NYC RIDE SHARE DATASET

SUNANTHA KANNAN
[230841125]

05-04-2024

—

ECS765P-BIG DATA PROCESSING

—

Table of CONTENTS

Table of CONTENTS

TASK 1: Merging Datasets

This task details the process of merging rideshare data with taxi zone information and further processing in a Spark environment to enhance the dataset for analysis. The process includes data loading, cleaning, joining, and transforming operations using PySpark's DataFrame and RDD.

Initializing Spark Session:

- Initialized a Spark session using the SparkSession object. Fetched environment variables related to the S3 bucket, such as the data repository bucket, endpoint URL, access key ID, secret access key, and bucket name. Configured Hadoop settings for the Spark session using the hadoopConf object.

Helper Functions:

- good_ride_line` and `good_taxi_line` are the functions for splitting the lines with comma from the data and also it checks for the correct number of fields in each row.

Configuration and S3 Access:

The code snippet configures Apache Spark to access an S3 storage by giving necessary parameters:

```
# shared read-only object bucket containing datasets
s3_data_repository_bucket = os.environ['DATA_REPOSITORY_BUCKET']
s3_endpoint_url = os.environ['S3_ENDPOINT_URL']+':'+os.environ['BUCKET_PORT']
s3_access_key_id = os.environ['AWS_ACCESS_KEY_ID']
s3_secret_access_key = os.environ['AWS_SECRET_ACCESS_KEY']
s3_bucket = os.environ['BUCKET_NAME']

hadoopConf = spark.sparkContext._jsc.hadoopConfiguration()
hadoopConf.set("fs.s3a.endpoint", s3_endpoint_url)
hadoopConf.set("fs.s3a.access.key", s3_access_key_id)
hadoopConf.set("fs.s3a.secret.key", s3_secret_access_key)
hadoopConf.set("fs.s3a.path.style.access", "true")
hadoopConf.set("fs.s3a.connection.ssl.enabled", "false")
```

- **Extract Environment Variables**: Retrieving S3 credentials and connection details from the environment.
- **Initialize Hadoop Configuration**: Using Spark's context to access Hadoop's configuration for S3 access.
- **Set Endpoint**: Specified the URL of the S3 service for Spark to connect.
- **Set Access Credentials**: Provide the access key and secret key for authentication with S3.
- **Enable Path-Style Access**: Use path-style URLs for S3 buckets, which is necessary for certain S3-compatible services.

- **Disable SSL**: Turn off SSL/TLS for connections, useful in non-production environments or if the S3 service doesn't support SSL.

The script configures the Spark context to access data stored in an S3 bucket. This involves setting various Hadoop configuration properties like endpoint, access key, secret key, and others.

#1}. Load rideshare_data.csv and taxi_zone_lookup.csv:

Process:

Data is loaded from S3 into RDDs, filtered to remove bad lines, split into columns, and then converted into DataFrames for easy manipulation.

After loading, data is cleaned to remove unwanted characters (e.g., double quotes).

APIs Used:

spark.sparkContext.textFile: Loads data from a text file into an RDD

filter(): Applies a boolean condition to each element of the RDD to filter out unwanted records.

map(): Transforms each element of the RDD, used here to split each line into a tuple of values.

toDF(): Converts an RDD into a DataFrame foranalysis**.**

regexp_replace: A function used within withColumn to replace specific characters in DataFrame columns, crucial for cleaning textual data.

VISUALIZATION:

Loaded Ride share data:

```
+--------+-------------+---------------+---------------+-----------+----------------+
|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|
+--------+-------------+---------------+---------------+-----------+----------------+----------------+-----------------+-----------+---
--------+-------------+---------------+---------------+-----------+----------------+
|    Uber|          151|            244|           4.98|      226.0|           761.0|            19.0|            780.0|    morning|168
4713600|        22.82|          13.69|           9.13|      63.18|            2.75|
|    Uber|          244|             78|           4.35|      197.0|          1423.0|           120.0|           1543.0|    morning|168
4713600|        24.27|           19.1|           5.17|      44.56|            4.39|
|    Uber|          151|            138|           8.82|      171.0|          1527.0|            12.0|           1539.0|    morning|168
4713600|        47.67|          25.94|          21.73|      60.68|            2.94|
|    Uber|          138|            151|           8.72|      260.0|          1761.0|            44.0|           1805.0|    morning|168
4713600|        45.67|          28.01|          17.66|      55.86|            3.21|
|    Uber|           36|            129|           5.05|      208.0|          1762.0|            37.0|           1799.0|    morning|168
4713600|        33.49|          26.47|           7.02|      52.97|            5.24|
|    Uber|          138|             88|          12.64|      230.0|          2504.0|            29.0|           2533.0|    morning|168
4713600|        69.15|          40.15|           29.0|      57.06|            3.18|
|    Uber|          200|            138|           14.3|      337.0|          1871.0|           120.0|           1991.0|    morning|168
4713600|        62.35|          37.48|          24.87|      67.77|            2.62|
|    Uber|          182|            242|           1.05|      177.0|           323.0|            30.0|            353.0|    morning|168
4713600|         10.3|           6.54|           4.76|       66.7|            6.23|
|    Uber|          248|            242|           0.57|      195.0|           169.0|             2.0|            171.0|    morning|168
4713600|          8.1|           5.54|           2.56|     116.63|            9.72|
|    Uber|          242|             20|           2.08|      308.0|           822.0|           120.0|            942.0|    morning|168
4713600|         14.8|          10.61|           4.19|      40.55|             5.1|
+--------+-------------+---------------+---------------+-----------+----------------+----------------+-----------------+-----------+---
--------+-------------+---------------+---------------+-----------+----------------+
only showing top 10 rows

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)
```

Loaded Taxi data:

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

root
 |-- LocationID: string (nullable = true)
 |-- Borough: string (nullable = true)
 |-- Zone: string (nullable = true)
 |-- service_zone: string (nullable = true)
```

#2} Joining the Datasets:

PROCESS:

- **Initial Join**:
  - Performed an inner join between **ride_df** and **taxi_df** on the **pickup_location** and **LocationID** columns. The resulting DataFrame **joined_df1** contains all columns from both DataFrames where the condition matches.
  - We then rename the columns **Borough**, **Zone**, and **service_zone** to **Pickup_Borough**, **Pickup_Zone**, and **Pickup_service_zone** to distinguish them as pickup-related attributes.
  - The **LocationID** column from the taxi DataFrame is dropped since it's redundant after the join.
- **Second Join**:
  - A second join is executed on **joined_df1** and **taxi_df**, this time matching **dropoff_location** to **LocationID**. This enriches the **joined_df1** DataFrame with the dropoff location details.
  - Columns are similarly renamed to **Dropoff_Borough**, **Dropoff_Zone**, and **Dropoff_service_zone** to indicate they are related to the dropoff location.
  - Again, we drop the **LocationID** column post-join to eliminate duplication.
- **Display Joined DataFrame**:
  - The final joined DataFrame **nyc_df** is shown with **show(10)** to visualize the first 10 records and **printSchema()** to display the schema.

API's USED:

- **join()**: To perform a join operation between two DataFrames.
- **withColumnRenamed()**: To rename columns in the DataFrame.
- **drop()**: To drop specified columns from the DataFrame.

VISUALIZATION:

Joined ride share and taxi zone lookup dataframes.

```
2024-04-03 17:42:20,126 INFO codegen.CodeGenerator: Code generated in 23.330603 ms
+--------+---------------+----------------+-----------+-----------------+----------------+------------------+----------
---------+-----------+----------+--------------+-----------------+----------------+-----------+----------------+-------
-------+-----------+-------------------+---------------+------------+--------------------+
|business|pickup_location|dropoff_location|trip_length|request_to_pickup|total_ride_time|on_scene_to_pickup|on_scene_t
o_dropoff|time_of_day|      date|passenger_fare|driver_total_pay|rideshare_profit|hourly_rate|dollars_per_mile|Pickup_
Borough|Pickup_Zone|Pickup_service_zone|Dropoff_Borough|Dropoff_Zone|Dropoff_service_zone|
+--------+---------------+----------------+-----------+-----------------+----------------+------------------+----------
---------+-----------+----------+--------------+-----------------+----------------+-----------+----------------+-------
-------+-----------+-------------------+---------------+------------+--------------------+
|    Uber|            169|             125|      12.52|             71.0|          1832.0|              17.0|
   1849.0|    morning|1677974400|         43.77|            39.64|            4.13|      77.18|            3.17|
  Bronx| Mount Hope|          Boro Zone|      Manhattan|   Hudson Sq|         Yellow Zone|
|    Uber|            169|             125|      12.75|            250.0|          1487.0|             121.0|
   1608.0|      night|1684281600|          53.7|            30.74|           22.96|      68.82|            2.41|
  Bronx| Mount Hope|          Boro Zone|      Manhattan|   Hudson Sq|         Yellow Zone|
|    Uber|            169|             125|      12.79|            270.0|          1754.0|              35.0|
   1789.0|    morning|1675987200|         53.57|             32.1|           21.47|      64.59|            2.51|
  Bronx| Mount Hope|          Boro Zone|      Manhattan|   Hudson Sq|         Yellow Zone|
|    Uber|            169|             125|       12.5|            170.0|          1961.0|              14.0|
   1975.0|    morning|1675468800|         42.87|            33.68|            9.19|      61.39|            2.69|
  Bronx| Mount Hope|          Boro Zone|      Manhattan|   Hudson Sq|         Yellow Zone|
|    Uber|            169|             125|      12.55|            164.0|          1988.0|              25.0|
   2013.0|    morning|1680912000|         43.72|            35.51|            8.21|      63.51|            2.83|
  Bronx| Mount Hope|          Boro Zone|      Manhattan|   Hudson Sq|         Yellow Zone|
|    Uber|            169|             125|      11.67|            781.0|          1758.0|             120.0|
   1878.0|    morning|1680566400|         46.19|            31.85|           14.34|      61.05|            2.73|
  Bronx| Mount Hope|          Boro Zone|      Manhattan|   Hudson Sq|         Yellow Zone|
|    Uber|            169|             125|      12.45|            326.0|          3980.0|              11.0|
   3991.0|    morning|1683676800|         56.53|            53.77|            2.76|       48.5|            4.32|
  Bronx| Mount Hope|          Boro Zone|      Manhattan|   Hudson Sq|         Yellow Zone|
|    Uber|            169|             125|      12.28|            346.0|          3559.0|             121.0|
   3680.0|    morning|1683676800|          54.3|             49.6|             4.7|      48.52|            4.04|
  Bronx| Mount Hope|          Boro Zone|      Manhattan|   Hudson Sq|         Yellow Zone|
|    Uber|            169|             125|      13.17|            146.0|          2678.0|               6.0|
   2684.0|    morning|1680307200|         59.07|            43.22|           15.85|      57.97|            3.28|
  Bronx| Mount Hope|          Boro Zone|      Manhattan|   Hudson Sq|         Yellow Zone|
|    Uber|            169|             125|      13.33|            279.0|          2904.0|             121.0|
   3025.0|  afternoon|1680307200|         52.28|            44.81|            7.47|      53.33|            3.36|
  Bronx| Mount Hope|          Boro Zone|      Manhattan|   Hudson Sq|         Yellow Zone|
+--------+---------------+----------------+-----------+-----------------+----------------+------------------+----------
```

#3} Convert the UNIX timestamp to the "yyyy-MM-dd" format:

PROCESS:

The UNIX timestamp in the date column of nyc_df is converted to a human-readable date format "yyyy-MM-dd" using the **date_format** and **from_unixtime** functions. The DataFrame is updated in place with the newly formatted date column.

API's USED:

- **withColumn()**: created a new column "date"
- **from_unixtime()**: To convert UNIX timestamp to a timestamp type.
- **date_format()**: To format the timestamp into the desired date format. "yyyy-MM-dd"
- **to_date()** : To convert the data type of "date" column from string to date data type

VISUALIZATION:

The below screenshot is the dataframe with changed timestamp in this format "yyyy-MM-dd".

