Yet another NYC Yellow taxi dataset problem

Problem Context

Using NYC "Yellow Taxi" Trips Data 2019 together with data about the different zones, calculate the hour of the day where trips between any two given zones are outliers. The result should be the hours and the zones (by name).

The data consists of a set of CSV files with yellow taxi trip records. The TLC Authority web provides a brief description of the dataset: *"Records include fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts. The records were collected and provided to the NYC Taxi and Limousine Commission (TLC) by technology service providers. The trip data was not created by the TLC, and TLC cannot guarantee their accuracy."*

The last part about the accuracy is very true. For the 2019's dataset I found different dates from other years that 2019. So following some best practices we should "clean" the dataset before starting our work and thus we reduce the number of outliers cleaning beforehand.

So, to get a glipmse of the dataset structure we have to take a look at the data dictionary, also provided by the TCL Authority:

https://www1.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf

After that, we can see that we have to work with some Datetime types from the pickup and dropoff columns. Also other important properties for our problem are the PuLocationID and the DoLocationID as they are ids and we need to get the zone name (text) for our study results. The zone name is not in the dataset, and we have to get it from another descriptive dataset as stated in the TLC Authority website:

"PULocationid & DOLocationID matching zone numbers to the map Each of the trip records contains a field corresponding to the location of the pickup or drop-off of the trip (or in FHV records before 2017, just the pickup), populated by numbers ranging from 1-263. These numbers correspond to taxi zones, which may be downloaded as a table or map/shapefile and matched to the trip records using a join. The data is currently available on the Open Data Portal at: https://data.cityofnewyork.us/Transportation/NYC-Taxi-Zones/d3c5-ddgc"

If we inspect the new table with the zone names we can see that, we indeed need to join both datasets to get the zone name, as stated in the description above.

Using the Tinybird platform to get the solution

So as we are going to use Tinybird as our analytics platform we need to create a couple of datasources, using the cli tool, one for the main yellow taxi trips and another for the zone name matching.

It is advisable to check the Cli tool tutorial at https://docs.tinybird.co/cli.html

First, we have to download, from the TLC website the first 3 months CSVs from 2019 year. Next, we are going to create a main datasource named yellow_tripdata_2019_q1 and another one for matching the zones with the name taxi_zone_lookup:

```
tb datasource generate ./yellow_tripdata_2019_q1.csv
** Generated datasources/yellow_tripdata_2019_q1.datasource
**   => Run `tb push datasources/yellow_tripdata_2019_q1.datasource` to create it on the server
**   => Add data with `tb datasource append yellow_tripdata_2019_q1 ./yellow_tripdata_2019_q1_2sample.csv`
**   => Generated fixture datasources/fixtures/yellow_tripdata_2019_q1.csv
```

Then we need to append the next two months to the datasource with `tb datasource append` and create another datasource for the matching zones *taxi_zone_lookup*.

The main problem we're trying to solve is to obtain the outliers for the trips between two zones, but before tackling the main problem we need to clean and prepare the dataset. The main prerequisite is that the output should look something like this:

```
----------------------------------------
pickup_datetime: 2019-03-29 12:36:26
dropoff_datetime: 2019-03-29 12:51:42
puzone: Yorkville West
dozone: Upper East Side North
passenger_count: 1
trip_distance: 3.29
ratecodeid: 1
store_and_fwd_flag: N
payment_type: 1
fare_amount: 13.5
extra: 0.5
mta_tax: 0.5
tip_amount: 3.46
tolls_amount: 0
improvement_surcharge: 0.3
total_amount: 20.76
congestion_surcharge: 2.5
----------------------------------------
```

And not like this

```
vendorid: 2
tpep_pickup_datetime: 2019-03-29 12:36:26
tpep_dropoff_datetime: 2019-03-29 12:51:42
pulocationid: 230
dolocationid: 262
passenger_count: 1
trip_distance: 3.29
ratecodeid: 1
store_and_fwd_flag: N
payment_type: 1
fare_amount: 13.5
extra: 0.5
```

```
mta_tax: 0.5
tip_amount: 3.46
tolls_amount: 0
improvement_surcharge: 0.3
total_amount: 20.76
congestion_surcharge: 2.5
```

So we first create a new source to clean and prepare the data and this new source will have the pickup and dropoff zone names added. Tinybird can manage materialized views, a very nice feature to ingest and transform data on the fly. Materialized views are recommended for this type of work: ingest data and add new aggregated or joined columns from another datasource (denormalize). Aggregate and joins are expensive operations that we can do it once, during the ingesting process and leave the dataset optimized for the analytic operations.

