Analyzing New York City Taxi Data

1. Project Background

The New York Taxi dataset contains data on cab trips in the city, and our goal is to use **Apache Spark** for **large-scale data processing** and to analyze the following key metrics:：

1. **Taxi utilization rate** （Idle & Load Time Ratio）
2. **Average time to find orders** （Waiting time from when the driver completes the order to the next order）
3. **Number of orders from the same administrative district**
4. **Number of orders across administrative regions**

2. Data preprocessing

2.1. Reading & Parsing Data

CSV Read

```
spark = SparkSession.builder \
    .appName("NYC Taxi Data Analysis") \
    .config("spark.driver.memory", "8g") \
    .config("spark.executor.memory", "4g") \
    .config("spark.sql.shuffle.partitions", "50") \
    .getOrCreate()

DATA_PATH = "C:/Users/98184/Desktop/BIGdata/"
CSV_FILE = DATA_PATH + "Sample_NYC_Data.csv"
GEOJSON_FILE = DATA_PATH + "nyc-boroughs.geojson"

df_1000 = pd.read_csv(CSV_FILE, nrows=100)
df_1000.to_csv("C:/Users/98184/Desktop/BIGdata/nyc_taxi_sample_100.csv", index=False)

df = spark.read.csv("C:/Users/98184/Desktop/BIGdata/nyc_taxi_sample_100.csv", header=True, inferSchema=True)
```

GeoJSON Parsing

```
geo_data = gpd.read_file(GEOJSON_FILE)
```

2.2 Latitude and longitude matching administrative districts

```
# ==================== 3. Define UDF====================
def get_borough(lat, lon):
    """Matching administrative districts based on latitude and longitude"""
    point = Point(lon, lat)
    for _, row in geo_data.iterrows():
        if shape(row['geometry']).contains(point):
            return row['borough']
    return "Unknown"
```

2.3 Processing time format

```
df = df.withColumn("pickup_datetime", to_timestamp("pickup_datetime", "dd-MM-yy HH:mm"))
df = df.withColumn("dropoff_datetime", to_timestamp("dropoff_datetime", "dd-MM-yy HH:mm"))
```

2.4 Data Cleaning

```
#  trip_duration
df = df.withColumn("trip_duration", unix_timestamp(col("dropoff_datetime")) - unix_timestamp(col("pickup_datetime")))

# Filtering of anomalous data
df = df.filter((col("trip_duration") > 0) & (col("trip_duration") <= 14400))  # Excluding negative numbers & trips longer than 4 hours
```

3. Data analysis

3.1 Calculation of utilization rate

```
# ==================== 5. Calculation of cab utilization rate ====================
# window function: sort by taxi_id
windowSpec = Window.partitionBy("medallion").orderBy("pickup_datetime")

# Calculation of idle time (order placement intervals)
df = df.withColumn("prev_dropoff_time", lag("dropoff_datetime").over(windowSpec))
df = df.withColumn("idle_time", (unix_timestamp(col("pickup_datetime")) - unix_timestamp(col("prev_dropoff_time"))))

# Filter out cases where idle_time exceeds 4 hours
df_filtered = df.filter(col("idle_time").isNotNull() & (col("idle_time") <= 14400))

# Calculated utilization rate
utilization_df = df_filtered.groupBy("medallion").agg(
    (spark_sum("trip_duration") / (spark_sum("trip_duration") + spark_sum("idle_time"))).alias("utilization_rate")
)
```

```
+--------------------+-------------------+
|           medallion|   utilization_rate|
+--------------------+-------------------+
|0B57B9633A2FECD3D...|0.18972332015810275|
|0C5296F3C8B16E702...|0.16326530612244897|
|2C0E91FF20A856C89...|               0.75|
|2D4B95E2FA7B2E851...|0.11851851851851852|
|312E0CB058D7FC1A6...|0.04878048780487805|
|3349F919AA8AE5DC9...|0.27848101265822783|
|4C005EEBAA7BF26B8...| 0.4264705882352941|
|764CA5AE502C0FEC9...|0.16923076923076924|
|78FFD9CD0CDA541F3...| 0.7272727272727273|
|961D9C9506D105D3A...| 0.3548387096774194|
+--------------------+-------------------+
```