```
ok 0.625438 s
2024-04-03 17:34:39,119 INFO codegen.CodeGenerator: Code generated in 16.83016 ms
+----------+
|      date|
+----------+
|2023-05-13|
|2023-05-13|
|2023-05-13|
|2023-05-14|
|2023-05-14|
|2023-01-27|
|2023-01-27|
|2023-01-27|
|2023-01-27|
|2023-01-27|
+----------+
only showing top 10 rows

2024-04-03 17:34:39,146 INFO server.AbstractConnector: Stopped Spark@48fad223{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
2024-04-03 17:34:39,149 INFO ui.SparkUI: Stopped Spark web UI at http://task1-spark-app-cc81848ea5000bc8-driver-svc.da
ta-science-ec23863.svc:4040
```

#4} Number of rows:

Process:

Counting the number of rows in the final dataset to check for total dataset size.

APIs Used:

- **count()**: To count the number of rows in the DataFrame.
- **printSchema()**: To print the schema of the DataFrame.

VISUALIZATION:

```
6.629486 s
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)

2024-04-03 17:25:32,436 INFO server.AbstractConnector: Stopped Spark@711520cd{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
2024-04-03 17:25:32,437 INFO ui.SparkUI: Stopped Spark web UI at http://task1-spark-app-d9d4e08ea4f1edce-driver-svc.da
ta-science-ec23863.svc:4040
```

TASK 1 CHALLENGES AND SOLUTIONS:

- String columns contained unwanted double quotes. The `regexp_replace` function effectively cleaned these string literals from the data.

- While merging datasets and performing transformations, managing the schema can be challenging. Frequent use of **printSchema()** helps in understanding the current structure of the DataFrame.

TASK 1 KNOWLEDGE AND INSIGHTS:

Knowing the structure and size of the Data Frame helps in assessing the completeness of **the** data merging process and planning subsequent analysis steps. This task demonstrates the importance of data preparation, including cleaning, merging, and transformation for data analysis. Through this process, I had good understanding of the dataset, more detailed and accurate analyses of rideshare patterns across New York City.

Execution:

Execute the spark application	**Stream the logs of the drive container**
ccc create spark task1.py -s	oc logs -f task2-spark-app-driver

The Spark session is stopped using **stop()** method.

TASK 2: Aggregation of Data

The task involves aggregating New York City taxi and rideshare data to provide insights into trip counts, platform profits, and driver earnings, with an emphasis on monthly performance and differences between businesses.

PROCESS AND API's USED:

Data Conversion and Preparation:

- The **date** column is converted to a standard date format using **to_date**, and numerical fields **rideshare_profit** and **driver_total_pay** are cast to float types with **cast("float")**
- A **month** column is extracted from **date** using the **month** function, and aggregate by month

#1} Counting the number of trips for each business in each month
Aggregation of Trip Counts:

Data is grouped by **business** and **month** using **groupBy**, followed by counting the number of trips using **count("*")**.

#2} Calculate the platform's profits (rideshare_profit field) for each business in each month
Profit Calculation:

Similarly, the dataset is grouped by **business** and **month**, and **sum("rideshare_profit")** calculates the total platform profits for each group.

#3} Calculate the driver's earnings (driver_total_pay field) for each business in each month
Driver Earnings Calculation:

Again, grouping by **business** and **month**, the code uses **sum("driver_total_pay")** to calculate total driver earnings for each group.

Output and Data Storage:

- The aggregated DataFrames are shown in the output using **show()**.
- An S3 resource is created using **boto3.resource** to store the result in an S3 bucket.
- The path for storing the CSV file is prepared using the current date and time.
- The **coalesce(1)** function combines all DataFrame partitions into a single file to avoid multiple output files.
- The aggregated DataFrame is saved to a CSV file in the S3 bucket using **write.csv**.
- Commands are provided (commented out) to copy the result file from the S3 bucket to a local directory for further analysis.

```
# After the program execution, execute this below command to copy file from S3 bucket locally in the hub
    #ccc method bucket ls - to check our file exists or not and copy the file name
    #ccc method bucket cp -r bkt:your_directory_name output_directory_name

# Once the output directory stored in our system, we can able to plot histogram using matplotlib
```

VISUALIZATION

Code Snippet for storing data frames in S3 bucket:

```
# Created resource object for S3 bucket for storing trips data in S3 bucket
    bucket = boto3.resource(
        "s3",
        endpoint_url="http://" + s3_endpoint_url,
        aws_access_key_id=s3_access_key_id,
        aws_secret_access_key=s3_secret_access_key,
    )

    # To specify date and time in the file name
    now = datetime.now()
    date_time = now.strftime("%d-%m-%Y_%H:%M:%S")


    # To combine all the partition documents as single file
    output_df = trips_per_business_month.coalesce(1)

    # creating the S3 path for storing the result
    output_path1 = "s3a://" + s3_bucket + "/task2_" + date_time + "/trips_per_business_month.csv"
    output_path2 = "s3a://" + s3_bucket + "/task2_" + date_time + "/trips_per_business_month_profit.csv"
    output_path3 = "s3a://" + s3_bucket + "/task2_" + date_time + "/trips_per_business_month_driver_pay.csv"

    # Save the DataFrame to CSV on S3
    output_df.write.csv(path=output_path1, mode="overwrite", header=True)
    output_df.write.csv(path=output_path2, mode="overwrite", header=True)
    output_df.write.csv(path=output_path3, mode="overwrite", header=True)


    spark.stop()


# After the program execution, execute this below command to copy file from S3 bucket locally in the hub
    #ccc method bucket ls - to check our file exists or not and copy the file name
    #ccc method bucket cp -r bkt:your_directory_name output_directory_name

# Once the output directory stored in our system, we can able to plot histogram using matplotlib
```

Viewing data frame in s3 bucket:

Execution:

ccc method bucket ls

ccc method bucket ls task2output_03-04-2024_16:10:35/

```
jovyan@jupyter-ec23863:~/teaching_material/ECS765P/bigdata$ ccc method bucket ls
                           PRE bigdata_28-03-2024_14:12:44/
                           PRE olympic18-03-2024_19:24:14/
                           PRE olympic18-03-2024_19:25:14/
                           PRE olympic19-03-2024_12:36:30/
                           PRE sorted_counts_18-03-2024_20:11:09/
                           PRE sorted_counts_18-03-2024_20:12:51/
                           PRE sunantha_26-03-2024_11:43:41/
                           PRE sunantha_26-03-2024_12:08:48/
                           PRE sunantha_26-03-2024_16:45:46/
                           PRE task2(1)_28-03-2024_18:13:00/
                           PRE task2_28-03-2024_15:49:54/
                           PRE task2_28-03-2024_21:55:44/
                           PRE task2output_03-04-2024_16:10:35/
2024-03-18 20:10:47    2959972 sherlock.txt
jovyan@jupyter-ec23863:~/teaching_material/ECS765P/bigdata$ ccc method bucket ls task2output_03-04-2024_16:10:35/
                           PRE trips_per_business_month.csv/
                           PRE trips_per_business_month_driver_pay.csv/
                           PRE trips_per_business_month_profit.csv/
```

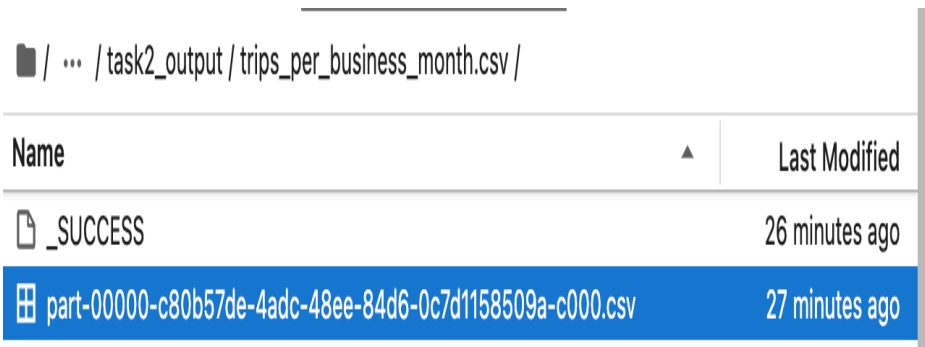
Copying data from S3 Bucket and Saving it locally:

Execution:

ccc method bucket cp -r bkt:task2output_03-04-2024_16:10:35/ task2_output

