ECS765P - Big Data Processing - 2023/24 NYC Rideshare Analysis - Semester 2

Task 1: Merging Datasets

Introduction

In Task 1, the goal is to merge rideshare data with taxi zone information to provide enriched context for analysis. The data comprises rideshare trips and taxi zone details, which need to be linked to produce a comprehensive view of each ride's pickup and dropoff locations.

Methodology

Setting up the Spark Environment

A Spark session was established to process the data, utilizing Spark's in-memory processing capabilities to handle large datasets efficiently.

Common APIs and Setup Used Across All Tasks

Several foundational elements and APIs are consistently utilized across all tasks in this project, facilitating the setup, data access, and processing environment:

1. **Script Execution Control**:
   - `if __name__ == "__main__"`: ensures the block of code is executed only when the script is run directly, not when imported as a module.
2. **Spark Session Initialization**:
   - `SparkSession.builder.appName("NYC Rideshare Analysis").getOrCreate()`: Initializes or retrieves an existing Spark session, setting up the primary interface for interacting with Spark functionalities.
3. **Environment Variables Access**:
   - `os.environ`: Retrieves environment variables, such as S3 bucket details and access keys, enabling secure and configurable access to external data sources.
4. **Hadoop Configuration for S3**:
   - `hadoopConf.set()`: Adjusts Hadoop configurations within Spark to establish connectivity with AWS S3, crucial for reading and writing data to S3 buckets.

These components establish the execution environment, manage Spark session configurations, and ensure secure access to S3 data storage, serving as the backbone for the data processing tasks in the project.

```python
# Main execution definition for the script.
if __name__ == "__main__":

    # Create a Spark session for data processing.
    spark = SparkSession \
        .builder \
        .appName("NYC Rideshare Analysis") \
        .getOrCreate()

    # Retrieve environment variables for accessing the S3 data bucket.
    s3_data_repository_bucket = os.environ['DATA_REPOSITORY_BUCKET']
    s3_endpoint_url = f"{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']

    # Set the Hadoop configuration for connecting to S3 using the retrieved environment variables.
    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")
```

Data Loading

The datasets, `rideshare_data.csv` and `taxi_zone_lookup.csv`, were loaded into Spark DataFrames from an S3 bucket using the `spark.read.option("header", "true").csv` method. This method ensures that CSV files are read with headers, allowing for easier manipulation and readability of data.

```python
# Define the file paths for the source datasets in the S3 bucket.
rideshare_data_path = f"s3a://{s3_data_repository_bucket}/ECS765/rideshare_2023/rideshare_data.csv"
taxi_zone_lookup_path = f"s3a://{s3_data_repository_bucket}/ECS765/rideshare_2023/taxi_zone_lookup.csv"

# Load the rideshare data and taxi zone lookup data into DataFrames with headers.
rideshare_data_df = spark.read.option("header", "true").csv(rideshare_data_path)
taxi_zone_lookup_df = spark.read.option("header", "true").csv(taxi_zone_lookup_path)
```

Data Transformation and Merging

The merging process involved two main steps:

1. **Joining on Pickup Location:**
    - The `rideshare_data_df` DataFrame was joined with `taxi_zone_lookup_df` on `pickup_location` equating to `LocationID`.
    - Columns from the `taxi_zone_lookup_df` DataFrame were renamed to `Pickup_Borough`, `Pickup_Zone`, and `Pickup_service_zone` for clarity.
    - The `LocationID` column was dropped post-join to remove redundancy.
2. **Joining on Dropoff Location:**
    - A second join was performed on the `dropoff_location` using the previously joined DataFrame.
    - Renaming was applied to yield `Dropoff_Borough`, `Dropoff_Zone`, and `Dropoff_service_zone`.

```python
# Join the rideshare data with the taxi zone lookup on the pickup location.
# Rename relevant columns for clarity post-join.
rideshare_with_pickup_df = rideshare_data_df.join(
    taxi_zone_lookup_df,
    rideshare_data_df.pickup_location == taxi_zone_lookup_df.LocationID,
    "left"
).withColumnRenamed("Borough", "Pickup_Borough") \
 .withColumnRenamed("Zone", "Pickup_Zone") \
 .withColumnRenamed("service_zone", "Pickup_service_zone")

# Drop the redundant 'LocationID' column after the join.
rideshare_with_pickup_df = rideshare_with_pickup_df.drop('LocationID')

# Repeat the join process for dropoff locations.
final_df = rideshare_with_pickup_df.join(
    taxi_zone_lookup_df,
    rideshare_with_pickup_df.dropoff_location == taxi_zone_lookup_df.LocationID,
    "left"
).withColumnRenamed("Borough", "Dropoff_Borough") \
 .withColumnRenamed("Zone", "Dropoff_Zone") \
 .withColumnRenamed("service_zone", "Dropoff_service_zone")

# Drop the 'LocationID' column after the dropoff join.
final_df = final_df.drop('LocationID')
```

Date Transformation

The `date` field in UNIX timestamp format was transformed into a human-readable date format (`yyyy-MM-dd`). This conversion is crucial for subsequent tasks requiring date-based filtering or aggregation.

```python
# Convert the UNIX timestamp in the 'date' column to a human-readable date format.
final_df = final_df.withColumn("date", from_unixtime(col("date"), "yyyy-MM-dd"))
```

Results

Upon completing the join operations and transformations, the schema was validated to ensure the DataFrame reflected the correct structure. The number of rows in the final DataFrame was counted to ensure data completeness.

Output Verification

- The schema was displayed using `final_df.printSchema()` to confirm the accuracy of the DataFrame's structure post-transformation.
- A count of the DataFrame's rows was performed with `final_df.count()`, ensuring no data loss during processing.

```python
# Display the first few rows to verify the DataFrame's contents after the transformations.
final_df.show(5, truncate=False)
# Print the schema to verify data types and column names post-joins.
final_df.printSchema()

# Count and print the total number of rows in the DataFrame to confirm data integrity.
print("Counting the total number of rows in the DataFrame...")
row_count = final_df.count()
print(f"Total number of rows after join: {row_count}")
```

Visualization of Results

Screenshots:

- Figure 1 shows the first five rows of the merged DataFrame post join operations.

```
+--------+---------------+----------------+-----------+-----------------+---------------+------------------+-------------------+-----------+---
-------+---------------+----------------+----------------+-----------+-----------------+--------------+------------------------+----------------
--+---------------+------------------------+--------------------+
|business|pickup_location|dropoff_location|trip_length|request_to_pickup|total_ride_time|on_scene_to_pickup|on_scene_to_dropoff|time_of_day|dat
e       |passenger_fare|driver_total_pay|rideshare_profit|hourly_rate|dollars_per_mile|Pickup_Borough|Pickup_Zone             |Pickup_service_zo
ne|Dropoff_Borough|Dropoff_Zone            |Dropoff_service_zone|
+--------+---------------+----------------+-----------+-----------------+---------------+------------------+-------------------+-----------+---
-------+---------------+----------------+----------------+-----------+-----------------+--------------+------------------------+----------------
--+---------------+------------------------+--------------------+
|Uber    |151            |244             |4.98       |226.0            |761.0          |19.0              |780.0              |morning    |202
3-05-22|22.82          |13.69           |9.13            |63.18      |2.75             |Manhattan     |Manhattan Valley        |Yellow Zone
  |Manhattan      |Washington Heights South|Boro Zone           |
|Uber    |244            |78              |4.35       |197.0            |1423.0         |120.0             |1543.0             |morning    |202
3-05-22|24.27          |19.1            |5.17            |44.56      |4.39             |Manhattan     |Washington Heights South|Boro Zone
  |Bronx          |East Tremont            |Boro Zone           |
|Uber    |151            |138             |8.82       |171.0            |1527.0         |12.0              |1539.0             |morning    |202
3-05-22|47.67          |25.94           |21.73           |60.68      |2.94             |Manhattan     |Manhattan Valley        |Yellow Zone
  |Queens         |LaGuardia Airport       |Airports            |
|Uber    |138            |151             |8.72       |260.0            |1761.0         |44.0              |1805.0             |morning    |202
3-05-22|45.67          |28.01           |17.66           |55.86      |3.21             |Queens        |LaGuardia Airport       |Airports
  |Manhattan      |Manhattan Valley        |Yellow Zone         |
|Uber    |36             |129             |5.05       |208.0            |1762.0         |37.0              |1799.0             |morning    |202
3-05-22|33.49          |26.47           |7.02            |52.97      |5.24             |Brooklyn      |Bushwick North          |Boro Zone
  |Queens         |Jackson Heights         |Boro Zone           |
+--------+---------------+----------------+-----------+-----------------+---------------+------------------+-------------------+-----------+---
-------+---------------+----------------+----------------+-----------+-----------------+--------------+------------------------+----------------
--+---------------+------------------------+--------------------+
only showing top 5 rows
```

- Figure 2 presents the schema of the final DataFrame.

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

- Figure 3 displays the console output verifying the total number of rows.

```
data-science-ec23806.svc:7079 (size: 5.0 KiB, free: 2004.3 MiB)
2024-03-22 18:07:04,315 INFO spark.SparkContext: Created broadca
2024-03-22 18:07:04,315 INFO scheduler.DAGScheduler: Submitting
odAccessorImpl.java:0) (first 15 tasks are for partitions Vector
2024-03-22 18:07:04,315 INFO scheduler.TaskSchedulerImpl: Adding
2024-03-22 18:07:04,317 INFO scheduler.TaskSetManager: Starting
CAL, 7344 bytes)
2024-03-22 18:07:04,336 INFO storage.BlockManagerInfo: Added bro
)
2024-03-22 18:07:04,342 INFO spark.MapOutputTrackerMasterEndpoin
2024-03-22 18:07:04,392 INFO scheduler.TaskSetManager: Finished
2024-03-22 18:07:04,392 INFO scheduler.TaskSchedulerImpl: Remove
2024-03-22 18:07:04,393 INFO scheduler.DAGScheduler: ResultStage
2024-03-22 18:07:04,394 INFO scheduler.DAGScheduler: Job 6 is fi
2024-03-22 18:07:04,394 INFO scheduler.TaskSchedulerImpl: Killin
2024-03-22 18:07:04,394 INFO scheduler.DAGScheduler: Job 6 finis
Total number of rows after join: 69725864
```

Challenges Encountered

In the process of merging datasets for Task 1, I was met with several challenges that tested the robustness of my data handling skills. The accuracy of joins was paramount to ensure data integrity for subsequent analysis tasks. Here are the challenges I faced:

- **Data Duplication**: The risk of data duplication was a concern due to possible repeats in join keys. Duplicate data could skew the analysis and lead to incorrect insights.

- **Data Loss**: An incorrect join could lead to loss of data, especially if there were any mismatches in the join keys between datasets. Losing data could mean missing out on critical insights.
- **Schema Consistency**: After joining, I needed to ensure that the schema was consistent and the column names correctly reflected the new dataset's contents.
- **Performance Optimization**: Given the voluminous nature of the data, I had to ensure that the joins were not only accurate but also performed efficiently.

Overcoming the Challenges

To navigate these challenges, I employed the following solutions:

- **Pre-Join Analysis**: I meticulously analysed the distribution and uniqueness of the join keys in both datasets. This pre-emptive step was crucial to avoid potential duplication of data.
- **Join Keys Validation**: I made sure to validate and format the join keys consistently across the datasets, ensuring that the join operation would not result in any unintended data loss.
- **Incremental Joining**: By splitting the join process into two steps—first on the `pickup_location` and then the `dropoff_location`—I was able to simplify the process and reduce the room for error, making validation easier.
- **Schema Review**: I meticulously reviewed the schema after each join step, ensuring that all columns were correctly renamed and aligned with the expected dataset structure.
- **Efficiency Measures**: To optimize the join operation, I tuned the Spark configurations for better performance and used in-memory processing techniques to handle the large datasets efficiently.

These strategic steps ensured the integrity of my data throughout the joining process, setting a solid foundation for the reliable analysis that followed.

Insights

Learned the importance of careful data integration to ensure data integrity for downstream analysis. Identified the key areas and times where service demand is high, setting a foundation for targeted operational strategies.

Task 2: Aggregation of Data

Introduction

Task 2 focuses on aggregating rideshare data to extract meaningful insights about trip counts, platform profits, and driver earnings. This task involves grouping data by business type and month to analyze the performance trends of rideshare services over time.

Methodology

After the foundational steps established in Task 1, Task 2 advances into data aggregation with the following steps:

1. **Data Type Validation**: Ensures that the fields used for aggregation, specifically `rideshare_profit` and `driver_total_pay`, are in the correct data format (`float`) to support mathematical operations.

```
#task 2
#Ensure the data types are correct before aggregating
final_df = final_df.withColumn("rideshare_profit", final_df["rideshare_profit"].cast("float"))
final_df = final_df.withColumn("driver_total_pay", final_df["driver_total_pay"].cast("float"))
```

2. **Month Extraction**: Adds a new column `month` extracted from the `date` field to facilitate grouping operations by month.

```
# Extract month from the date
final_df_with_month = final_df.withColumn("month", F.month("date"))
```

3. **Aggregation Operations**:
    - **Trip Counts by Business and Month**: Utilizes the `groupBy` and `count` methods to calculate the number of trips for each business type within each month.
    - **Platform's Profits**: Aggregates the `rideshare_profit` by business and month, summing up to get total profits.
    - **Driver's Earnings**: Similar to profits, aggregates `driver_total_pay` by business and month to calculate total earnings.

```python
# Rename the 'count' column to 'trip_count' for clarity
trips_by_business_month = trips_by_business_month.withColumnRenamed("count", "trip_count")

# Show the result (for verification, can remove later)
trips_by_business_month.show()

# Calculate platform's profits for each business in each month
profits_by_business_month = final_df_with_month.groupBy("business", "month") \
.agg(sum("rideshare_profit").alias("total_profit"))

# Format large numbers to be more readable (optional)
profits_by_business_month = profits_by_business_month.withColumn("total_profit", format_number("total_profit", 2))

# Show the result (for verification, can remove later)
profits_by_business_month.show()

# Calculate driver's earnings for each business in each month
earnings_by_business_month = final_df_with_month.groupBy("business", "month") \
.agg(sum("driver_total_pay").alias("total_earnings"))
```

