BigData 2025 Group 17

Figure 1: TartuLogo

Project Big Data is provided by University of Tartu.

Students: **Andres Caceres, Fidan Karimova, Moiz Ahmad, Siddiga Gadirova**

Introduction

This report presents an in-depth analysis of the New York City taxi dataset using Apache Spark. The primary goal is to examine taxi utilization, fare efficiency, and borough-based trip patterns to derive insights into urban mobility. The dataset includes key details such as trip durations, pickup and drop-off locations, and timestamps. Additionally, a GeoJSON file is used to map longitude-latitude coordinates to NYC boroughs for spatial analysis.

To address the core research objectives, we computed four main queries:

- Taxi Utilization Rate: Measuring the fraction of time a taxi is occupied by

passengers.
- Time to Find the Next Fare: Calculating the average idle time before a taxi picks up a new passenger, categorized by borough.
- Intra-Borough Trips: Counting trips that start and end within the same borough.
- Inter-Borough Trips: Counting trips that start in one borough and end in another.

By leveraging PySpark's distributed computing capabilities, along with geospatial and temporal processing, we analyze taxi service efficiency and urban travel patterns in NYC. This study provides insights that could help optimize taxi operations and improve transportation strategies in dense urban environments.

Files to be used

We have 3 notebooks in the repository:

- `checkpoint-1-sample.ipynb` - This notebook processes the sample data and does a small exploration of the data solving all the queries requested previously.
- `checkpoint-1-csv-to-parquet.ipynb` - This notebook pre-processes the production data and generates a parquet file to be used in the main notebook.
- `checkpoint-1-prod.ipynb` - This notebook processes the main data and solves all the queries requested previously.

Requirements

1. Dependencies are listed in the `requirements.txt` file
2. Run `docker compose up -d` in the terminal

How to run the project

Sample Data

- After the docker is up and running, open the `checkpoint-1-sample.ipynb` notebook and run all the cells. Ensure that `sample.csv` and `nyc-boroughs.geojson`.

Production Data

- Start by opening the `checkpoint-1-csv-to-parquet.ipynb` notebook and run all the cells. Ensure that `input/prod/trip_data_*.csv` is created, ensure the parquet file is created in the `output/prod/taxi_data.parquet` folder.
- Run the file `checkpoint-1-prod.ipynb` and visualize the results.

Observations

- There was a mismatch between the sample data and the production data related to the columns, there were 2 additional columns in the production data: `trip_distance` and `trip_time_in_secs`.
- For the production dataset, columns were removed before the parquet file was created. And then the processing happened on the parquet file. This is due to the fact that the reads are faster on parquet files.
- UDF functions were used to convert the longitude and latitude to boroughs. However these functions are not efficient and take a lot of time to process due to the fact that they run in Python.

Queries

1. Utilization: This is per taxi/driver. This can be computed by computing the idle time per taxi.

Timestamp Conversion

- Used `unix_timestamp` to convert string timestamps to numeric format
- Converted pickup and dropoff times to standardized unix timestamps:

```
df = df.withColumn("pickup_unix",
                   unix_timestamp(col("pickup_datetime"), "yyyy-MM-dd HH:mm:ss"))
df = df.withColumn("dropoff_unix",
                   unix_timestamp(col("dropoff_datetime"), "yyyy-MM-dd HH:mm:ss"))
```

Trip Duration Calculation

- Created a new `duration` column representing the time each trip took:

```
taxi_df = taxi_df.withColumn("duration", col("dropoff_unix") - col("pickup_unix"))
```

Window Functions for Idle Time

- Used PySpark's window functions to identify consecutive trips per taxi
- Created windows partitioned by taxi medallion and ordered by pickup time:

```
window_spec = Window.partitionBy("medallion").orderBy("pickup_unix")

taxi_df = taxi_df.withColumn("prev_dropoff_unix", lag("dropoff_unix").over(window_spec))

# Calculate idle time between trips
taxi_df = taxi_df.withColumn(
    "idle_time",
    when(col("prev_dropoff_unix").isNotNull(),
         when((col("pickup_unix") - col("prev_dropoff_unix")) <= four_hours_in_seconds,
              col("pickup_unix") - col("prev_dropoff_unix")
```

```
        ).otherwise(0)
    ).otherwise(0)
)
```