Again we use the cli tool to create this datasource by creating a file in the datasource directory named *nyc_taxi_zone_clean*:

```
SCHEMA >
    `pickup_datetime` DateTime,
    `dropoff_datetime` DateTime,
    `pulocationid` Int32,
    `dolocationid` Int32,
    `puzone` String,
    `dozone` String,
    `passenger_count` Int16,
    `trip_distance` Float32,
    `ratecodeid` Int16,
    `payment_type` Int16,
    `fare_amount` Int32,
    `extra` Int32,
    `mta_tax` Int32,
    `tip_amount` Int32,
    `tolls_amount` Int32,
    `total_amount` Int32

ENGINE "MergeTree"
ENGINE_SORTING_KEY "pickup_datetime, dropoff_datetime"
```

Because we're going to join two datasources we should be using ClickHouse's JOIN engine for our materialized view, but due to that the "dimension" taxi_zone_names table has very few rows:

```
SELECT count(*) FROM taxi_zone_lookup
(256)
```

the performance difference would be neglegible using the MergeTree engine instead of the Join. On the contrary if we had a pretty bigger fact and dimension tables then it would be more performant to use the Join engine. Nevertheless I created the data source with the join engine and a pipe to test the performance. You can find it in the repo code with the names *nyc_taxi_join* and *nyc_taxi_join_pipe*.

The query with the MergeTree engine executes in 31.74ms, processing 23M rows.

```
SELECT
        ytp.tpep_pickup_datetime AS pickup_datetime,
        ytp.tpep_dropoff_datetime AS dropoff_datetime,
        ytp.pulocationid,
        ytp.dolocationid,
        tzu.zone AS puzone,
        tzo.zone AS dozone,
        ytp.passenger_count,
        ytp.trip_distance,
        ytp.ratecodeid,
        ytp.payment_type,
        ytp.fare_amount,
        ytp.extra,
        ytp.mta_tax,
        ytp.tip_amount,
        ytp.tolls_amount,
        ytp.total_amount
    FROM yellow_tripdata_2019_q1 ytp
    INNER JOIN taxi_zone_lookup tzu ON ytp.pulocationid = tzu.locationid
    INNER JOIN taxi_zone_lookup tzo ON ytp.dolocationid = tzo.locationid
```

To test the difference, just create a node with this query in the *nyc_taxi_join_pipe*

After that we need a pipe *nyc_taxi_zone_clean_pipe* to transform and clean the data until we get to the desired dataset state to begin with the main problem.

The pipe:

```
NODE select_zone_name
DESCRIPTION>
        Joins the two datasources into a materialized view
        with two new columns with the pickup and dropoff zone names (puzone, do zone)
SQL >

    SELECT
        ytp.tpep_pickup_datetime AS pickup_datetime,
        ytp.tpep_dropoff_datetime AS dropoff_datetime,
        ytp.pulocationid AS pulocationid,
        ytp.dolocationid AS dolocationid,
        tzu.zone AS puzone,
        tzo.zone AS dozone,
        ytp.passenger_count AS passenger_count,
        ytp.trip_distance AS trip_distance,
        ytp.ratecodeid AS ratecodeid,
        ytp.payment_type AS payment_type,
        ytp.fare_amount AS fare_amount,
        ytp.extra AS extra,
        ytp.mta_tax AS mta_tax,
```

```
        ytp.tip_amount AS tip_amount,
        ytp.tolls_amount AS tolls_amount,
        ytp.total_amount AS total_amount
    FROM yellow_tripdata_2019_q1 ytp
    JOIN taxi_zone_lookup tzu ON ytp.pulocationid = tzu.locationid
    JOIN taxi_zone_lookup tzo ON ytp.dolocationid = tzo.locationid

NODE different_zones_2019
DESCRIPTION>
Gettting only the 2019 trips between different zones

SQL >
    SELECT pickup_datetime,
            dropoff_datetime,
            pulocationid,
            dolocationid,
            puzone,
            dozone,
            passenger_count,
            trip_distance,
            ratecodeid,
            payment_type,
            fare_amount,
            extra,
            mta_tax,
            tip_amount,
            tolls_amount,
            total_amount
    FROM select_zone_name
    WHERE pulocationid != dolocationid AND
            toYear(pickup_datetime) = 2019 AND
            toYear(dropoff_datetime) = 2019

TYPE materialized
DATASOURCE  nyc_taxi_zone_clean
```

Describing and solving the main problem

Now the main problem is this: we have to get the trip outliers between any two different zones. For the sake of simplicity of our model I'm going to explain what an outlier value is and how to simply detect them using simple statistical concepts like the average and standard deviation. We could write a piece of python code using numpy or pandas to get a more detailed analysis but as I said, let's start simple.