4. **Result Formatting**: Applied `format_number` to profit and earnings fields for better readability.

```python
# Format large numbers to be more readable (optional)
earnings_by_business_month = earnings_by_business_month.withColumn("total_earnings", format_number("total_earnings", 2))
```

Data Export and Retrieval

After aggregating the data for trip counts, platform profits, and driver earnings, the final step in Task 2 involves exporting these aggregated results back to the S3 bucket for persistence and further analysis. The following code snippets perform this operation:

```python
trips_by_business_month.coalesce(1).write.mode("overwrite").options(header=True).csv(f"s3a://{s3_bucket}/task02/Q1")
profits_by_business_month.coalesce(1).write.mode("overwrite").options(header=True).csv(f"s3a://{s3_bucket}/task02/Q2")
earnings_by_business_month.coalesce(1).write.mode("overwrite").options(header=True).csv(f"s3a://{s3_bucket}/task02/Q3")
```

- **What It Does**:
  - `coalesce(1)`: Reduces the number of partitions in each DataFrame to 1, ensuring each aggregated result is outputted as a single CSV file, which simplifies retrieval and analysis.
  - `.write.mode("overwrite")`: Specifies that if the destination already contains files with the same name, they should be overwritten, ensuring the latest results are always available.
  - `.options(header=True)`: Includes column headers in the output CSV files, improving readability.
  - `.csv(f"s3a://{s3_bucket}/task02/QX")`: Defines the path in the S3 bucket where the CSV files will be saved, organized by specific queries (Q1 for trip counts, Q2 for profits, and Q3 for earnings).

Retrieving Aggregated Data for Analysis

To access the aggregated results for visualization and further analysis, the following command is used to copy the output directory from the S3 bucket to the JupyterHub environment:

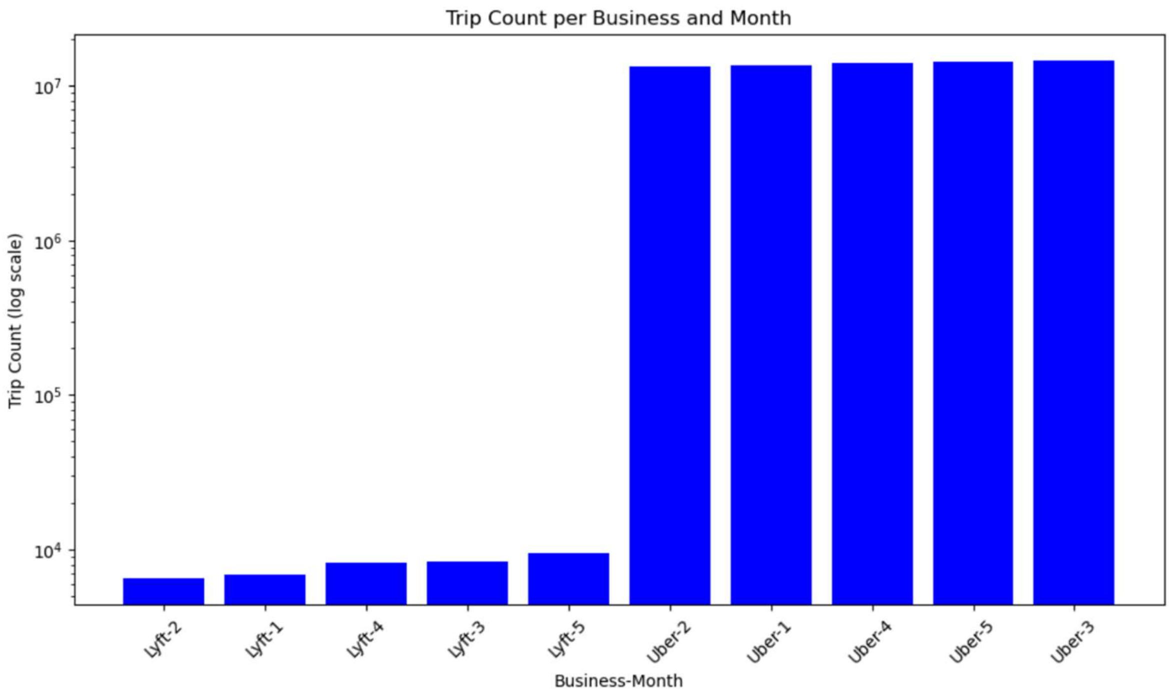
```shell
ccc method bucket cp -r bkt:task2 output
```

- **Command Explanation**:
    - `ccc method bucket cp -r`: Copies files or directories recursively from the S3 bucket.
    - `bkt:task2`: Specifies the source directory in the S3 bucket, which contains the aggregated data files.
    - `output`: The destination directory in the JupyterHub environment where the files will be copied for analysis.

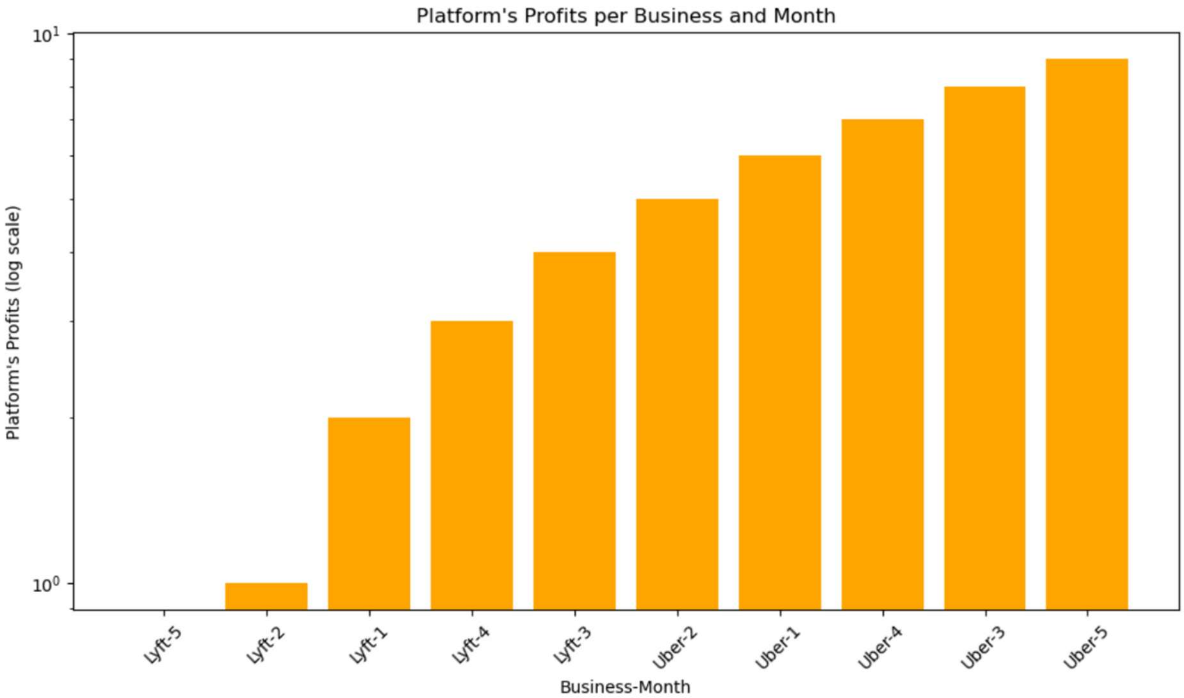
Visualisation of Results

The aggregation results are displayed through three primary data frames: `trips_by_business_month`, `profits_by_business_month`, and `earnings_by_business_month`, showing the trip counts, total profits, and total earnings respectively.