Aggregation for Utilization Metrics

- Used groupBy operations to calculate per-taxi metrics:

```
# Aggregate to compute totals per taxi
utilization_df = taxi_df.groupBy("medallion").agg(
    spark_sum("duration").alias("total_trip_time"),
    spark_sum("idle_time").alias("total_idle_time")
)

# Calculate utilization rate in grouped dataframe
utilization_df = utilization_df.withColumn(
    "utilization_rate",
    col("total_trip_time") / (col("total_trip_time") + col("total_idle_time"))
)
```

Vehicle Utilization Statistics Using `summary` from the dataframe it was possible to get the following statistics:

Statistic	total_trip_time (seconds)	total_trip_time (days)	total_idle_time (seconds)	total_idle_time (days)	utilization_rate
mean	9,369,729.377	108 days 10 hrs	10,493,901.201	121 days 10 hrs	48.59%
stddev	2,934,260.205	33 days 23 hrs	3,768,215.372	43 days 14 hrs	8.90%
min	1	0 days 0 hrs	0	0 days 0 hrs	0.76%
25%	8,435,580	97 days 15 hrs	8,870,100	102 days 15 hrs	44.83%
50%	10,126,844	117 days 5 hrs	11,188,662	129 days 11 hrs	47.43%
75%	11,290,790	130 days 16 hrs	13,151,771	152 days 5 hrs	50.40%
max	14,555,220	168 days 11 hrs	18,421,293	213 days 5 hrs	100.00%

We can see the average taxi in the fleet operated for a total of approximately 230 days (108 + 121) days as shown above. The average utilization rate is 48.59%, indicating that taxis are occupied for nearly half of their operational time, there might be some room for improvement in terms of utilization. Seeing the idle time is quite high per year (121 days on average), shows we can improve dispatching and do routing optimization to reduce idle time.

2. The average time it takes for a taxi to find its next fare(trip) per destination borough. This can be computed by finding the difference of time, e.g. in seconds, between the drop off of a trip and the pick up of the next trip.

The analysis of time to next fare per borough leverages PySpark's window functions to track sequential trips by the same taxi:

Window Function Implementation

```
taxi_window = Window.partitionBy("medallion").orderBy("dropoff_unix")
```

- Created a window partitioned by `medallion` (taxi ID)
- Ordered by `dropoff_unix` to sequence trips chronologically
- This window organizes all trips by a specific taxi in order of completion

Finding the Next Pickup Time

```
taxi_df = taxi_df.withColumn("next_pickup_unix", lead("pickup_unix").over(taxi_window))
```

- Used the `lead()` function to look forward in the window
- For each trip, identified the next pickup time by the same taxi
- `lead()` reaches into the "future" of the ordered window to find the next value

Calculating Time to Next Fare

```
taxi_df = taxi_df.withColumn(
    "time_to_next_fare",
    when(
        (col("next_pickup_unix").isNotNull()) & (col("next_pickup_unix") >= col("dropoff_uni
        col("next_pickup_unix") - col("dropoff_unix")
    ).otherwise(None)  # Ignore invalid (negative) idle times
)
```

- Calculated idle time by finding the difference between:
  - Current trip's dropoff time
  - Next trip's pickup time
- Included validation to handle edge cases:
  - Ensured the next pickup is after current dropoff
  - Used `None` for last trips (no next pickup)

Borough-Level Aggregation

```
next_fare_df = taxi_df \
    .filter(col("time_to_next_fare").isNotNull()) \
    .groupBy("dropoff_borough") \
    .agg(avg("time_to_next_fare").alias("avg_time_to_next_fare"))
```

- Filtered out null values (last trips for each taxi)
- Grouped by `dropoff_borough` to analyze by area
- Used `avg()` aggregation to find the mean wait time per borough

The `lead()` function is crucial here - while `lag()` looks at previous values in a window, `lead()` looks at upcoming values, making it perfect for analyzing what happens *after* each trip.

The results of the analysis are as follows:

dropoff_borough	avg_time_to_next_fare (seconds)	avg_time_to_next_fare (hours)
Queens	5452.72	1h 30m 52s
Unknown	1495.17	0h 24m 55s
Manhattan	5703.44	1h 35m 3s
Staten Island	7951.62	2h 12m 31s

This shows the average wait time for drivers to get their next fare after dropping off passengers in different boroughs. Staten Island has the longest wait time at over 2 hours, while the "Unknown" category has the shortest at under 25 minutes. Manhattan and Queens both have wait times of approximately 1.5 hours.