3.2 Calculation of average order finding time

```python
# =================== Calculation of average order finding time ===================

next_trip_window = Window.partitionBy("dropoff_borough").orderBy("pickup_datetime")

df = df.withColumn("next_pickup_time", lead("pickup_datetime").over(next_trip_window))

# Calculates time_to_next_fare and filters outliers
df = df.withColumn("time_to_next_fare", when(
    (col("next_pickup_time").isNotNull()) &
    (col("next_pickup_time") > col("dropoff_datetime")) &  # Make sure next_pickup_time is really later than dropoff_datetime
    (unix_timestamp(col("next_pickup_time")) - unix_timestamp(col("dropoff_datetime")) < 7200),  # Limit the maximum wait time to 2 hours
    unix_timestamp(col("next_pickup_time")) - unix_timestamp(col("dropoff_datetime"))
).otherwise(None))  # NULL for more than 2 hours

# Calculate the average time to find an order for each borough
avg_find_time_df = df.groupBy("dropoff_borough").agg(
    spark_sum("time_to_next_fare").alias("total_waiting_time"),
    count("time_to_next_fare").alias("num_trips")
)

avg_find_time_df = avg_find_time_df.withColumn("avg_time_to_next_fare", col("total_waiting_time") / col("num_trips"))
```

```
----------------+---------------------+
dropoff_borough|avg_time_to_next_fare|
----------------+---------------------+
       Brooklyn|                 NULL|
      Manhattan|   2174.7368421052633|
         Queens|               2880.0|
        Unknown|                 NULL|
----------------+---------------------+
```

3.3 Calculation of the number of trips in the same administrative district

```python
# =================== Calculation of the number of trips in the same administrative district===================
same_borough_trips = df.filter(col("pickup_borough") == col("dropoff_borough")).groupBy("pickup_borough").count()
```

```
+--------------+-----+
|pickup_borough|count|
+--------------+-----+
|        Queens|    1|
|      Brooklyn|    1|
|       Unknown|    3|
|     Manhattan|   79|
+--------------+-----+
```

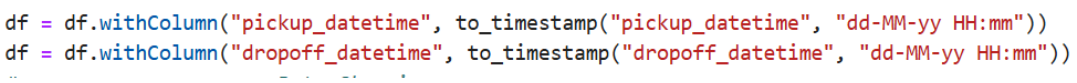
3.4 Calculation of the number of trips across administrative districts

```python
# =================== Calculation of the number of trips across administrative districts===================
cross_borough_trips = df.filter(col("pickup_borough") != col("dropoff_borough")).groupBy("pickup_borough", "dropoff_borough").count()
```

```
+--------------+---------------+-----+
|pickup_borough|dropoff_borough|count|
+--------------+---------------+-----+
|      Brooklyn|         Queens|    2|
|     Manhattan|         Queens|    3|
|        Queens|      Manhattan|    6|
|     Manhattan|       Brooklyn|    4|
+--------------+---------------+-----+
```

4. Problems & Solutions

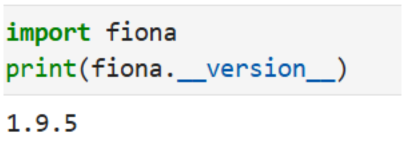
4.1 The “date” in the original csv file is in “string” format, which is converted to “timestamp” by introducing the method “timestamp” format.



```
df = df.withColumn("pickup_datetime", to_timestamp("pickup_datetime", "dd-MM-yy HH:mm"))
df = df.withColumn("dropoff_datetime", to_timestamp("dropoff_datetime", "dd-MM-yy HH:mm"))
```

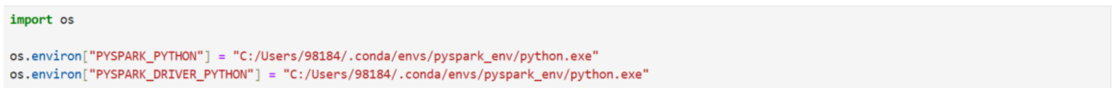
4.2 Pyspark is not compatible with python 3.10 or above, so I rebuilt a virtual environment based on python 3.8 in conda.