```
jovyan@jupyter-ec23863:~/teaching_material/ECS765P/bigdata$ ccc method bucket cp -r bkt:task2output_03-04-2024_16:10:35/ task2_output
download: s3://object-bucket-ec23863-a8c006ac-6de2-43e5-aeac-947029a1d739/task2output_03-04-2024_16:10:35/trips_per_business_month_driver_pay.c
sv/part-00000-3d3cc3c0-929f-45f4-9afa-e90a73996e13-c000.csv to task2_output/trips_per_business_month_driver_pay.csv/part-00000-3d3cc3c0-929f-45
f4-9afa-e90a73996e13-c000.csv
download: s3://object-bucket-ec23863-a8c006ac-6de2-43e5-aeac-947029a1d739/task2output_03-04-2024_16:10:35/trips_per_business_month_profit.csv/_
SUCCESS to task2_output/trips_per_business_month_profit.csv/_SUCCESS
download: s3://object-bucket-ec23863-a8c006ac-6de2-43e5-aeac-947029a1d739/task2output_03-04-2024_16:10:35/trips_per_business_month_driver_pay.c
sv/_SUCCESS to task2_output/trips_per_business_month_driver_pay.csv/_SUCCESS
download: s3://object-bucket-ec23863-a8c006ac-6de2-43e5-aeac-947029a1d739/task2output_03-04-2024_16:10:35/trips_per_business_month.csv/_SUCCESS
 to task2_output/trips_per_business_month.csv/_SUCCESS
download: s3://object-bucket-ec23863-a8c006ac-6de2-43e5-aeac-947029a1d739/task2output_03-04-2024_16:10:35/trips_per_business_month_profit.csv/p
art-00000-cc4bf579-3872-4e7e-9ebe-f60296ac34cb-c000.csv to task2_output/trips_per_business_month_profit.csv/part-00000-cc4bf579-3872-4e7e-9ebe-
f60296ac34cb-c000.csv
download: s3://object-bucket-ec23863-a8c006ac-6de2-43e5-aeac-947029a1d739/task2output_03-04-2024_16:10:35/trips_per_business_month.csv/part-000
00-c80b57de-4adc-48ee-84d6-0c7d1158509a-c000.csv to task2_output/trips_per_business_month.csv/part-00000-c80b57de-4adc-48ee-84d6-0c7d1158509a-c
000.csv
```

Output file created in JupyterHub locally after the execution of above command:



	A	B	C
1	business	month	trip_count
2	Lyft	4	8173
3	Uber	5	14276372
4	Uber	4	13995860
5	Lyft	3	8444
6	Uber	2	13280761
7	Lyft	1	6887
8	Uber	3	14554308
9	Uber	1	13579077
10	Lyft	2	6491
11	Lyft	5	9491
12			

/ ··· / task2_output / trips_per_business_month_profit.csv /

Name	Last Modified
_SUCCESS	16 minutes ago
part-00000-cc4bf579-3872-4e7e-9ebe-f60296ac34cb-c000.csv	17 minutes ago

	A	B	C
1	business	month	Platform Profit
2	Lyft	4	-90197.13001759537
3	Uber	5	1.6313361550055736E8
4	Uber	4	1.502698201941708E8
5	Lyft	3	-99403.93998675235
6	Uber	2	1.3062880563633616E8
7	Lyft	1	-72633.3500049822
8	Uber	3	1.5207287641912192E8
9	Uber	1	1.3319711162465 69E8
10	Lyft	2	-70064.72000297531
11	Lyft	5	-107719.21000343934
12			

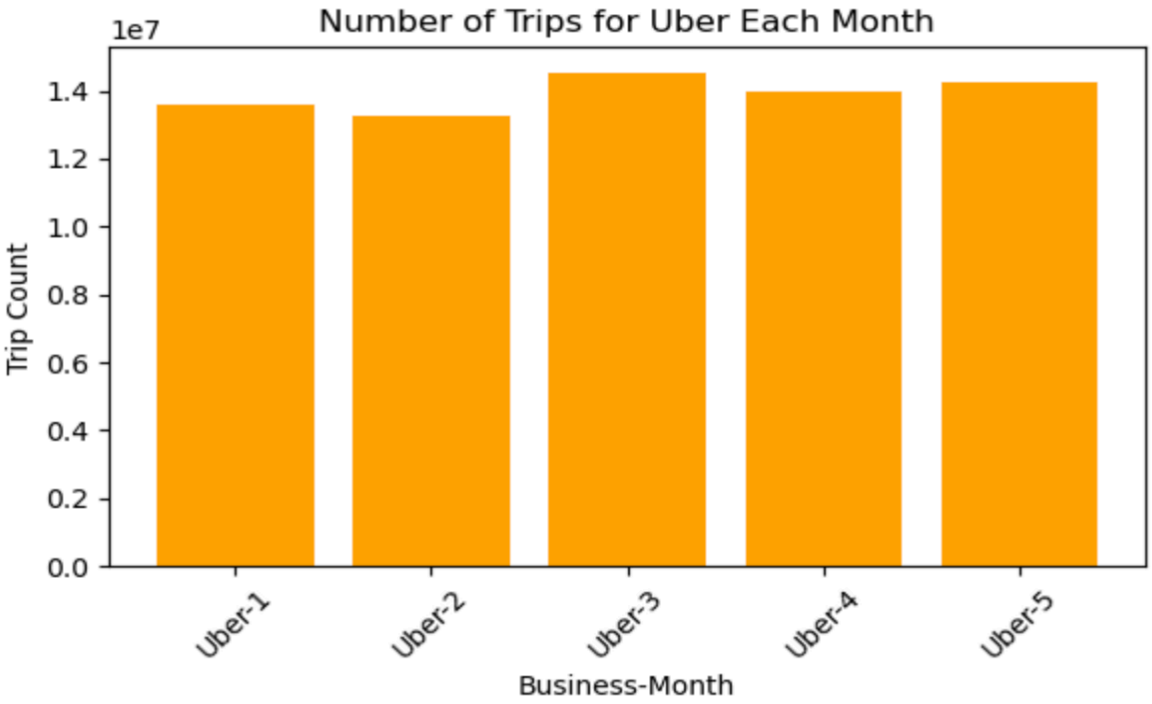
/ ··· / task2_output / trips_per_business_month_driver_pay.csv /

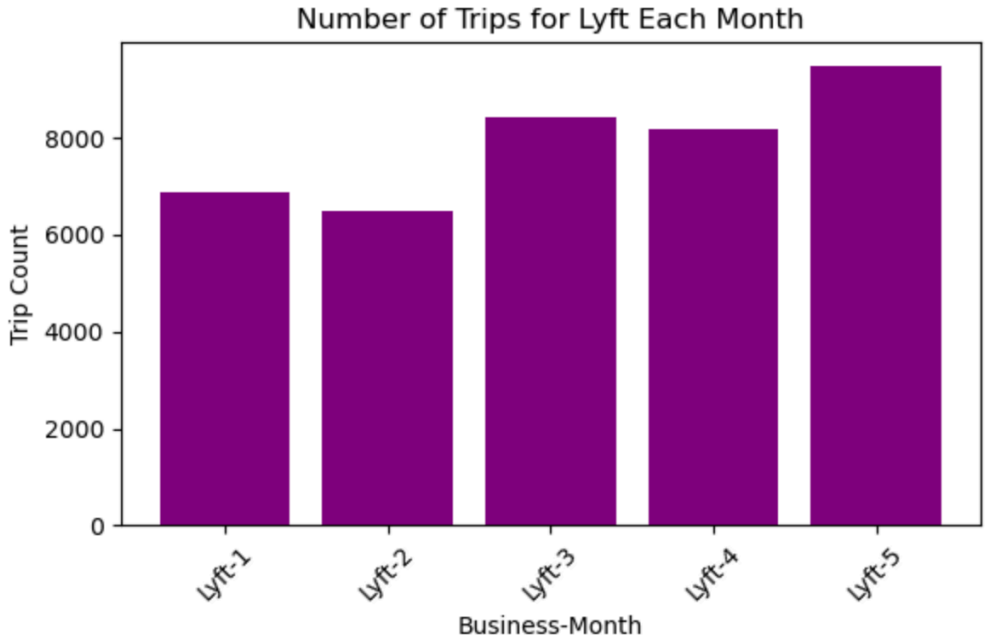
Name	Last Modified
_SUCCESS	9 minutes ago
part-00000-3d3cc3c0-929f-45f4-9afa-e90a73996e13-c000.csv	9 minutes ago

	A	B	C
1	business	month	Driver Earnings
2	Lyft	4	297815.3794941902
3	Uber	5	3.1300511453083175E8
4	Uber	4	2.950689272036143E8
5	Lyft	3	310276.54944992065
6	Uber	2	2.521559770842398 7E8
7	Lyft	1	239932.2593984604
8	Uber	3	2.959584959978399E8
9	Uber	1	2.5025348066162878E8
10	Lyft	2	234875.5296087265
11	Lyft	5	360408.08971881866
12			

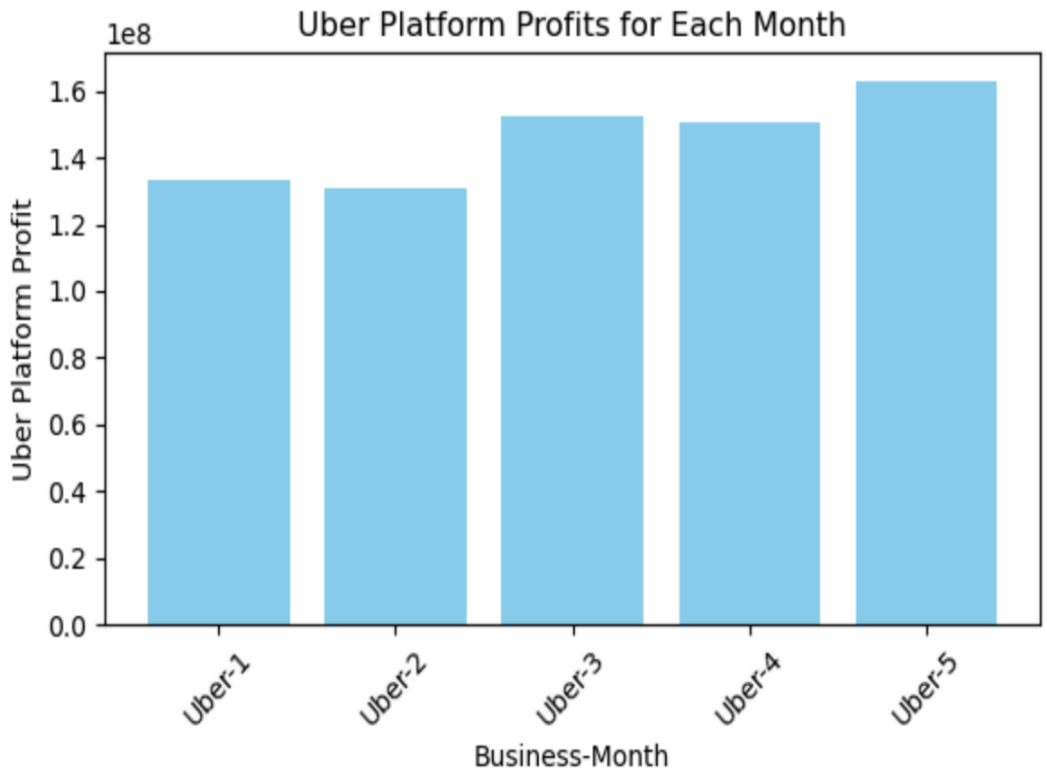
After the execution and saving of the data locally from s3 bucket:
Histograms or bar charts can be created using **matplotlib** to visualize the trip counts, platform profits, and driver earnings per business for each month.

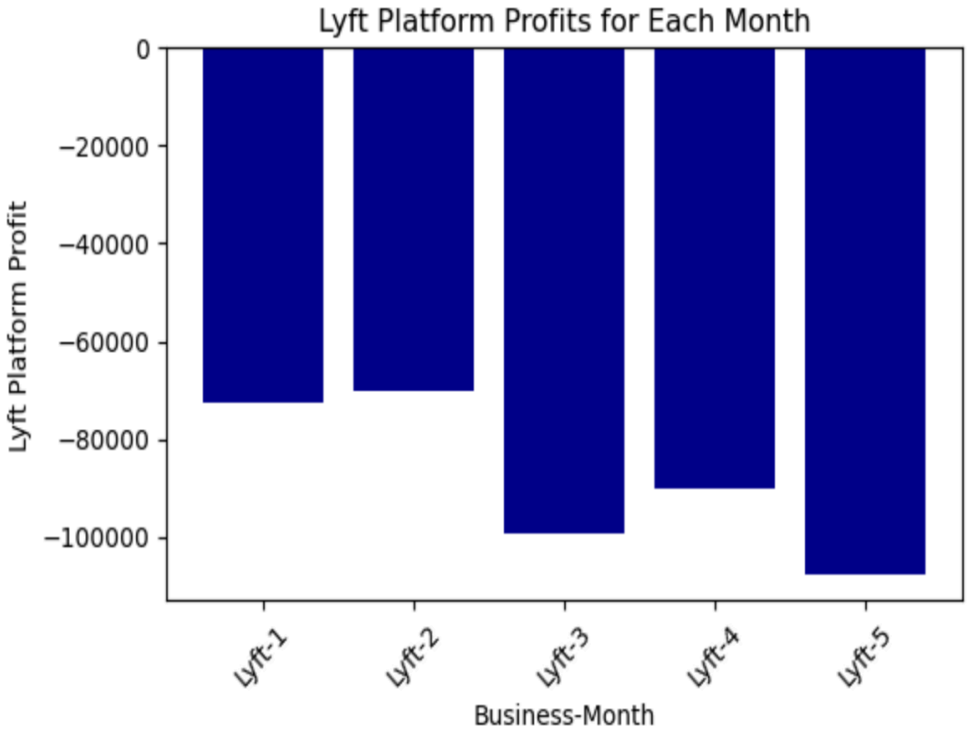
Visualization of Trip Counts (Uber and Lyft):