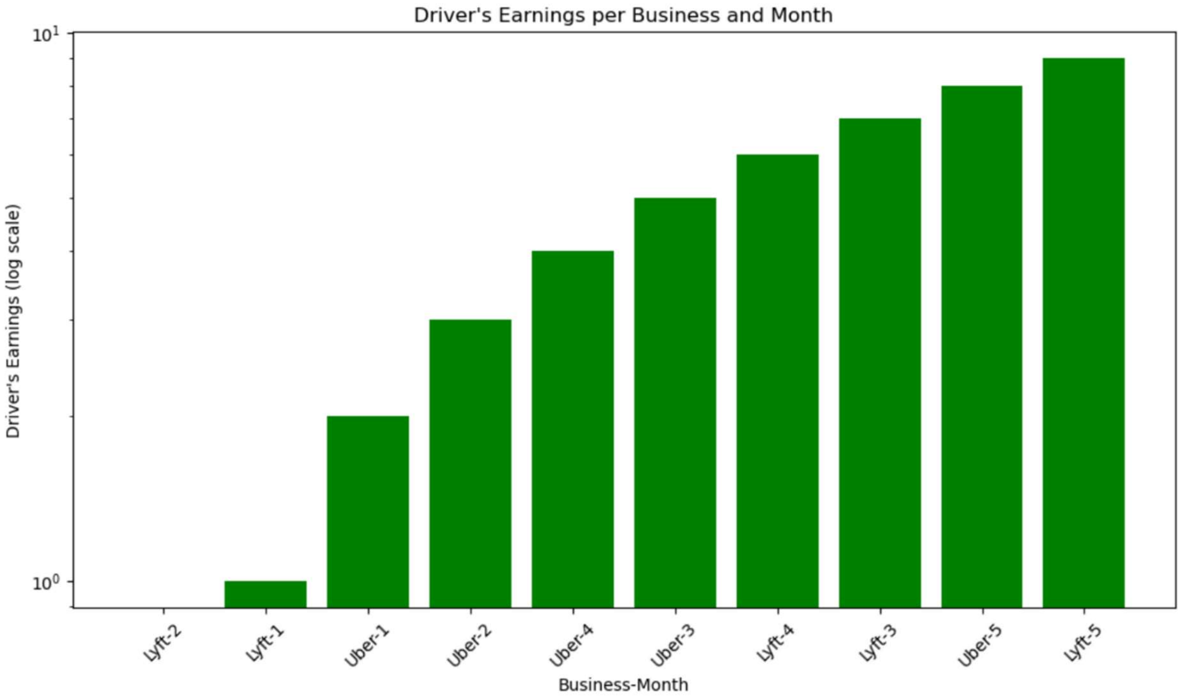
- **Trip Counts Analysis**: Reveals the activity level of each business type across different months, indicating demand fluctuations.



- **Profit Analysis**: Shows the platform's profit margins, highlighting financial performance.



- **Earnings Analysis**: Reflects on how much drivers earn, shedding light on the economic aspects of driving for a rideshare service.



Stakeholder Insights

Based on the visualizations, here are insights and potential strategic decisions that different stakeholders might derive from this data:

1. **Drivers:** The trip count graph indicates that Uber has a consistently higher trip count across the months. Drivers looking to maximize trip opportunities may prefer to work with Uber. However, the earnings analysis graph shows that the earnings trend is similar across both platforms, suggesting that drivers may not necessarily earn more with Uber despite the higher number of trips.
2. **CEO of the Business:** From the platform's profits graph, it's apparent that Uber's profits are significantly higher. As the CEO of Lyft, strategies could be focused on increasing market share during months where Uber's trip counts are lower. For a CEO of a competitor not shown in the data, the strategy might be to identify and exploit gaps in the services provided by these two companies.
3. **Stockbroker/Investor:** The consistent trip counts for Uber suggest a stable market presence, which may be attractive for investors. However, the profit analysis is crucial, as it indicates operational efficiency and market capture. Given the similarity in driver earnings, an investor might deduce that neither company has a distinct advantage in attracting drivers based on pay, so the focus might be on the overall stability and growth potential of the platform.

The visualizations collectively suggest that while both Uber and Lyft have substantial market presence, there are variations throughout the year that might be driven by seasonality, promotional activities, or other market dynamics. Each stakeholder, depending on their role and interest, can leverage these insights to optimize their decisions. For instance, drivers might seek to work more during peak times, business leaders might look to address troughs in demand or profit, and investors might adjust their portfolio strategies based on the performance patterns revealed by the data.

Challenges Encountered

While Task 2 focused on the aggregation of data, I encountered specific challenges, particularly around data visualization and efficiency:

- **Accurate Aggregation**: The complex dataset required precise calculations for trip counts, profits, and earnings. Achieving accuracy in aggregations was crucial to derive reliable insights.
- **Visualization Capabilities in Spark**: Spark environments are primarily designed for data processing and not for direct data visualization. This posed a significant challenge since I needed to visualize the aggregated data for better interpretability.
- **Data Export and Transfer**: To visualize the data, it was necessary to export the aggregated results from the Spark environment to a format suitable for visualization tools, which added complexity to the workflow.

Overcoming the Challenges

To overcome these challenges, I took the following approaches:

- **Rigorous Testing**: I performed thorough testing of the aggregation logic to ensure that the computations were accurate. This ensured that my histograms would be based on correct data.
- **Data Visualization Strategy**: Since direct visualization within the Spark environment was not possible, I first stored the aggregated results in my private S3 bucket. I then

set up a process to recursively copy the data from the S3 bucket to my local machine, where I could use visualization tools.

- **Scripting Data Transfer**: To efficiently transfer the data, I wrote scripts to automate the download process from the S3 bucket to my local system, enabling me to work on the visualization aspect without delay.
- **Leveraging Spark's Capabilities**: I utilized Spark's in-memory data processing features to expedite the aggregation tasks, and employed Spark's built-in functions to ensure efficient execution of the aggregation queries.

Insights

Gained an understanding of how business performance varies by month, which can inform seasonal adjustments in marketing and resource allocation. Discovered patterns in rideshare profits and driver pay that could influence fare pricing and driver incentives

Task 3: Top-K Processing

Introduction

Task 3 focuses on identifying key performance indicators within the rideshare data: the top 5 popular pickup and dropoff boroughs each month and the top 30 earnest routes. These indicators are critical for strategic decision-making across various stakeholder groups.