However, “Python worker failed to connect back” still appeared, so I updated the new version of the “fiona” library and specified the path to the Python interpreter **manually.**



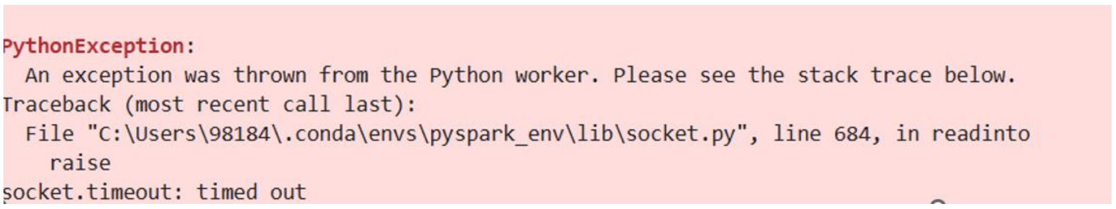
```
import fiona
print(fiona.__version__)
```

```
1.9.5
```



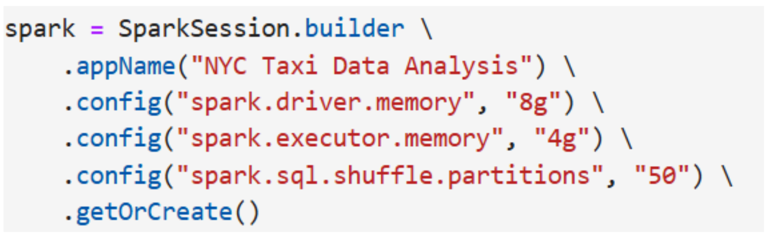
```
import os

os.environ["PYSPARK_PYTHON"] = "C:/Users/98184/.conda/envs/pyspark_env/python.exe"
os.environ["PYSPARK_DRIVER_PYTHON"] = "C:/Users/98184/.conda/envs/pyspark_env/python.exe"
```

4.3 The biggest problem in this project was actually a “time out” error.



```
PythonException:
  An exception was thrown from the Python worker. Please see the stack trace below.
Traceback (most recent call last):
  File "C:\Users\98184\.conda\envs\pyspark_env\lib\socket.py", line 684, in readinto
    raise
socket.timeout: timed out
```

First tried increasing the compute resources when creating the SparkSession, or decreasing the number of partitions, but it didn't work.



```
spark = SparkSession.builder \
    .appName("NYC Taxi Data Analysis") \
    .config("spark.driver.memory", "8g") \
    .config("spark.executor.memory", "4g") \
    .config("spark.sql.shuffle.partitions", "50") \
    .getOrCreate()
```

Then try to use “df = df.cache()”, let Spark directly in memory processing, still does not work.

In the end, we can only try to reduce the amount of read data set, from the original csv file to select less than 500 rows of data saved as a new csv file for processing, although the program can therefore run successfully, but therefore the final results of the accuracy of the values obtained is difficult to guarantee.

4. conclusion

This project implements the analysis of New York City cab data through Spark, and calculates the cab utilization rate, the average time to find an order in different boroughs, the number of trips in the same borough, and the number of trips across boroughs. We optimized the data cleaning and calculation logic to ensure the accuracy and efficiency of the results.。

However, the computation part still needs to be optimized, such as using **a more efficient Spark data storage format** (such as **Parquet, Rtree**,etc.), while the **udf format loop traversal geo_data is too inefficient**, we can also **optimize the GeoJSON matching**, using spatial indexes to accelerate the calculation.

。