edges.
   - Each schema is a list of StructField objects, specifying the **column name**, data type, and nullability.

2. Schema Definitions
   - *Vertex Schema (vertex_Schema):*
     - id *(StringType, Not Null):* Unique identifier for each vertex (taxi zone). This field cannot be null to ensure every zone has a distinct ID.
     - Borough *(StringType, Nullable):* Borough name where the taxi zone is located (e.g., "Manhattan", "Queens"). This field can be null if the information is unavailable.
     - Zone *(StringType, Nullable):* Zone identifier within the borough. This field can be null if the information is unavailable.
     - service_zone *(StringType, Nullable):* Service zone identifier associated with the taxi zone. This field can be null if the information is unavailable.
   - *Edge Schema (edge_Schema):*
     - src *(StringType, Not Null):* Source vertex ID (pickup location taxi zone). This field cannot be null to establish a starting point for the ride.
     - dst *(StringType, Not Null):* Destination vertex ID (dropoff location taxi zone). This field cannot be null to define the ride's endpoint.

Explanation:
1. The **vertexSchema** is established to define the structure of the vertices dataframe, which represents various locations or taxi zones.
2. The **edgeSchema** is defined to outline the structure of the edges dataframe, which captures the rideshare trips between locations.

Knowledge Gained:

- Learned defining schemas for vertices and edges with the Spark SQL **StructType API**.
- Learned how transform dataframes into a graph structure.

8.2 CONSTRUCT EDGES, VERTICES DATAFRAMES AND GRAPH

This section details the steps and APIs used to construct the vertices and edges DataFrames.

Steps and API's used:

1. *Vertices DataFrame:*
    - Selected the relevant columns from the taxi_df DataFrame using the **`select`** function. These columns represent the properties of each location (vertex) in the graph.

2. *Edges DataFrame:*
    - Selected the pickup_location and dropoff_location columns from the ride_df DataFrame using the **`select`** function. These columns define the connections (edges) between locations in the graph.

3. *GraphFrame Creation:*
    - The selected vertices and edges DataFrames were used to construct a GraphFrame named **graph_main** using the **`GraphFrame function`**. This **GraphFrame** combines the vertex and edge information for **graph** analysis.

4. *Displaying Sample Data:*
    - Utilized the **`show`** function on both **`graph.vertices`** and **`graph.edges`** to display the first 10 rows of each DataFrame.

5. *API's Used:*
    - select, alias
    - GraphFrame

Output:

```
2024-03-31 21:36:00,749 INFO codegen.CodeGenerator: Code generated in 22.418454 ms
+---+-------------+--------------------+------------+
| id|      Borough|                Zone|service_zone|
+---+-------------+--------------------+------------+
|  1|          EWR|      Newark Airport|         EWR|
|  2|       Queens|         Jamaica Bay|   Boro Zone|
|  3|        Bronx|Allerton/Pelham G...|   Boro Zone|
|  4|    Manhattan|       Alphabet City| Yellow Zone|
|  5|Staten Island|       Arden Heights|   Boro Zone|
|  6|Staten Island|Arrochar/Fort Wad...|   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
```

Vertices Dataframe

```
2024-04-04 18:33:56,591 INFO codegen.CodeGenerator: Code generated in 21.367195 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
```

Edges Dataframe

8.3 CREATING A GRAPH USING VERTICES AND EDGES

Steps and API's used:

1. Finding Connected Vertices:

- Employed the **`triplets`** operation on the graph object with **`distinct()`** method. This operation transforms the GraphFrame into a DataFrame representation, where each row represents a connection (edge) in the graph and returns only unique rows.

2. Displaying Sample Data:

- Used the **`show`** function on the resulting DataFrame from the triplets operation using **`graph.triplets.distinct().show(10, truncate=False).`** Kept the truncate parameter to False to ensure all data is displayed.

4. API's used:

- triplets
- show

Challenges:

- Finding right method to get output same as given.
- Got stuck with same similar rows, then resolved it by making use of distinct method.

Output:

```
2024-04-05 01:40:01,584 INFO codegen.CodeGenerator: Code generated in 12.22405 ms
+--------------------------------------------------------+----------+-------------------------------------------------------------+
|src                                                     |edge      |dst                                                          |
+--------------------------------------------------------+----------+-------------------------------------------------------------+
|[133, Brooklyn, Kensington, Boro Zone]                  |[133, 124]|[124, Queens, Howard Beach, Boro Zone]                       |
|[65, Brooklyn, Downtown Brooklyn/MetroTech, Boro Zone]  |[65, 124] |[124, Queens, Howard Beach, Boro Zone]                       |
|[66, Brooklyn, DUMBO/Vinegar Hill, Boro Zone]           |[66, 124] |[124, Queens, Howard Beach, Boro Zone]                       |
|[133, Brooklyn, Kensington, Boro Zone]                  |[133, 7]  |[7, Queens, Astoria, Boro Zone]                              |
|[93, Queens, Flushing Meadows-Corona Park, Boro Zone]   |[93, 7]   |[7, Queens, Astoria, Boro Zone]                              |
|[34, Brooklyn, Brooklyn Navy Yard, Boro Zone]           |[34, 234] |[234, Manhattan, Union Sq, Yellow Zone]                      |
|[256, Brooklyn, Williamsburg (South Side), Boro Zone]   |[256, 234]|[234, Manhattan, Union Sq, Yellow Zone]                      |
|[223, Queens, Steinway, Boro Zone]                      |[223, 200]|[200, Bronx, Riverdale/North Riverdale/Fieldston, Boro Zone]|
|[47, Bronx, Claremont/Bathgate, Boro Zone]              |[47, 200] |[200, Bronx, Riverdale/North Riverdale/Fieldston, Boro Zone]|
|[230, Manhattan, Times Sq/Theatre District, Yellow Zone]|[230, 200]|[200, Bronx, Riverdale/North Riverdale/Fieldston, Boro Zone]|
+--------------------------------------------------------+----------+-------------------------------------------------------------+
only showing top 10 rows

*********************************************************************
```

8.4 COUNTING CONNECTED VERTICES WITH THE SAME BOROUGH AND SAME SERVICE ZONE

Steps and API's used:

1. Extracting Graph Triples::

- Utilized the **`find`** operation on the graph object to specify a pattern for finding connected vertices that is, **`(a)-[e]->(b)`** which represents an edge **`e`** connecting vertex **`a`** to vertex **`b`**.