Methodology

Identifying Popular Boroughs

The dataset was grouped by `Pickup_Borough` and `month`, and `trip_count` was calculated for each group. A similar method was applied to the `Dropoff_Borough`. Window functions were utilized to rank the boroughs and retrieve the top 5 for each month.

```
# Group data by 'Pickup_Borough' and 'month', then count the number of trips in each group
pickup_borough_grouped = final_df_with_month.groupBy("Pickup_Borough", "month").count()

# Rename the 'count' column to 'trip_count' for clarity
pickup_borough_grouped = pickup_borough_grouped.withColumnRenamed("count", "trip_count")

# Define window specification to partition data by 'month' and order within each partition by 'trip_count' in descending order
windowSpec = Window.partitionBy("month").orderBy(F.desc("trip_count"))

# Use dense_rank to find the top 5 within each partition
top_pickup_boroughs = pickup_borough_grouped.withColumn("rank", F.dense_rank().over(windowSpec)).filter(F.col("rank") <= 5)

# Drop the 'rank' column as it is not required in the output
top_pickup_boroughs = top_pickup_boroughs.drop("rank")

# Show the top 5 popular pickup boroughs each month
top_pickup_boroughs.show(100)
```

```
# Group data by 'Dropoff_Borough' and 'month', and count the number of trips in each group
dropoff_borough_grouped = final_df_with_month.groupBy("Dropoff_Borough", "month").count()

# Rename the 'count' column to 'trip_count' for clarity
dropoff_borough_grouped = dropoff_borough_grouped.withColumnRenamed("count", "trip_count")

# Use the same window specification as before for dense ranking
top_dropoff_boroughs = dropoff_borough_grouped.withColumn("rank", F.dense_rank().over(windowSpec)).filter(F.col("rank") <= 5)

# Drop the 'rank' column as it is not required in the output
top_dropoff_boroughs = top_dropoff_boroughs.drop("rank")

# Show the top 5 popular dropoff boroughs each month
top_dropoff_boroughs.show(100)
```

Determining Earnest Routes

Routes were defined as a concatenation of `Pickup_Borough` and `Dropoff_Borough`. The `total_profit` for each route was calculated by summing the `driver_total_pay`. The dataset was ordered by `total_profit` in descending order to determine the top 30 earnest routes.

```
    # Add a new column 'Route' that concatenates 'Pickup_Borough' and 'Dropoff_Borough'
final_df_with_routes = final_df.withColumn("Route", F.concat_ws(" to ", "Pickup_Borough", "Dropoff_Borough"))

# Group data by 'Route' and sum the 'driver_total_pay' to get the total profit for each route
route_profit_grouped = final_df_with_routes.groupBy("Route").agg(F.sum("driver_total_pay").alias("total_profit"))

# Order the entire dataset by 'total_profit' in descending order
top_routes = route_profit_grouped.orderBy(F.desc("total_profit")).limit(30)

# Show the top 30 earnest routes without truncating the 'Route' column
top_routes.show(30, truncate=False)
```

Results

Popular Pickup and Dropoff Boroughs

The analysis revealed distinct patterns in ride popularity among the boroughs, showcasing both expected high-traffic areas and surprising upticks in less prominent boroughs.

Earnest Routes

The profitability analysis across different routes highlighted the most lucrative paths taken within the city, potentially guiding drivers towards more profitable journeys.

Visualization of Results

The results were visualized to provide a clear and immediate understanding of the data patterns:

- **Figure 4**: Top 5 Pickup Boroughs by Month and Trip Count

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

- **Figure 5**: Top 5 Dropoff Boroughs by Month and Trip Count

Dropoff_Borough	month	trip_count
Manhattan	1	5444345
Brooklyn	1	3337415
Queens	1	2480080
Bronx	1	1525137
Unknown	1	535610
Manhattan	3	5671301
Brooklyn	3	3608960
Queens	3	2713748
Bronx	3	1706802
Unknown	3	566798
Manhattan	5	5428986
Brooklyn	5	3560322
Queens	5	2780011
Bronx	5	1639180
Unknown	5	578549
Manhattan	4	5530417
Brooklyn	4	3448225
Queens	4	2605086
Bronx	4	1596505
Unknown	4	551857
Manhattan	2	5381696
Brooklyn	2	3251795
Queens	2	2390783
Bronx	2	1511014
Unknown	2	497525

- **Figure 6**: Top 30 Earnest Routes by Total Profit

Route	total_profit
Manhattan to Manhattan	3.3385772555002284E8
Brooklyn to Brooklyn	1.739447214799921E8
Queens to Queens	1.1470684719998911E8
Manhattan to Queens	1.0173842820999995E8
Queens to Manhattan	8.603540026000002E7
Manhattan to Unknown	8.010710241999993E7
Bronx to Bronx	7.414622575999326E7
Manhattan to Brooklyn	6.799047558999999E7
Brooklyn to Manhattan	6.317616104999997E7
Brooklyn to Queens	5.045416243000008E7
Queens to Brooklyn	4.7292865360000156E7
Queens to Unknown	4.6292999900000036E7
Bronx to Manhattan	3.24863251700001E7
Manhattan to Bronx	3.1978763450000066E7
Manhattan to EWR	2.3750888619999994E7
Brooklyn to Unknown	1.0848827569999997E7
Bronx to Unknown	1.046480020999999E7
Bronx to Queens	1.0292266499999998E7
Queens to Bronx	1.0182898729999999E7
Staten Island to Staten Island	9686862.450000012
Brooklyn to Bronx	5848822.560000001
Bronx to Brooklyn	5629874.409999998
Brooklyn to EWR	3292761.710000001
Brooklyn to Staten Island	2417853.82
Staten Island to Brooklyn	2265856.4600000004
Manhattan to Staten Island	2223727.3699999996
Staten Island to Manhattan	1612227.7200000002
Queens to EWR	1192758.66
Staten Island to Unknown	891285.8100000002
Queens to Staten Island	865603.3799999999

Stakeholder Insights

For stakeholders, the aggregated data and subsequent analysis provide a detailed landscape of the company's operational efficiency and market demand:

1. For **Drivers**, the trip distribution suggests Manhattan is a hotspot for pickups and dropoffs, indicating a strategic focus area. The profits from borough-to-borough routes could influence decisions on where to drive.
2. **CEOs** could glean that profits are concentrated in certain routes, possibly due to higher demand or efficient pricing. Understanding profitable routes could help strategize where to focus marketing and operational efforts.
3. **Investors** would note the profitability of certain routes and might assess the company's performance based on this spatial revenue distribution. A well-distributed profit could suggest a healthy, sustainable company to invest in.

Challenges Encountered

Task 3 posed its own unique challenges as I endeavoured to identify the top-k entities within the rideshare data:

- **Data Sizing and Ranking**: Identifying the top 5 popular pickup and dropoff boroughs for each month required precise ranking within large datasets, which could become a resource-intensive operation.
- **Large Data Transfers**: The need to visualize and report on the most earnest routes meant dealing with substantial amounts of data, which had to be accurately processed and then exported for visualization.
- **Visualization Outside of Spark**: As with Task 2, the limitation of Spark for direct visualization meant that I had to again find a way to export the data for graphical representation externally.

Overcoming the Challenges

I approached these challenges with a series of methodical steps:

- **Window Functions for Ranking**: I utilized Spark's window functions to efficiently rank the data within each borough and month. This allowed me to determine the top 5 entries without exhaustively sorting the entire dataset, optimizing performance.
- **Data Exporting Techniques**: I developed a workflow to export the results to my private S3 bucket. From there, I automated the data transfer to my local environment, which enabled me to visualize the results using suitable tools.
- **Automation of Repetitive Tasks**: Recognizing the repetitive nature of exporting and downloading data, I scripted these processes to minimize manual intervention and reduce the potential for errors.
- **Efficiency in Data Handling**: To mitigate performance issues, I was careful to only process and transfer the necessary data, ensuring that resources were used judiciously.