3. The number of trips that started and ended within the same borough

This query identifies trips where both the pickup and drop-off locations belong to the same borough. These intra-borough trips help understand local taxi demand within boroughs.

```
borough_polygons = {}
for _, row in geojson_data.iterrows():
    borough_polygons[row['borough']] = row['geometry']
```

The first line initializes an empty dictionary to store borough names as keys and their geometries as values. The loop will iterate through geojson data and add an entry to the dictionary mapping the borough name to its geometry.

```
borough_broadcast = spark.sparkContext.broadcast(borough_polygons)
```

This line broadcasts the `borough_polygons` dictionary to all worker nodes in a distributed Spark environment. It ensures efficient lookup of borough geometries without redundant data transfers between nodes.

```
def get_borough(lon, lat):
    try:
        lon, lat = float(lon), float(lat)
        point = Point(lon, lat)
```

```python
        print(f"Checking: lon={lon}, lat={lat}")
        for borough, polygon in borough_broadcast.value.items():
            if polygon.contains(point):
                print(f"Matched: {lon}, {lat} -> {borough}")
                return borough
    except Exception as e:
        print(f"Error processing ({lon}, {lat}): {e}")
    return "Unknown"
# Register the function as a Spark UDF again
to_borough_udf = spark.udf.register("to_borough", get_borough, StringType())
```

This code defines the `get_borough function`, which determines the borough for a given longitude (lon) and latitude (lat).

- It first converts the coordinates to floats and creates a `Point` object.
- It then iterates over `borough_broadcast.value`, checking if the point is inside any borough's polygon using .contains().
- If a match is found, it returns the borough name; otherwise, it returns `Unknown`.
- Finally, the function is registered as a Spark UDF (User-Defined Function) called `to_borough` for use in Spark SQL queries.

```python
taxi_df = taxi_df.withColumn("pickup_borough", to_borough_udf(col("pickup_longitude"), col("
taxi_df = taxi_df.withColumn("dropoff_borough", to_borough_udf(col("dropoff_longitude"), col
```

This line adds two new columns `pickup_borough` and `dropoff_borough` to the taxi_df DataFrame by applying the to_borough_udf function to each row's pickup and dropoff coordinates.

```python
same_borough_df = taxi_df.filter(col("pickup_borough") == col("dropoff_borough"))
same_borough_count = same_borough_df.groupBy("pickup_borough").agg(count("medallion").alias(
```

The code filters taxi trips where the pickup and dropoff boroughs are the same. Then, it groups by `pickup_borough` and counts the number of such trips.

Results:

pickup_borough	same_borough_trips
Queens	224,883
Unknown	15,785,053
Staten Island	3,179
Manhattan	57

This table shows the number of taxi trips that both started and ended in the same borough. The "Unknown" category has by far the highest number at over 15.7 million trips, followed by Queens with about 224,883 trips. Staten Island has significantly fewer at 3,179 trips, while Manhattan has only 57 same-borough trips in this dataset.

4. The number of trips that started in one borough and ended in another one

This is exactly the same as Query 3 but the only difference is it will show the trips which started in one borough and ended in another. Major part of implementation will remain the same as query only difference will come in comparing the columns.

```
diff_borough_df = taxi_df.filter(col("pickup_borough") != col("dropoff_borough"))
diff_borough_count = diff_borough_df.groupBy("pickup_borough", "dropoff_borough").agg(count(
```

The code filters taxi trips where the pickup and dropoff boroughs are not the same. Then, it groups by `pickup_borough` and `dropoff_borough` and counts the number of such trips.

Results:

pickup_borough	dropoff_borough	cross_borough_trips
Queens	Unknown	6,324,787
Unknown	Queens	6,195,547
Unknown	Staten Island	24,341
Queens	Staten Island	7,710
Unknown	Manhattan	5,402
Manhattan	Unknown	1,010
Queens	Manhattan	731
Staten Island	Unknown	650
Staten Island	Queens	46
Manhattan	Queens	15

This table shows the number of taxi trips between different boroughs. The highest volume is for trips from Queens to Unknown locations (6.3M trips) and from Unknown locations to Queens (6.2M trips). The least common cross-borough trips in this dataset are from Manhattan to Queens, with only 15 recorded trips.

Conclusion

Our analysis highlights significant patterns in taxi utilization and travel efficiency across NYC boroughs. The results show varying idle times, borough-specific demand trends, and differences in trip frequencies within and between boroughs. These insights can help optimize taxi operations, reduce idle times, and improve urban mobility strategies.