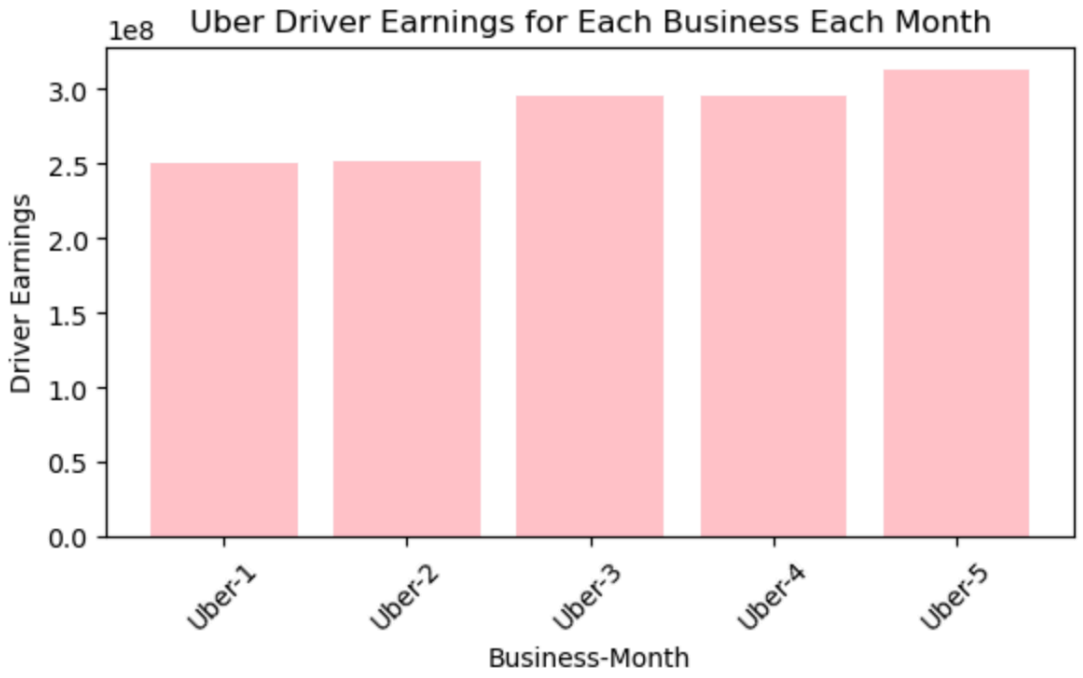


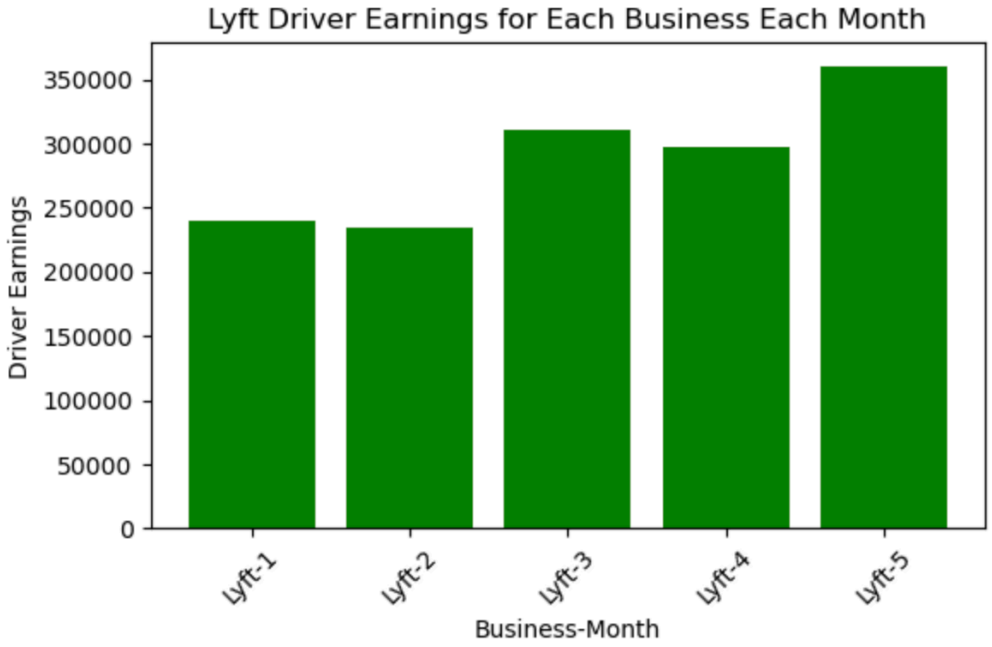
Visualization of Platform Profits (Uber and Lyft):





Visualization of Driver Earnings (Uber and Lyft):





#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?

Analyzing the taxi trip data shows that drivers could earn more by working more during busy months identified by high trip counts. A CEO of a rideshare company might use profit trends to decide when to spend more on advertising or change prices. An investor could look at these same trends to figure out the best times to buy or sell shares in the rideshare company, based on how well they think the company will do. This practical use of trip data can help each person make better decisions for their own work or investments.

TASK 2 CHALLENGES AND SOLUTIONS

Data Type Consistency:

The need to convert data types to make sure accurate calculations. The **cast** function helped to perform correct calculations by converting the respective field to float .

Single File Output:

Spark outputs each partition as a separate file, which is not good for storage and visualization. Using **coalesce(1)** to consolidate all partitions into a single file.

Data Export:

Writing the processed data to an external storage system. Configuring S3 bucket access and use Spark's **write** function to store the results.

TASK 2 KNOWLEDGE AND INSIGHTS:

- Had knowledge on transforming and aggregating large datasets efficiently within PySpark, specifically handling date and integer data types.
- Learned how to calculate monthly trip counts, platform profits, and driver earnings for rideshare businesses, providing insights into trends and financial performance.
- Developed skills in integrating PySpark with cloud storage (S3) to save and retrieve data, paving the way for future data analysis and visualization tasks.

Execution

Execute the spark application	Stream the logs of the drive container
ccc create spark task2.py -s	oc logs -f task2-spark-app-driver

The Spark session is stopped using **stop()** method.

TASK3: Top-K Processing

This task covers the process and insights gained from analyzing taxi trip data to determine the top popular pickup and drop-off boroughs in New York City each month, as well as identifying the top 30 most profitable routes.

#1}: Top 5 Popular Pickup Boroughs Each Month

PROCESS AND API's USED:

1. Group and Aggregate: The data was grouped by `Pickup_Borough` and `month` using the `groupBy` function, and then aggregated to count the number of trips using the `count("*")` function. This step is important to summarize the data based on unique combinations of boroughs and months.

2.Window Specification: A window specification (`windowSpec`) was defined using `Window.partitionBy("month").orderBy(desc("trip_count"))` to segment the data by month and order within each segment by the trip count in descending order. This step is essential for ranking boroughs within each month based on their popularity.

3. Top 5 Filtering: The `row_number` window function was applied over the defined window specification to each row in a partitioned group, and then filtered to retain rows where the row number was less than or equal to 5, by selecting the top 5 boroughs by trip count for each month.

4. **Sorting**: The results were then ordered by month and trip count in ascending and descending order respectively, to prepare the data for visualization and interpretation.

VISUALIZATION:

```
2024-03-28 17:37:22,472 INFO codegen.CodeGenerator: Code generated in 17.193323 ms
2024-03-28 17:37:22,508 INFO codegen.CodeGenerator: Code generated in 18.438018 ms
+--------------+-----+----------+
|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} Top 5 Popular Dropoff Boroughs Each Month

The steps and APIs used here same like #1, with the only difference on `Dropoff_Borough` instead of `Pickup_Borough`.

VISUALIZATION:

```
2024-03-28 17:49:53,380 INFO codegen.CodeGenerator: Code generated in 13.133583 ms
2024-03-28 17:49:53,401 INFO codegen.CodeGenerator: Code generated in 10.008569 ms
+---------------+-----+----------+
|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|
+---------------+-----+----------+

2024-03-28 17:49:53,430 INFO server.AbstractConnector: Stopped Spark@d37a266{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
2024-03-28 17:49:53,432 INFO ui.SparkUI: Stopped Spark web UI at http://task3-spark-app-c641f18e86279a5e-driver-svc.
40
```

3} Top 30 Earnest Routes

PROCESS AND API's USED

1.Concatenation of Boroughs and casting driver total pay: The `**concat_ws**` function was used to create a new column `Route` by concatenating `Pickup_Borough` and `Dropoff_Borough`, which helped in identifying unique routes. Then casting the data type of the column "driver_total_pay" to float data type for calculation using **cast("float").**

2. Group and Aggregate for Profit: The `**groupBy**` function was then used on `Route`, with aggregation using `sum("driver_total_pay")` to find the total profit per route.

3. Top 30 Selection: The `**orderBy**` function sorted the routes by `total_profit` in descending order, with the `limit(30)` function used to select the top 30 routes.

VISUALIZATION:

```
2024-04-03 15:14:28,889 INFO codegen.CodeGenerator: Code generated in 14.056486 ms
+------------------------------+--------------------+
|Route                         |total_profit        |
+------------------------------+--------------------+
|Manhattan to Manhattan        |3.338577255269214E8 |
|Brooklyn to Brooklyn          |1.7394472146560934E8|
|Queens to Queens              |1.1470684718672623E8|
|Manhattan to Queens           |1.0173842820749661E8|
|Queens to Manhattan           |8.60354002623074E7  |
|Manhattan to Unknown          |8.010710241910338E7 |
|Bronx to Bronx                |7.41462257518282E7  |
|Manhattan to Brooklyn         |6.799047559133713E7 |
|Brooklyn to Manhattan         |6.31761610487396E7  |
|Brooklyn to Queens            |5.045416242985292E7 |
|Queens to Brooklyn            |4.729286535949615E7 |
|Queens to Unknown             |4.629299989943378E7 |
|Bronx to Manhattan            |3.2486325168083E7   |
|Manhattan to Bronx            |3.1978763449171744E7|
|Manhattan to EWR              |2.375088861989542E7 |
|Brooklyn to Unknown           |1.0848827571691632E7|
|Bronx to Unknown              |1.0464800210008174E7|
|Bronx to Queens               |1.0292266499867737E7|
|Queens to Bronx               |1.0182898730611693E7|
|Staten Island to Staten Island|9686862.448563514   |
|Brooklyn to Bronx             |5848822.56057135    |
|Bronx to Brooklyn             |5629874.409598887   |
|Brooklyn to EWR               |3292761.709862232   |
|Brooklyn to Staten Island     |2417853.819514513   |
|Staten Island to Brooklyn     |2265856.459864326   |
|Manhattan to Staten Island    |2223727.3698619604  |
|Staten Island to Manhattan    |1612227.7201343775  |
|Queens to EWR                 |1192758.6599292755  |
|Staten Island to Unknown      |891285.8100587726   |
|Queens to Staten Island       |865603.3800287247   |
+------------------------------+--------------------+

2024-04-03 15:14:28,918 INFO server.AbstractConnector: Stopped Spark@6d9ce99f{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
```

#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 the driver perspective:

Monthly Trip Counts:

Certain boroughs like Manhattan and Brooklyn show consistently higher trip counts, especially in specific months. I might focus my driving hours in these boroughs during peak months to maximize passenger pickups.

Drop-off Data:

Similar to pickups, drop-offs are also higher in Manhattan and Brooklyn, indicating these areas have a steady flow of passengers. Ensuring I'm available in these areas could lead to reduced waiting times for the next fare and potentially higher earnings.

Profitable Routes:

Routes like Brooklyn to Bronx and Staten Island to Brooklyn show significant profits. I could prioritize these routes or position myself to take advantage of such trips when they come up on the dispatch system.

TASK 3 CHALLENGES AND SOLUTIONS:

Complex Window Functions: Understanding and applying window functions can be complex. Reviewing documentation and examples helped to understand better.

String Manipulation: Creating the `Route` column required careful concatenation and handling of strings. Ensuring data consistency before this step was crucial.

TASK 3 KNOWLEDGE AND INSIGHTS:

- This task gives the use of PySpark to analyze taxi trip data, focusing on identifying top pickup and dropoff boroughs each month and determining the most profitable routes.

- From a knowledge standpoint, it provides understanding of data aggregation, window functions, and string manipulation in PySpark for extracting meaningful insights from large datasets.

- Key learnings include how to group and aggregate data based on specific criteria, apply window specifications to rank results within partitions, and concatenate strings to create unique route identifiers for further analysis.

Execution:

Execute the spark application	**Stream the logs of the drive container**
ccc create spark task3.py -s	oc logs -f task3-spark-app-driver

The Spark session is stopped using **stop()** method.

TASK 4: Average of Data

This task demonstrates how to process and analyze a dataset of New York City taxi trips (`nyc_df`) to check insights regarding driver earnings and trip lengths across different times of the day.

#1} Highest Average 'driver_total_pay' by Time of Day:

PROCESS AND API's USED:

1. **Data Type Conversion:** Converting the data types of `trip_length` and `driver_total_pay` to float using **`cast('float')`,** requires calculation on these columns.

2. **GroupBy and Aggregation:** Groups the data by `time_of_day using **groupBy()** and calculates the average `driver_total_pay` for each group with the help of **avg().** The result is ordered in descending order of average pay using **orderBy()** to find the time of day with the highest earnings.

VISUALIZATION:

```
2024-04-03 15:31:22,307 INFO codegen.CodeGenerator: Code generated in 11.878683 ms
+----------+----------------------+
|time_of_day|average_drive_total_pay|
+----------+----------------------+
|  afternoon|      21.21242875569636|
|      night|      20.08743800270718|
|    evening|     19.777427701749232|
|    morning|     19.633332792748213|
+----------+----------------------+
```

#2} Highest Average 'trip_length' by Time of Day

PROCESS AND API's USED:

Follows a similar process to 1 of task4, focusing on calculating and ordering the average `trip_length` by `time_of_day'.

VISUALIZATION:

```
2024-04-03 15:41:56,420 INFO codegen.CodeGenerator: Code generated in 18.0392 ms
2024-04-03 15:41:56,446 INFO codegen.CodeGenerator: Code generated in 10.512229 ms
+-----------+-------------------+
|time_of_day|average_trip_length|
+-----------+-------------------+
|      night|  5.323984802300154|
|    morning| 4.9273718666272845|
|  afternoon|   4.86141052588458|
|    evening|  4.484750367647451|
+-----------+-------------------+
```

#3}: Calculating Average Earnings Per Mile by Time of Day

PROCESS AND API's USED

1.Joining DataFrames: Joins the two previous results on `time_of_day` using **join()** to combine average pay and trip length data.

2. Calculating Earnings Per Mile: Creates a new column `average_earning_per_mile` using **withColumn**() by dividing the average total pay by the average trip length for each time of day.

VISUALIZATION:

```
2024-04-03 15:59:34,434 INFO scheduler.TaskSchedulerImpl: Killing all running tasks in stage 48: Stage finished
2024-04-03 15:59:34,434 INFO scheduler.DAGScheduler: Job 9 finished: showString at NativeMethodAccessorImpl.java:0, took 0.101860 s
+-----------+-----------------------+
|time_of_day|average_earning_per_mile|
+-----------+-----------------------+
|afternoon  |4.363430869034982      |
|night      |3.773008141200681      |
|morning    |3.984544565374343      |
|evening    |4.4099283305536146     |
+-----------+-----------------------+

2024-04-03 15:59:34,499 INFO server.AbstractConnector: Stopped Spark@711520cd{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
2024-04-03 15:59:34,501 INFO storage.BlockManagerInfo: Removed broadcast_14_piece0 on task4-spark-app-e229cc8ea4a5f222-driver-svc.data-science-ec23863.svc:7079 in memory (size: 26.2 KiB, free: 2004.2 MiB)
```

4} What do you find (i.e., insights) from the three results? How do the findings help you make strategies or make decisions?