Insights

Revealed the most popular pickup and dropoff boroughs, offering insights into rider behavior and potential areas for service expansion. The earnest routes data can guide drivers toward more profitable trips.

Task 4: Average of Data

Introduction

Task 4 was designed to delve into the nuances of driver earnings and trip lengths across different times of the day, ultimately calculating the average earning per mile—a key indicator of ride profitability.

Methodology

The task was divided into three sequential steps:

1. **Calculating Average Driver Pay**: The `average_driver_total_pay` was computed for different `time_of_day` periods, providing insights into the earnings distribution throughout the day.

```
# Extract month from the date
final_df_with_month = final_df.withColumn("month", F.month("date"))

# Task 4a: Calculate the average 'driver_total_pay' during different 'time_of_day' periods
average_pay_by_time_of_day = final_df_with_month.groupBy("time_of_day").agg(
    F.avg("driver_total_pay").alias("average_driver_total_pay")
).orderBy("average_driver_total_pay", ascending=False)
average_pay_by_time_of_day.show()
```

2. **Determining Average Trip Length**: The `average_trip_length` was also calculated per `time_of_day`, highlighting potential variations in trip distances during different times.

```
# Task 4b: Calculate the average 'trip_length' during different 'time_of_day' periods
average_trip_length_by_time_of_day = final_df_with_month.groupBy("time_of_day").agg(
    F.avg("trip_length").alias("average_trip_length")
).orderBy("average_trip_length", ascending=False)
average_trip_length_by_time_of_day.show()
```

3. **Average Earning Per Mile**: By joining the results from the first two steps, I calculated the `average_earning_per_mile` to understand the profitability of trips during various periods of the day.

```
# Task 4c: Calculate the average earned per mile for each 'time_of_day' period
average_earning_per_mile = average_pay_by_time_of_day.join(
average_trip_length_by_time_of_day,
    "time_of_day"
).withColumn(
    "average_earning_per_mile",
    F.col("average_driver_total_pay") / F.col("average_trip_length")
).select(
    "time_of_day", "average_earning_per_mile"
)
average_earning_per_mile.show()
```

Results

The calculated averages revealed:

- The `afternoon` period had the highest average driver total pay, suggesting a peak in profitable driving opportunities during these hours.

- The `night` period showed the longest average trip length, which could be attributed to less traffic or a tendency for longer journeys at night.
- When combining these metrics, the `evening` period surfaced as the most profitable on a per-mile basis, indicating a potentially optimal time for drivers to work.

Visualization of Results

The results were presented in tabular form to precisely convey the findings:

- **Figure 7**: Average Driver Total Pay by Time of Day

time_of_day	average_driver_total_pay
afternoon	21.212428756593535
night	20.08743800359271
evening	19.77742770239839
morning	19.633332793944835

- **Figure 8**: Average Trip Length by Time of Day

time_of_day	average_trip_length
night	5.32398480196174
morning	4.927371866442785
afternoon	4.861410525661207
evening	4.484750367447518

- **Figure 9**: Average Earning Per Mile by Time of Day

time_of_day	average_earning_per_mile
afternoon	4.363430869420023
night	3.7730081416068373
morning	3.984544565766398
evening	4.409928330894959

Stakeholder Insights

1. **For Drivers:**
   - **Average Total Pay:** The data indicates that drivers earn more in the afternoon and least in the morning. To maximize earnings, drivers could focus on afternoon shifts or strategize to be available during those hours.
   - **Average Trip Length:** Trips are longer at night. For drivers who prefer fewer but longer rides, working at night might be more advantageous.
   - **Average Earning Per Mile:** The afternoon is also lucrative on a per-mile basis, reinforcing the strategy of prioritizing afternoons for work.

2. **For Ride-sharing Company Executives:**
   - **Average Total Pay and Earning Per Mile:** Knowing drivers earn more in the afternoons might suggest a higher demand, allowing executives to consider surge pricing or special offers during lower-income times to balance earnings throughout the day.
   - **Average Trip Length:** The longer trips at night might indicate that users are traveling to areas not accessible by other forms of public transport. This could be an opportunity to focus marketing efforts or provide incentives for drivers to meet this demand.
3. **For City Planners or Transport Authorities:**
   - **Average Total Pay and Trip Length:** These metrics suggest a higher afternoon and evening demand for transport services, potentially guiding infrastructure development or public transit scheduling.
4. **For Investors or Market Analysts:**
   - **All Three Metrics:** Insight into driver pay, trip length, and earnings per mile can inform predictions about company revenue and profitability, influencing investment decisions.

The insights gained could lead to targeted strategies, like incentive structures for drivers during low-earning times or marketing pushes to increase ride demand during usually slower periods. It could also influence operational strategies, such as fleet distribution and management to optimize availability where longer, more profitable trips are likely.

Challenges Encountered

In Task 4, I encountered the challenge of accurately computing average values in a large-scale, distributed environment. The primary difficulties included:

- **Ensuring Computational Accuracy**: With extensive data points, calculating precise averages to reflect driver pay and trip length across varying times of the day was complex and required careful execution.
- **Efficient Data Handling**: Managing and processing large datasets in Spark for multiple aggregation operations demanded efficient code and use of resources to prevent performance bottlenecks.

Overcoming the Challenges

To address these issues, I implemented the following solutions:

- **Robust Data Aggregation Techniques**: I utilized Spark's advanced aggregation functions to ensure accurate calculations of averages. This was critical in producing reliable metrics for driver total pay and trip length.
- **Optimized Spark Operations**: By optimizing Spark transformations and actions, I managed to efficiently process the large volume of data. I paid particular attention to the execution plans to avoid unnecessary shuffles and to cache intermediate results where applicable.

Insights

Calculated the average earnings per mile across different times of the day, providing a clear picture of the most profitable times for drivers to operate. This information can be used for dynamic pricing and scheduling.

Task 5: Finding Anomalies

Introduction

In Task 5, the objective was to scrutinize the average waiting times across the days of January to detect any anomalies. This task was crucial for identifying days with unusual wait times that could indicate operational hiccups or external factors impacting service efficiency.

