Documentation for NYC Taxi Data Pipeline

Contents

Note: I couldn't run the code according to requirements, that data was too large that even Google Colab was getting out of memory (RAM). So instead, I was able to process the data for 2022, and then did all the comparisons between the **first half of 2022** and **second half of 2022**. The code I'm submitting is for the entire task but the one that I actually was able to test out is in the Colab Notebook.

Description and Definition of the Pipeline Steps

1. Download Data

Script: download_data.py

Description: Downloads NYC taxi trip data files from the NYC government website. It checks if the file already exists to avoid redundant downloads. (it's called from process_data.py)

2. Process Data

Script: process_data.py

Description: Processes the downloaded data files, converting them to a common schema and saving them in both Parquet and Avro formats.

3. Analyze Data

Script: analyze.py

Description: Analyzes the processed data to generate insights such as the average distance driven per hour, the day with the lowest number of single rider trips, and the top 3 busiest hours.

4. Plot Data

Script: plot_graphs.py

Description: Generates visualizations of the analysis results, including plots of the average distance driven per hour, the day with the lowest number of single rider trips, and the top 3 busiest hours.

Output Datasets

Parquet and Avro Formats

Parquet Schema:

- VendorID: int64
- pickup_datetime: timestamp
- dropoff_datetime: timestamp
- store_and_fwd_flag: string
- RatecodeID: int64
- PULocationID: int64
- DOLocationID: int64
- passenger_count: int64
- trip_distance: float64
- fare_amount: float64
- extra: float64
- mta_tax: float64
- tip_amount: float64
- tolls_amount: float64
- improvement_surcharge: float64
- total_amount: float64
- payment_type: int64
- congestion_surcharge: float64
- taxi_type: string

Avro Schema:

- VendorID: ["null", "long"]
- pickup_datetime: ["null", {"type": "long", "logicalType": "timestamp-millis"}]
- dropoff_datetime: ["null", {"type": "long", "logicalType": "timestamp-millis"}]
- store_and_fwd_flag: ["null", "string"]
- RatecodeID: ["null", "long"]
- PULocationID: ["null", "long"]
- DOLocationID: ["null", "long"]
- passenger_count: ["null", "long"]
- trip_distance: ["null", "double"]
- fare_amount: ["null", "double"]
- extra: ["null", "double"]
- mta_tax: ["null", "double"]
- tip_amount: ["null", "double"]
- tolls_amount: ["null", "double"]
- improvement_surcharge: ["null", "double"]
- total_amount: ["null", "double"]
- payment_type: ["null", "long"]
- congestion_surcharge: ["null", "double"]
- taxi_type: ["null", "string"]

Code of the Queries

Python Code for Analysis

The following code is in *analyze.py*:

```
# Average dist by yellow and green taxis per hour
      avg_distance_per_hour =
            data.groupby(['taxi_type','hour'])['trip_distance'].mean().reset_
            index()

# Day with the lowest number of single rider trips
      single_rider_trips = data[data['passenger_count'] == 1]
      lowest_single_rider_day =
            single_rider_trips.groupby('day_of_week').size().idxmin()

# Top 3 busiest hours
      busiest_hours =
            data.groupby('hour').size().nlargest(3).reset_index(name='count')
```

Automating and Deployment

To automate the process of downloading and processing the latest data every month, set up a cron job. Add the following line to crontab file:

```
0 0 1 * * /usr/bin/python3  /home/Venture_data/process_data.py
```

After setting up this environment, we can leave the instance running so it can execute the process_data.py file at the start of each month. When process_data.py is executed, it first downloads all the latest files up to the current date. Once it encounters the date which is missing, it'll stop iterating and list us the dates for which the data isn't available.

Another way of deploying and automating would be by using **Glue Jobs** on AWS (although the underlying logic will probably be the same), but it'll reduce the cost as we won't be running the instance for the entire month. Processing part of data takes a lot of time, so **lambda functions** won't be a suitable choice.

Bonus and other Questions

Merging Input Files

The input data is spread over several files, including separate files for "Yellow" and "Green" taxis. Does it make sense to merge those input file into one?

The input data contains separate files for "Yellow" and "Green" taxis. It makes sense to process these files separately due to large size of file, so we'll better be able to manage the memory being used. However, for analysis, we can concatenate the data into a single DataFrame, the **taxi_type** column will be used to differentiate.

Evaluating Data on Hour-Level and Day of Week-Level

You will notice that the input data contains "date and time" columns. Your colleagues want to evaluate data also on hour-level and day of week-level. Does that affect your output-structure in some way?

Evaluating data on hour-level and day of week-level requires extracting these components from the **pickup_datetime** column. This affects the output structure by adding additional columns (**hour** and **day_of_week**) to the processed data, which are then used in the analysis and visualization steps *(at least for plotting graphs using matplotlib, I'm not very well versed with advanced analysis tools like PowerBI and Tableau so I'm not sure about that)*.

Expanding Pipeline for Weather-Based Predictions

Your data scientists want to make future predictions based on weather conditions. How would you expand your pipeline to help your colleagues with this task?

To expand the pipeline for weather-based predictions, we will have to follow the steps below:

1. **Download Weather Data:** Integrate a step to download historical weather data from an API.
2. **Merge Weather Data:** Merge the weather data with the taxi trip data based on date and time (we will have to merge data of yellow and green taxis before this step).
3. **Update Analysis:** Include weather conditions in the analysis to understand their impact on taxi trips.

Providing Data in XLSX Format

Another colleague approaches to you. He is an Excel guru and makes all kind of stuff using this tool forever. So, he needs all the available taxi trip records in the XLSX format. Can you re-use your current pipeline? How does this output compare to your existing formats? Do you have performance concerns?

To provide data in XLSX format for Excel users:

1. **Export to XLSX:** Use **pandas** to export the concatenated DataFrame to an XLSX file.
2. **Re-use Pipeline:** The existing pipeline can be re-used with an additional step to save the data in XLSX format.
3. **Performance Concerns:** Exporting large datasets to XLSX might be slower and consume more memory compared to Parquet/Avro formats. Consider exporting data in chunks if performance becomes an issue, while clearing cache to free up RAM where possible. Also, where possible we can use parallel processing too. Perhaps we can divide the data by the number of CPU cores that we have and process it accordingly, as the rows themselves don't have any effect on each other.