1. **For Ride-Sharing Companies**:
    - Companies can make drivers to work during times when average trip lengths are longer, such as at night to be available for longer time to get more profits.
2. **For Drivers**:
    - Understanding when they can earn more and balance it with trip lengths can help drivers plan their schedules to maximize earnings.
    - Drivers might prefer working during times with higher average earnings per mile, or if trip lengths are longer at night, they may focus on those hours if they want fewer but longer and more profitable trips.
3. **For Commuters**:
    - Commuters can use this information to understand the best times for hailing rides, considering cost and trip length.

TASK 4 CHALLENGES AND SOLUTIONS:

Type Mismatch: The initial challenge was make sure the data types were compatible for aggregation. This was addressed by casting string columns to integers.

Consistency in Aggregation: Ensuring the aggregation method is consistently applied across different metrics.

TASK 4 IKNOWLEDGE AND INSIGHTS:

- Learned PySpark SQL functions in data aggregation, sorting, joining, and performing calculations across Data Frame columns. These functions are essential for manipulating and analysing large datasets to derive meaningful metrics and insights.

- The analysis gives insights, such as which times of day yield the highest average driver earnings, how trip lengths vary by time, and the efficiency of earnings per mile across different periods. These insights are valuable for drivers seeking to maximize income and for taxi companies planning optimal operation times.

Execution:

Execute the spark application	**Stream the logs of the drive container**
ccc create spark task4.py -s	oc logs -f task4-spark-app-driver

The Spark session is stopped using **stop()** method.

TASK 5: Finding Anomalies

This task processes a dataset (`nyc_df`) of New York City taxi trip data to identify anomalies in waiting times during January. It involves data preparation, extraction, and aggregation steps to pinpoint days with unusually long average waiting times.

#1}: Average Waiting Time in January

PROCESS AND APIS USED:

1.Date Conversion: The `**to_date**` function converts the `date` column to a date type using the specified format "yyyy-MM-dd". This step gives that date-related operations can be performed accurately.

2. Extract Month: The `**month**` function extracts the month part from the `date` column, for the need of filtering by month.

3. Filtering for January: Used `**filter**` to select records from January (`month("date") == 1`).

4. Grouping and Aggregation: Grouped data by day of the month **(`dayofmonth("date")`)** and calculates the average waiting time **(`avg("request_to_pickup")`).** The result is ordered by day to check daily changes.

VISUALIZATION:

Used the below code snippet to store “Average Waiting Time in January” data frames in S3 bucket and copying in the hub locally to create histogram.

```
# Created resource object for S3 bucket for storing average waiting time per day data in S3 bucket
bucket = boto3.resource(
    "s3",
    endpoint_url="http://" + s3_endpoint_url,
    aws_access_key_id=s3_access_key_id,
    aws_secret_access_key=s3_secret_access_key,
)

# To specify date and time in the file name
now = datetime.now()
date_time = now.strftime("%d-%m-%Y_%H:%M:%S")

# To combine all the partition documents as single file
output_df = avg_waiting_time_per_day.coalesce(1)

# creating the S3 path for storing the result
output_path = "s3a://" + s3_bucket + "/bigdata_" + date_time + "/waiting_time_per_day.csv"

# Save the DataFrame to CSV on S3
output_df.write.csv(path=output_path, mode="overwrite", header=True)

# After the program execution, execute this below command to copy file from S3 bucket locally in the hub
# ccc method bucket ls - to check our file exists or not and copy the file name
# ccc method bucket cp -r bkt:your_directory_name output_directory_name

# Once the output directory stored in our system, we can able to plot histogram using matplotlib
```

Viewing data frame in s3 bucket:

Execution:

ccc method bucket ls

ccc method bucket ls bigdata_20-03-2024_14:12:44/

Copying data from S3 Bucket and Saving it locally:

Execution:

ccc method bucket cp -r bkt: bigdata_20-03-2024_14:12:44/output_task5

```
jovyan@jupyter-ec23863:~/teaching_material/ECS765P/bigdata$ ccc method bucket ls
                           PRE bigdata_28-03-2024_14:12:44/
                           PRE olympic18-03-2024_19:24:14/
                           PRE olympic18-03-2024_19:25:14/
                           PRE olympic19-03-2024_12:36:30/
                           PRE sorted_counts_18-03-2024_20:11:09/
                           PRE sorted_counts_18-03-2024_20:12:51/
                           PRE sunantha_26-03-2024_11:43:41/
                           PRE sunantha_26-03-2024_12:08:48/
                           PRE sunantha_26-03-2024_16:45:46/
2024-03-18 20:10:47    2959972 sherlock.txt
jovyan@jupyter-ec23863:~/teaching_material/ECS765P/bigdata$ ccc method bucket ls bigdata_28-03-2024_14:12:44/
                           PRE waiting_time_per_day.csv/
jovyan@jupyter-ec23863:~/teaching_material/ECS765P/bigdata$ ccc method bucket cp -r bkt:bigdata_28-03-2024_14:12:44/ output_task5
download: s3://object-bucket-ec23863-a8c006ac-6de2-43e5-aeac-947029a1d739/bigdata_28-03-2024_14:12:44/waiting_time_per_day.csv/part-00000-73418
422-1703-451b-ad27-2a507cdbe9af-c000.csv to output_task5/waiting_time_per_day.csv/part-00000-73418422-1703-451b-ad27-2a507cdbe9af-c000.csv
download: s3://object-bucket-ec23863-a8c006ac-6de2-43e5-aeac-947029a1d739/bigdata_28-03-2024_14:12:44/waiting_time_per_day.csv/_SUCCESS to outp
ut_task5/waiting_time_per_day.csv/_SUCCESS
```

Output file created in JupyterHub locally after the execution of above command:

Folder has been created locally

/ output_task5 / waiting_time_per_day.csv /

Name	Last Modified
_SUCCESS	6 minutes ago
part-00000-73418422-17...	8 minutes ago

	A	B
1	day	average_waiting_time
2	1	396.5318744409635
3	2	246.05148716456986
4	3	235.68026834234155
5	4	228.85434668408274
6	5	226.08877381422872
7	6	230.35306927438575
8	7	233.25699185710533
9	8	246.41358687741243
10	9	229.265944341545
11	10	225.65276195086662
12	11	224.40468798627612
13	12	255.17599322195403
14	13	239.22308233638282
15	14	247.49345781069232
16	15	268.5346481777792
17	16	251.55102299494047
18	17	240.5772885527869
19	18	231.90770494488552
20	19	272.02203820618143
21	20	243.43761253646377
22	21	266.6804386133228
23	22	277.49287089443135
24	23	247.32448989998323
25	24	234.81737623786302
26	25	261.2912811176952
27	26	242.56764282565965
28	27	236.93431696586904

Drawing the histogram with 'days' on the x-axis and ‘average waiting time’ on the y-axis

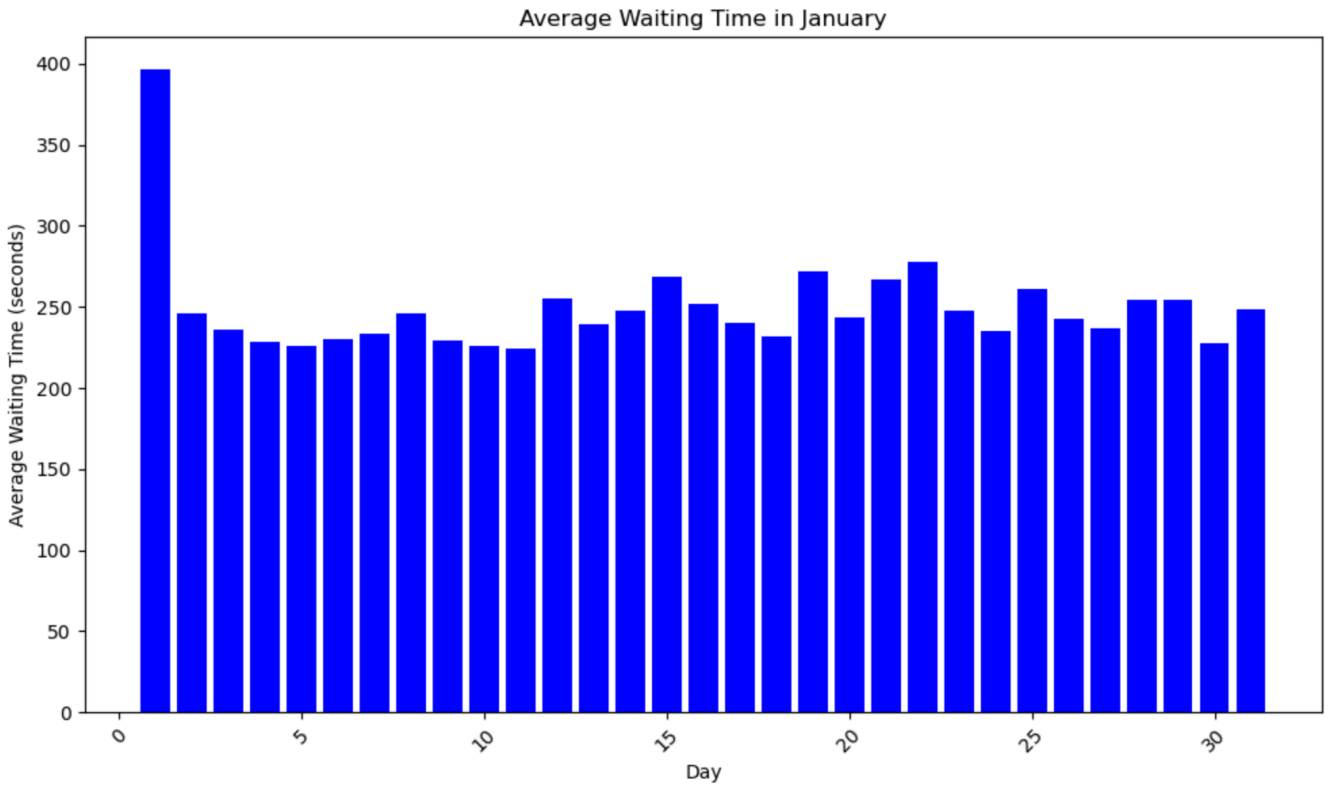
TASK 5 –
Extract the data in January and calculate the average waiting time (use the "request_to_pickup" field) over time. You need to sort the output by day. Draw the histogram with 'days' on the x-axis and ‘average waiting time’ on the y-axis

```
import pandas as pd
import matplotlib.pyplot as plt
avg_waiting_df = pd.read_csv('output_task5/waiting_time_per_day.csv/part-00000-73418422-1703-451b-ad27-2a507cdbe9af-c000.csv')

# Plot the histogram
plt.figure(figsize=(10, 6))
plt.bar(avg_waiting_df['day'], avg_waiting_df['average_waiting_time'], color='blue')

plt.xlabel('Day')
plt.ylabel('Average Waiting Time (seconds)')
plt.title('Average Waiting Time in January')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Average Waiting Time in January



#2}: Identifying Days with Waiting Time Over 300 Seconds

PROCESS AND API's USED

Filtering Based on Condition: The `**filter**` function is used again, this time to select days where the average waiting time exceeds 300 seconds from the aggregated dataset.

VISUALIZATION:

```
2024-03-28 13:47:05,089 INFO codegen.CodeGenerator: Code generated in 15.821134 ms
+---+------------------+
|day|average_waiting_time|
+---+------------------+
|  1|  396.5318744409635|
+---+------------------+


2024-03-28 13:47:05,107 INFO server.AbstractConnector: Stopped Spark@c3c5d20{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
2024-03-28 13:47:05,108 INFO ui.SparkUI: Stopped Spark web UI at http://task5-spark-app-9646978e8549a6d8-driver-svc.d
40
```

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

On days like New Year's Day January 1, it's common to see a significant spike in demand for taxis:

1. **Increased Celebrations**: Many people go out to celebrate, leading to a higher reliance on taxis and ride-sharing services, especially given that people may avoid driving due to alcohol consumption.
2. **Reduced Public Transport**: On holidays, public transportation schedules may be reduced, prompting more people to use taxis.
3. **Limited Driver Availability**: Simultaneously, some drivers may take the day off to celebrate, leading to a reduced supply of available taxis.

TASK 5 CHALLENGES AND SOLUTIONS:

Date Handling: Converting and extracting date parts required understanding of date functions in Spark. Studying Spark's API documentation was helpful.

Conditional Filtering: The challenge was to apply a condition on aggregated results accurately. The solution involved understanding how to use `filter` after aggregation.

TASK 5 KNOWLEDGE AND INSIGHTS GAINED:

- Skills in filtering, aggregating, and acquired the technique to filter aggregated data based on a threshold (waiting times over 300 seconds), useful for spotting outliers or anomalies.

- This analysis provides insight into how waiting times fluctuate daily in January, identifying potential operational inefficiencies or external factors affecting waiting times.

- Identifying days with unusually long waiting times can help in check issues within the taxi service operations or external events impacting service efficiency.

Execution:

Execute the spark application	**Stream the logs of the drive container**
ccc create spark task5.py -s	oc logs -f task5-spark-app-driver

The Spark session is stopped using **stop()** method.

TASK 6: Filtering Data

This task is filtering New York City taxi data for specific trip count criteria, focusing on various times of the day and boroughs, and identifying specific inter-borough trip patterns.

#1} Finding trip counts greater than 0 and less than 1000 for different 'Pickup_Borough' at different 'time_of_day'

PROCESS AND API's USED:

Grouping by Pickup Borough and Time of Day:

Using **groupBy** followed by **agg**, the code calculates the number of trips (**trip_count**) for each **Pickup_Borough** and **time_of_day** combination.

Filtering Trip Counts:

Applying **filter** to the aggregated data to select records where the **trip_count** is greater than 0 and less than 1000, narrowing down to a range that excludes extremes.

VISUALIZATION:

```
2024-03-28 13:32:07,923 INFO codegen.CodeGenerator: Code generated in 9.88417 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       |
+--------------+-----------+----------+

2024-03-28 13:32:08,118 INFO codegen.CodeGenerator: Code generated in 14.120896 ms
```

#2} Calculate the number of trips for each 'Pickup_Borough' in the evening time.

PROCESS AND API's USED:

Evening Trip Counts:

A second **filter** operation separates trip counts specifically for the evening, focusing on the dataset for evening trips across all boroughs.

VISUALIZATION:

```
2024-03-28 13:35:23,007 INFO scheduler.TaskSchedulerImpl: Killing all running tasks in stage 89: Stage finished
2024-03-28 13:35:23,008 INFO scheduler.DAGScheduler: Job 18 finished: showString at NativeMethodAccessorImpl.java:0, took 0.063474 s
+-------------+-----------+----------+
|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       |
+-------------+-----------+----------+

2024-03-28 13:35:23,033 INFO server.AbstractConnector: Stopped Spark@3568868{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
2024-03-28 13:35:23,035 INFO ui.SparkUI: Stopped Spark web UI at http://task6-spark-app-a680a78e8539309e-driver-svc.data-science-ec238
```

#3} The number of trips that started in Brooklyn (Pickup_Borough field) and ended in Staten Island (Dropoff_Borough field)

PROCESS AND API's USED:

Brooklyn to Staten Island Trips:

A last **filter** extracts trips that start in Brooklyn and end in Staten Island. **select** is used to choose relevant columns and **count** to tally the number of such specific trips.

VISUALIZATION:

```
2024-03-28 13:12:37,148 INFO scheduler.DAGScheduler: Job 7 finished: showString at NativeMetho
2024-03-28 13:12:37,182 INFO codegen.CodeGenerator: Code generated in 13.555765 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

2024-03-28 13:12:37,435 INFO codegen.CodeGenerator: Code generated in 9.445144 ms
```

```
2024-03-28 13:26:04,172 INFO scheduler.DAGScheduler: Job 7 finished: showString at NativeMethodAccessorImpl.ja
2024-03-28 13:26:04,206 INFO codegen.CodeGenerator: Code generated in 14.281588 ms
+--------------+---------------+--------------------+
|Pickup_Borough|Dropoff_Borough|         Pickup_Zone|
+--------------+---------------+--------------------+
|      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 ...|
|      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

2024-03-28 13:26:04,472 INFO codegen.CodeGenerator: Code generated in 9.490304 ms
2024-03-28 13:26:04,491 INFO codegen.CodeGenerator: Code generated in 14.891281 ms
2024-03-28 13:26:04,505 INFO codegen.CodeGenerator: Code generated in 9.830926 ms
```

```
2024-03-28 13:16:25,117 INFO scheduler.DAGScheduler: Job 8 finished: count at NativeMethodAccessorImpl.java:0, took 227.54
Count of trips from Brooklyn to Staten Island:  69437
2024-03-28 13:16:25,159 INFO server.AbstractConnector: Stopped Spark@3568868{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
```

TASK 6 CHALLENGES AND SOLUTIONS:

Complex Filtering Conditions: Applying multiple conditions in filters could lead to errors if not carefully managed. Using the **&** operator and ensuring correct syntax in the **filter** function allowed for correct data extraction.

Counting Specific Trip Patterns: Identifying trips between two specific boroughs required a specific query that could potentially return a large dataset. The filter was carefully considered to match exact borough names, and additional **select** was used to narrow down the output.

TASK 6 KNOWLEDGE AND INSIGHTS GAINED:

- Applied filtering to refine the dataset, specifically targeting trips with certain counts and those occurring in the evening.
- Developed a method to extract specific route data, notably from Brooklyn to Staten Island, enhancing the ability to analyse point-to-point trip flows.
- The approach provided insights into typical trip counts, evening trip patterns across boroughs, and detailed travel behaviour between Brooklyn and Staten Island.

Execution:

Execute the spark application	Stream the logs of the drive container
ccc create spark task6.py -s	oc logs -f task6-spark-app-driver

The Spark session is stopped using **stop()** method.

TASK 7: Route Analysis

This task demonstrates a PySpark analysis focusing on comparing the volume of trips between Uber and Lyft across different routes in New York City, as defined by pickup and dropoff service zones.

PROCESSES AND API's USED:

1: Define a New Route Column

withColumn: Adds a new column to DataFrame.

concat_ws: A Spark SQL function that concatenates multiple column values into a single string with a separator which combines 'Pickup_Zone' and 'Dropoff_Zone' into a single route descriptor.

2: Aggregate Uber Trips Data

filter: Filtering the data to only include records where the 'business' column is 'Uber'.

groupBy: Groups the DataFrame by the 'Route' column.

agg: Performs aggregation functions on the grouped data.

count("*"): Counting the number of records (trips) for each group.

alias: Renames the result of the aggregation to 'uber_count' .

3: Aggregate Lyft Trips Data:

The process mirrors Step 2, considering the Lyft business to produce a DataFrame with trip counts per route named as 'lyft_count'.

4: Join Uber and Lyft Route Data:

Join: Merges the Uber and Lyft route data on the common 'Route' column.

withColumn: A new column 'total_count' that sums 'uber_count' and 'lyft_count' to reflect the total trips per route for both services.

select: Chooses specific columns to feature in the resulting DataFrame, which includes the route and the respective counts.

5: Identify Top 10 Most Popular Routes:

orderBy: Arranges the routes in descending order based on 'total_count' to highlight the most frequented routes.

limit: Trims the list to the top 10 routes.

show: Outputs the results, with `truncate=False` ensuring full visibility of the route information.

VISUALIZATION:

```
2024-04-05 10:17:34,511 INFO codegen.CodeGenerator: Code generated in 12.050992 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      |
+-----------------------------------+---------+----------+-----------+