Methodology

Data Extraction for January

I began by isolating the dataset to January alone, using date filtering techniques within Spark to ensure an accurate subset for analysis.

```
# Extract the January data
january_data = final_df.filter(month(col("date")) == 1)
```

Daily Average Waiting Time Calculation

The average waiting time for each day was computed using the `request_to_pickup` field, reflecting the period customers waited for their rideshare service.

```
# Calculate the average waiting time ('request_to_pickup' field) by day
average_waiting_time = january_data.withColumn("day", dayofmonth(col("date"))) \
                                   .groupBy("day") \
                                   .agg(F.avg("request_to_pickup").alias("average_waiting_time")) \
                                   .orderBy("day")

# Show the calculated average waiting times
average_waiting_time.show(31)

# Collect the data for plotting
average_waiting_time_pd = average_waiting_time.toPandas()

average_waiting_time.coalesce(1).write.mode("overwrite").options(header=True).csv(f"s3a://{s3_bucket}/task05/Q1")
```

Anomaly Identification

I set a threshold of 300 seconds to spotlight any extraordinary waiting times. Days exceeding this threshold were earmarked for further examination.

```
# Subtask 5b: Identify the days where the average waiting time exceeds 300 seconds
days_exceeding_300 = average_waiting_time_pd[average_waiting_time_pd['average_waiting_time'] > 300]['day'].tolist()
print(f"Days with average waiting time exceeding 300 seconds: {days_exceeding_300}")
```

Results

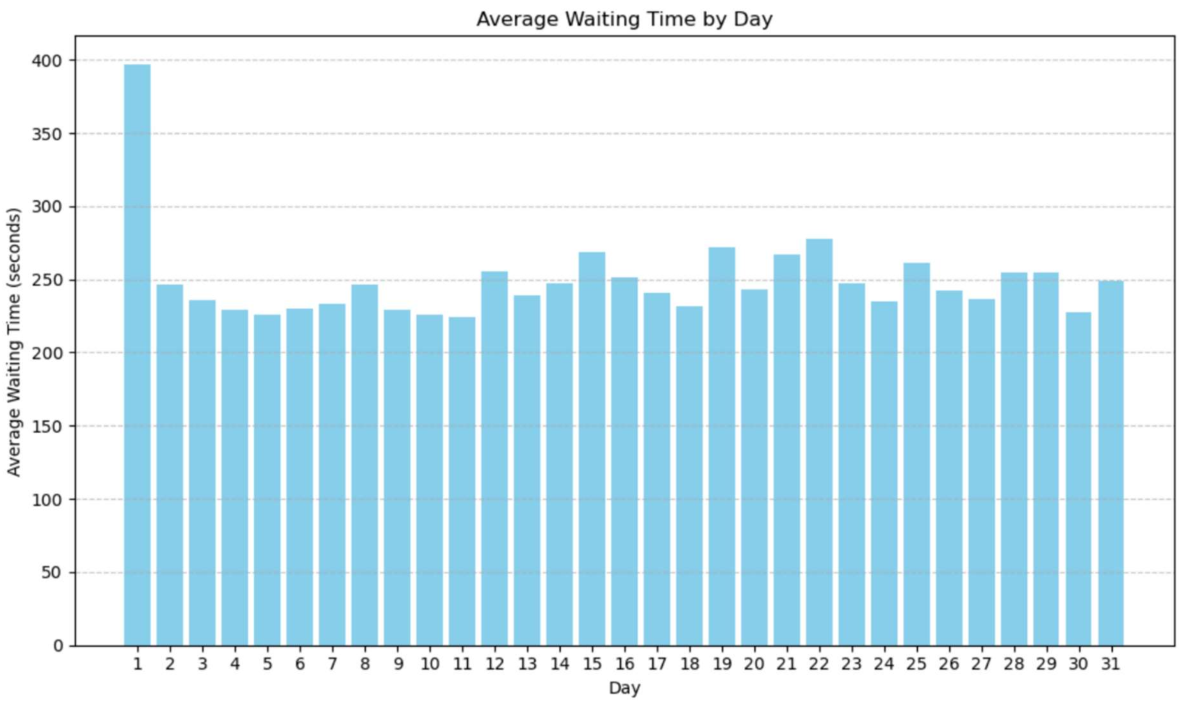
The analysis revealed that:

- Most days in January stayed below the 300-second waiting time threshold, indicating a consistent service level.
- The day when the average waiting time exceeded 300 seconds in January was day 1, as indicated in your provided text output.
- The longer average waiting time on New Year's Day could be attributed to a surge in demand as people use ride-sharing services to attend or return from celebrations. On top of that, road closures for festivities and increased traffic congestion due to public events are common, which can significantly impact travel times. Drivers may also be less available, as many might take time off to celebrate, leading to a shortage that can't meet the heightened demand. Understanding that these factors are likely the reason for the anomaly on this particular day helps to anticipate and plan for such occurrences in the future.

Visualization of Results

The calculated averages were visualized in a histogram to provide a transparent view of the waiting times across the month:

- **Figure 10**: Histogram illustrating Average Waiting Time by Day.



- **Figure 11**: Identification of days exceeding the 300-second average waiting time threshold.

```
Days with average waiting time exceeding 300 seconds: [1]
```

Challenges Encountered

- **Time Series Analysis**: Handling and analyzing time-series data within Spark required careful data manipulation and precise calculations.
- **Data Extraction**: Filtering the dataset for a specific month posed a challenge in maintaining focus solely on the target data.
- **Visualization**: The inability to visualize data within Spark necessitated exporting the data to visualize the results externally.

Overcoming the Challenges

- **Aggregation Techniques**: I utilized Spark's SQL functions for efficient and accurate aggregation of waiting times by day.
- **Filtering Accuracy**: Applying strict filtering criteria ensured that only January data was analyzed, maintaining the task's specificity.
- **External Visualization**: Data was exported and visualized outside of Spark to overcome the platform's limitations in this area.
- **Anomaly Detection**: Logical operations identified days with average waiting times beyond the normal range, revealing potential anomalies.

Insights

Identified days with unusual wait times which could indicate service disruptions or external events. This information can be critical for real-time response and long-term service improvements.

Task 6: Filtering Data

Introduction

Task 6 required detailed filtering of rideshare data to investigate trip counts across different pickup boroughs and times of day, and specifically to analyse trips between Brooklyn and Staten Island. This task aimed to provide insights into travel patterns and potentially uncover data on lesser-serviced areas or times.

Methodology

The task was split into three key objectives:

1. **Trips by Pickup Borough and Time of Day**: I filtered the dataset for trip counts greater than zero and less than a thousand, sorting by `Pickup_Borough` and `time_of_day`.

```python
# Task 6a: Find trip counts greater than 0 and less than 1000
trips_by_pickup_borough_time_of_day = final_df.groupBy("Pickup_Borough", "time_of_day") \
                                              .count() \
                                              .withColumnRenamed("count", "trip_count") \
                                              .filter((col("trip_count") > 0) & (col("trip_count") < 1000)) \
                                              .orderBy("Pickup_Borough", "time_of_day")

# Show the results for Task 6a
trips_by_pickup_borough_time_of_day.show()
```

2. **Evening Trips by Pickup Borough**: I specifically focused on trips occurring in the evening, again grouped by `Pickup_Borough`, to understand the distribution of service during this time.

```python
# Task 6b: Calculate the number of trips for each 'Pickup_Borough' in the evening
evening_trips_by_pickup_borough = final_df.filter(col("time_of_day") == "evening") \
                                          .groupBy("Pickup_Borough") \
                                          .count() \
                                          .withColumnRenamed("count", "trip_count") \
                                          .withColumn("time_of_day", F.lit("evening")) \
                                          .select("Pickup_Borough", "time_of_day", "trip_count") \
                                          .orderBy("Pickup_Borough")

# Show the results for Task 6b
evening_trips_by_pickup_borough.show()
```