2. Filtering by Borough and Service Zone:

- Applied a **`filter(a.Borough = b.Borough AND a.service_zone = b.service_zone )`** on the result of the **`find`** operation. This filter expression ensures that only connections between vertices with the same borough and service zone are retained.

3. Selecting and Renaming Columns:

- Selected the desired columns (**`a.id, b.id, a.Borough, and a.service_zone`**) from the filtered DataFrame.

4. Displaying Sample Data:

- Used the **`show`** function on the resulting DataFrame from the select operation using **`borough_service_vertices.distinct().show(10, truncate=False).`** Kept the truncate parameter to False to ensure all data is displayed and **`distinct`** method to get only unique values from **`borough_service_vertices`**.

5. Counting and Displaying Results:

- Used **`count()`** function on **`borough_service_vertices`** to get count of total connections and stored as **`total_connected_vertices`** including duplicates. Printed it out on terminal using print command as **`print(f"Count of total connected vertices with the same Borough and service zone is: {total_connected_vertices}")`**

4. API’s used:

- find
- filter
- select
- distinct
- show

Knowledge gained:

- Learned how to traverse a graph by using the **`find`** operation to identify vertices connected by specific edge patterns.
- Use of **`filter`** to refine the search results and focus on specific pattern relevant to the task.
- Learned to remove duplicates using the **`distinct`** operation.

Output:

```
2024-04-04 19:10:06,225 INFO scheduler.DAGScheduler: Job 5 finished: showString at NativeMethodAccessorImpl.java:0, took 527.748352
2024-04-04 19:10:06,245 INFO codegen.CodeGenerator: Code generated in 11.262327 ms
+---+---+-------------+------------+
|id |id |Borough      |service_zone|
+---+---+-------------+------------+
|252|19 |Queens       |Boro Zone   |
|206|245|Staten Island|Boro Zone   |
|131|207|Queens       |Boro Zone   |
|111|178|Brooklyn     |Boro Zone   |
|186|90 |Manhattan    |Yellow Zone |
|64 |95 |Queens       |Boro Zone   |
|121|95 |Queens       |Boro Zone   |
|144|158|Manhattan    |Yellow Zone |
|37 |65 |Brooklyn     |Boro Zone   |
|164|68 |Manhattan    |Yellow Zone |
+---+---+-------------+------------+
only showing top 10 rows
```

```
2024-04-05 01:45:53,619 INFO scheduler.TaskSchedulerImpl: Killing all running tasks in stage 15: Stage finished
2024-04-05 01:45:53,619 INFO scheduler.DAGScheduler: Job 5 finished: count at NativeMethodAccessorImpl.java:0, took 351.519515 s
Count of total connected vertices with the same Borough and service zone is: 46886992
*******************************************************************
2024-04-05 01:45:53,643 INFO server.AbstractConnector: Stopped Spark@9fe4561{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
2024-04-05 01:45:53,645 INFO ui.SparkUI: Stopped Spark web UI at http://task8-spark-app-40843c8eabe2119e-driver-svc.data-science-ec23
```

8.5 PERFORMING PAGE RANKING ON THE GRAPH DATAFRAME

Steps and API's used:

1. Applying PageRank:

- Utilized the **`pageRank`** function on the graph_main object. This function implements the PageRank algorithm, which iteratively calculates a score (PageRank) for each vertex, reflecting its relative importance based on the incoming links from other vertices

2. Setting Parameters:

- Set two parameters for the PageRank algorithm **`resetProbability (0.17) and tol(0.01)`** as intructed.
- **`resetProbability (0.17):`** This parameter represents the probability of a random jump to any vertex during the PageRank calculation. We used a value of 0.17.
- **`tol (0.01):`** This parameter defines the tolerance threshold for convergence. The algorithm iterates until the PageRank values for all vertices change by less than this threshold. We used a tolerance of 0.01.

3. Sorting and Displaying Results:

- Used the **`sort`** function on the result of the **`pageRank`** operation after selecting vertices.
- Sorted the vertices in descending order based on their calculated PageRank values (pagerank) using **`sort('pagerank', ascending=False)`** function.
- Then selected the columns id and pagerank using **`select("id" , "pagerank")`** and displayed the first 5 rows using **`show`** with truncate=False.

4. API's used:

- pageRank
- sort
- select
- show

Knowledge gained:

- This task demonstrated understanding of PageRank algorithm which is a valuable technique for ranking vertices in a graph based on their connectivity and influence within the network.

Output:

```
2024-04-04 18:00:59,349 INFO scheduler.DAGScheduler: Job 40 finished: showString at NativeMethodAccessorImpl.java:0, toc
2024-04-04 18:00:59,366 INFO codegen.CodeGenerator: Code generated in 11.955213 ms
+---+------------------+
|id |pagerank          |
+---+------------------+
|265|11.105433344107194|
|1  |5.4718454249167205|
|132|4.551132572067087 |
|138|3.5683223416564713|
|61 |2.6763973653412996|
+---+------------------+
only showing top 5 rows
```