2024-04-05 10:17:34,539 INFO server.AbstractConnector: Stopped Spark@2989a773{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
2024-04-05 10:17:34,542 INFO ui.SparkUI: Stopped Spark web UI at http://new-spark-app-2fcc3c8eadb590ed-driver-svc.data-science-ec23863.svc
```

TASK 7 CHALLENGES AND SOLUTIONS:

- Make sure the concatenated **Route** column was consistent and meaningful required careful planning. The **concat_ws** function helped to address this challenge.

- Aligning the Uber and Lyft datasets on the **Route** column could lead to mismatches if the **Route** values are not perfectly aligned. Ensuring data consistency before the join operation is performed correctly.

TASK 7 KNOWLEDGE AND INSIGHTS:

- Learned to combine multiple columns into a single composite column to represent unique routes, helped to analyse data on a per-route basis.
- Filtering datasets based on business names (Uber and Lyft) and using aggregation to count trips, which is important for understanding service usage.
- Merging datasets to compare trip counts between two services on the same routes, essential for comparative analysis.

Execution:

Execute the spark application	**Stream the logs of the drive container**
ccc create spark task7.py -s	oc logs -f task7-spark-app-driver

The Spark session is stopped using **stop()** method.

TASK 8: Graph Processing

This task is to apply graph theory and analytics for extracting actionable insights from New York City taxi trip data, utilizing PySpark and GraphFrames.

#1}Defining the StructType for vertexSchema and edgeSchema:

PROCESS AND API's USED:

- `**StructType**` and `**StructField**` from PySpark SQL are used to define the schema for vertices and edges of the graph.
- A vertex represents an entity (such as a taxi zone), and an edge represents a relationship (such as a trip between two zones).
- The `**id**` field in the vertex schema and the `**src**` and `**dst**` fields in the edge schema are defined as non-nullable, ensuring data integrity for graph construction.

VISUALIZATION:

```
#1} Define the StructType of vertexSchema and edgeSchema.
    |
    # Schema for vertices dataframe based on the taxi zone lookup data
    vertexSchema = StructType([
        StructField("id", IntegerType(), False),  # False for nullable indicates this field cannot be null
        StructField("Borough", StringType(), True),
        StructField("Zone", StringType(), True),
        StructField("service_zone", StringType(), True)
    ])

    # Schema for edges dataframe based on the rideshare data
    edgeSchema = StructType([
        StructField("src", IntegerType(), False),
        StructField("dst", IntegerType(), False)
    ])
```

#2) Constructing edges and vertices DataFrames:

PROCESS AND API's USED:

- DataFrames `**vertices**` and `**edges**` are created by selecting and renaming the relevant columns from the taxi zone and rideshare using **select()** and **alias().**
- Aliases are used for the naming convention required by **GraphFrames, i.e., `id` for vertices and `src`/`dst` for edges.**

VISUALIZATION:

```
2024-04-03 09:46:52,782 INFO scheduler.TaskSchedulerImpl: Killing all running tasks in stage 6: Stage finished
2024-04-03 09:46:52,784 INFO scheduler.DAGScheduler: Job 6 finished: showString at NativeMethodAccessorImpl.java:0, took 0.942926 s
+---+-------------+--------------------+------------+
| 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

2024-04-03 09:46:52,854 INFO codegen.CodeGenerator: Code generated in 11.224382 ms
2024-04-03 09:46:52,860 INFO spark.SparkContext: Starting job: showString at NativeMethodAccessorImpl.java:0
2024-04-03 09:46:52,861 INFO scheduler.DAGScheduler: Got job 7 (showString at NativeMethodAccessorImpl.java:0) with 1 output partitions
2024-04-03 09:46:53,004 INFO scheduler.DAGScheduler: Job 7 finished: showString at NativeMethodAccess
2024-04-03 09:46:53,025 INFO codegen.CodeGenerator: Code generated in 12.746302 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
```

#3) Creating a graph using the vertices and edges:

PROCESS AND API's USED:

- **GraphFrame-** is constructed using the vertices and edges DataFrames
- **Triplets-** are called on the GraphFrame to view the source vertex (`src`), the edge (`e`), and the destination vertex (`dst`).
- **distinct()-** is applied to ensure unique triplets are displayed, which is required for analyzing the unique connections within the graph.

VISUALIZATION:

Printing the graphframe using show() method:

```
2024-04-03 18:02:42,084 INFO codegen.CodeGenerator: Code generated in 11.290808 ms
+----------------------------------+----------+--------------------------------------+
|src                               |edge      |dst                                   |
+----------------------------------+----------+--------------------------------------+
|[169, Bronx, Mount Hope, Boro Zone]|[169, 125]|[125, Manhattan, Hudson Sq, Yellow Zone]|
|[169, Bronx, Mount Hope, Boro Zone]|[169, 125]|[125, Manhattan, Hudson Sq, Yellow Zone]|
|[169, Bronx, Mount Hope, Boro Zone]|[169, 125]|[125, Manhattan, Hudson Sq, Yellow Zone]|
|[169, Bronx, Mount Hope, Boro Zone]|[169, 125]|[125, Manhattan, Hudson Sq, Yellow Zone]|
|[169, Bronx, Mount Hope, Boro Zone]|[169, 125]|[125, Manhattan, Hudson Sq, Yellow Zone]|
|[169, Bronx, Mount Hope, Boro Zone]|[169, 125]|[125, Manhattan, Hudson Sq, Yellow Zone]|
|[169, Bronx, Mount Hope, Boro Zone]|[169, 125]|[125, Manhattan, Hudson Sq, Yellow Zone]|
|[169, Bronx, Mount Hope, Boro Zone]|[169, 125]|[125, Manhattan, Hudson Sq, Yellow Zone]|
|[169, Bronx, Mount Hope, Boro Zone]|[169, 125]|[125, Manhattan, Hudson Sq, Yellow Zone]|
|[169, Bronx, Mount Hope, Boro Zone]|[169, 125]|[125, Manhattan, Hudson Sq, Yellow Zone]|
+----------------------------------+----------+--------------------------------------+
only showing top 10 rows

2024-04-03 18:02:42,107 INFO server.AbstractConnector: Stopped Spark@1578e326{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
2024-04-03 18:02:42,109 INFO ui.SparkUI: Stopped Spark web UI at http://task8-spark-app-7a70f08ea51c0e08-driver-svc.da
ta-science-ec23863.svc:4040
```

Printing the output distinct() method to show unique triplet combinations:

```
2024-04-05 09:40:41,172 INFO codegen.CodeGenerator: Code generated in 15.548746 ms
+--------------------------------------------------+---------+--------------------------------------------------+
|src                                               |edge     |dst                                               |
+--------------------------------------------------+---------+--------------------------------------------------+
|[66, Brooklyn, DUMBO/Vinegar Hill, Boro Zone]     |[66, 124]|[124, Queens, Howard Beach, Boro Zone]            |
|[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]           |
|[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]                   |
|[256, Brooklyn, Williamsburg (South Side), Boro Zone]|[256, 234]|[234, Manhattan, Union Sq, Yellow Zone]           |
|[34, Brooklyn, Brooklyn Navy Yard, Boro Zone]     |[34, 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

2024-04-05 09:40:41,210 INFO server.AbstractConnector: Stopped Spark@7cb4aaf1{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
2024-04-05 09:40:41,213 INFO ui.SparkUI: Stopped Spark web UI at http://task8-spark-app-2b581c8ead94c404-driver-svc.data-science-ec23863.svc:40
40
```

#4) Counting connected vertices within the same Borough and service_zone:

PROCESS:

A motif finding pattern **`(a)-[e]->(b)`** is used to find relationships where both vertices have the same `Borough` and `service_zone`. The result is a DataFrame of vertices connected by edges within the same borough and service zone, which may indicate a strong relationship in the taxi network.

API's USED:

find
distinct
select
count
show

VISUALIZATION:

Printing the graphframe using show() method:

```
272.523033 s
2024-04-04 13:07:07,440 INFO codegen.CodeGenerator: Code generated in 15.837765 ms
+---+---+---------+------------+
|id |id |Borough  |service_zone|
+---+---+---------+------------+
|125|249|Manhattan|Yellow Zone |
|125|249|Manhattan|Yellow Zone |
|125|249|Manhattan|Yellow Zone |
|125|249|Manhattan|Yellow Zone |
|125|249|Manhattan|Yellow Zone |
|125|249|Manhattan|Yellow Zone |
|125|249|Manhattan|Yellow Zone |
|125|249|Manhattan|Yellow Zone |
|125|249|Manhattan|Yellow Zone |
|125|249|Manhattan|Yellow Zone |
+---+---+---------+------------+
only showing top 10 rows

2024-04-04 13:07:07,465 INFO server.AbstractConnector: Stopped Spark@76a8489f{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
2024-04-04 13:07:07,467 INFO ui.SparkUI: Stopped Spark web UI at http://task8-spark-app-2acac18ea9337ba7-driver-svc.data-
science-ec23863.svc:4040
2024-04-04 13:07:07,471 INFO k8s.KubernetesClusterSchedulerBackend: Shutting down all executors
```

Printing the output distinct() method to show unique combinations:

```
2024-04-05 09:52:44,041 INFO codegen.CodeGenerator: Code generated in 13.018027 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 09:52:44,213 INFO codegen.CodeGenerator: Code generated in 9.495648 ms
2024-04-05 09:52:44,232 INFO codegen.CodeGenerator: Code generated in 13.520469 ms
2024-04-05 09:52:44,301 INFO spark.SparkContext: Starting job: count at NativeMethodAccessorImpl.java:0
2024-04-05 09:52:44,302 INFO scheduler.DAGScheduler: Registering RDD 40 (count at NativeMethodAccessorImpl.java:0) as input to shuffle 5
```

Total connected Vertices Output:

```
Total connected vertices with the same Borough and service zone: 46886992
```

#5) Performing page ranking on the graph DataFrame:

PROCESS:

The PageRank algorithm is used to measure the importance of each vertex in the graph.

- `**resetProbability**` is set to account for the probability of random jumps.
- `**tol**` parameter determines the convergence tolerance of the PageRank algorithm; lower values lead to more accurate but potentially longer computations.

API's USED:

pageRank
sort
select
show

VISUALIZATION:

```
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|3.568322341656471 3|
|61 |2.6763973653412996|
+---+-----------------+
only showing top 5 rows
```