3. **Brooklyn to Staten Island Trips**: I filtered the dataset for trips originating in Brooklyn and ending in Staten Island, providing a detailed view of this particular route.

```python
# task 6c
# A DataFrame is created by filtering the original 'final_df' DataFrame.
# This DataFrame contains only the records where the pickup was in Brooklyn and the dropoff was in Staten Island.
brooklyn_to_staten_island_df = final_df.filter(
    (F.col("Pickup_Borough") == "Brooklyn") &
    (F.col("Dropoff_Borough") == "Staten Island")
)

# The 'count()' action is used on the filtered DataFrame to find the total number of trips that meet the filter criteria.
# This count is stored in the variable 'num_trips'.
num_trips = brooklyn_to_staten_island_df.count()

# A smaller DataFrame 'brooklyn_to_staten_island_samples' is created by selecting only the relevant columns
# that identify the pickup borough, dropoff borough, and the specific pickup zone.
brooklyn_to_staten_island_samples = brooklyn_to_staten_island_df.select(
    "Pickup_Borough",
    "Dropoff_Borough",
    "Pickup_Zone"
)

# The first 10 records of the 'brooklyn_to_staten_island_samples' DataFrame are displayed.
# 'truncate=False' ensures that the data is shown completely without being cut off.
brooklyn_to_staten_island_samples.show(10, truncate=False)
```

Results

The results were as follows:

- For the first objective, I identified several boroughs with relatively low trip counts within specific times of the day, which might indicate opportunities for service improvement.
- The evening trips analysis showed significant variations in trip counts across different boroughs, with some, like Manhattan, showing substantially higher trip activity.
- The Brooklyn to Staten Island route analysis provided a snapshot of the zones within Brooklyn that most frequently initiated trips to Staten Island, which could help in strategic planning for service allocation.

Visualization of Results

Data was presented in tabular format to provide a concise view of the findings:

- **Figure 12**: Trip Counts by Pickup Borough and Time of Day

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

- **Figure 13**: Evening Trip Counts by Pickup Borough

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

- **Figure 14**: Top 10 Trips from Brooklyn to Staten Island

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

Figure 15: The number of trips from Brooklyn to Staten Island

```
Number of trips from Brooklyn to Staten Island: 69437
```

Challenges Encountered

- **Data Granularity**: Filtering data to such specific criteria required a granular approach, ensuring that no critical data was overlooked.

Overcoming the Challenges

Detailed Spark Queries: In addressing the challenge of data granularity and ensuring that no critical data was overlooked, the Spark SQL query that was utilized specifically is the `groupBy` function combined with `count`, `filter`, and `orderBy` methods.

For instance, to find the trip counts that are greater than 0 and less than 1000, a detailed Spark SQL query is constructed as follows:

```python
#Task 6a: Find trip counts greater than 0 and less than 1000
trips_by_pickup_borough_time_of_day = final_df.groupBy("Pickup_Borough", "time_of_day") \
                                              .count() \
                                              .withColumnRenamed("count", "trip_count") \
                                              .filter((col("trip_count") > 0) & (col("trip_count") < 1000)) \
                                              .orderBy("Pickup_Borough", "time_of_day")
```

This particular query groups the data by 'Pickup_Borough' and 'time_of_day', counts the occurrences, renames the resulting column for clarity, filters based on the specified criteria, and orders the results. It ensures that the analysis is precise and reflective of the specific subsets of data required for the task at hand.

Insights

Explored trip patterns across boroughs and times of day, uncovering potential underserved areas and times. Analysed specific routes, such as from Brooklyn to Staten Island, to identify service demand and planning routes efficiently.

Task 7: Routes Analysis

Introduction

Task 7 focused on evaluating the routes between pickup and dropoff zones to determine the top 10 most popular routes based on trip count. This assessment was aimed at identifying the routes that are most frequented by riders, which could have significant implications for strategic planning and resource allocation.

Methodology

To accomplish this task, the following steps were undertaken:

1. **Route Creation**: A new column, `Route`, was created by concatenating the `Pickup_Zone` and `Dropoff_Zone` to establish a clear identifier for each unique route.

```
final_df_with_routes = final_df.withColumn("Route", concat_ws(" to ", col("Pickup_Zone"), col("Dropoff_Zone")))
```

2. **Aggregation and Counting**: The data was then grouped by these routes, and trip counts were aggregated for both Uber and Lyft services to determine the total count for each route.

```
# Aggregate to find the total trip counts for each unique route for Uber and Lyft
# Assuming there's an indicator column 'business' in your DataFrame which distinguishes between Uber and Lyft
route_counts = final_df_with_routes.groupBy("Route").pivot("business").count()

# Calculate the total count across Uber and Lyft for each route
route_counts = route_counts.fillna(0)  # Replace nulls with 0s for accurate calculations
route_counts = route_counts.withColumn("total_count", col("Uber") + col("Lyft"))

# Rename the columns for clarity
route_counts = route_counts.withColumnRenamed("Uber", "uber_count") \
                           .withColumnRenamed("Lyft", "lyft_count")
```

3. **Ranking and Selection**: The routes were ranked by their total trip count, and the top 10 were selected for final analysis.

```
# Order the entire dataset by 'total_count' in descending order and take the top 10
top_routes = route_counts.orderBy(col("total_count").desc()).limit(10)

# Show the top 10 earnest routes without truncating the 'Route' names
top_routes.show(10, truncate=False)
```

Results

The analysis produced a ranked list of routes, showcasing the most to least popular routes within the dataset. The results were as follows:

- The most popular route was "JFK Airport to NA," indicating a high frequency of trips from the airport to various destinations.
- Other notable routes included "East New York to East New York" and "Canarsie to Canarsie," suggesting a high volume of intra-borough travel within these areas.
- The total trip counts provided a clear indication of route popularity and potentially high-demand areas.

Visualization of Results

The key findings were displayed in a tabular format:

- **Figure 16**: Top 10 Popular Routes with Uber and Lyft Counts

```
+----------------------------------------+----------+----------+-----------+
|Route                                   |lyft_count|uber_count|total_count|
+----------------------------------------+----------+----------+-----------+
|JFK Airport to NA                       |46        |253211    |253257     |
|East New York to East New York          |184       |202719    |202903     |
|Borough Park to Borough Park            |78        |155803    |155881     |
|LaGuardia Airport to NA                 |41        |151521    |151562     |
|Canarsie to Canarsie                    |26        |126253    |126279     |
|South Ozone Park to JFK Airport         |1770      |107392    |109162     |
|Crown Heights North to Crown Heights North|100     |98591     |98691      |
|Bay Ridge to Bay Ridge                  |300       |98274     |98574      |
|Astoria to Astoria                      |75        |90692     |90767      |
|Jackson Heights to Jackson Heights      |19        |89652     |89671      |
+----------------------------------------+----------+----------+-----------+
```

Challenges Encountered

- **Data Presentation**: Presenting the final data in a clear and concise manner required thoughtful consideration to ensure the insights were accessible.

Overcoming the Challenges

- **Clear Data Summarization**: Data was summarized in a non-truncated, tabulated format that allowed for immediate comparison and analysis.

Insights

Determined the most popular routes to understand where demand is highest, which is essential for fleet distribution and anticipating rider needs. Recognized the potential for targeted marketing campaigns in high-demand areas.