Outliers or anomalies are values that are out of the normal distribution of the dataset. Having a quick look at it we can see that there are trips with different pickup and dropoff dates (very long trips of > than 8h in Manhattan?) or a passenger_count of 9, when the average passenger is between 1 and 2. So we have detect these anomalies with a couple of easy statistical functions. The original dataset has many variables (columns) that can be studied like fare_amount but for the sake of the simplicity we're going to work with trip time (difference between the pickup and dropoff times) the passenger_count and trip_distance. We can extrapolate this model to work other variables like fare_amount.

A good place to start looking for a reasonable value are the mean and we can use the standard deviation to get a range of possible values. Also with these two functions we can build a score to detect if a value is within an acceptable range: The **Z-score**

The *Z-score* or *standard score* is the number of standard deviations from the mean. If we work with the mean and standard deviation creating an upper and lower limit, our acceptable range will be one standard deviation from the mean: a Z-score in the range ±1. With the Z-score we can move the upper and lower bounds to get more precise outlier results.

The model for the passenger_count variable:

(passenger_count - passenger_series_avg) / passenger_series_stddev AS passenger_zscore

So we can calculate the mean and standard deviation of the dataset for the passenger_count, trip_distance (miles) and trip_time (minutes using the function **dateDiff**) variables with a query to the materialized view:

```
SELECT
        avg(passenger_count) AS avg_passenger,
        avg(trip_distance) AS avg_distance,
        avg(dateDiff('minute',pickup_datetime,dropoff_datetime)) AS avg_time,
        stddevPop(passenger_count) AS std_passenger,
        stddevPop(trip_distance) AS std_distance,
        stddevPop(dateDiff('minute',pickup_datetime,dropoff_datetime)) AS std_time
FROM nyc_taxi_zone_clean
```

```
-----------------------------------------
avg_passenger: 1.5756512724734397
avg_distance: 3.0367553475515585
avg_time: 17.780161769434212
std_passenger: 1.2302451685284468
std_distance: 3.8699927
std_time: 73.32320011058835
-----------------------------------------
```

The query above returns the avg and std values of the three different variables we're going to study (trip_time, passenger_count, trip_distance). In order to test our model we're going to calculate the z-score manually because we will have to adjust the acceptable range for the outliers. The query above will be a node of our pipe named *calculate_avg_std*.

As avg and std are aggregated values from the 3 variable series, probably we could use a CTE, subquery ~~or a function~~ (UDF not supported ClickHouse) to automate the calculus of the z-score for each trip. Of course querying directly the endpoint with a Python script would be the easiest solution.

UPDATE 18-05-2021: *After giving it some thoughts and reading the ClickHouse/Tinybird documentation, I think there is a possible solution to automate the z-score calculus, using the* `AggregatingMergeTree` *Engine and the* `AggregateFunction()`. *I'll give it a try. Still the possibility of the Python script is there*.

The next query is the node name *calculate_z_scores* using the data obtained in the *calculate_avg_std* node:

```
SELECT pickup_datetime,
        dropoff_datetime,
        dateDiff('minute', pickup_datetime, dropoff_datetime) AS trip_time,
        trip_time - 17.780161769434212 / 73.32320011058835 as z_time,
        passenger_count,
        passenger_count - 1.5756512724734397 / 1.2302451685284468 as z_passenger,
        trip_distance,
        trip_distance - 3.0367553475515594 / 3.8700316 as z_trip
FROM nyc_taxi_zone_clean
```

So if we want to use Python to query the data via [Tinybird's query API]() we only have to publish the node *calculate_z_scores* as an API endpoint and use the generated token like this:

```
import requests
import pandas as pd

# Token anonymized
params = {
    'token':
  'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
    'q':'SELECT pickup_datetime, dropoff_datetime, puzone, dozone, trip_time, passenger_count, trip_distance FROM calculate_z_scores
  LIMIT 1000'
}

url = 'https://api.tinybird.co/v0/pipes/nyc_taxi_zone_clean_pipe.json'

response = requests.get(url, params=params, stream=True)
stream = response.json()
print(stream.keys())
```

This is something I'll develop in another section when I have time.

Let's continue using Tinybird's UI.

Trip Time variable study

As we see in the avg and std values of the series, the average time for a trip between zones is 17 minutes with a deviation of 73 minutes, so we can start with the lower limit at 0 and the upper one at 120.

```
SELECT pickup_datetime,
      dropoff_datetime,
      puzone,
      dozone,
      trip_time,
      z_time NOT BETWEEN 0 AND 120 AS is_outlier
FROM calculate_z_scores
```

The results are promising: the lower limit is for trips of 0 minutes that can be considered as outliers. Probably 1 minute trips could not be considered outliers because they probably occur during rush hours and the passenger decided on his own to dropoff. Looking at the values we can see that there are 70+ trips to Airports and other zones so the upper bound could be 120 minutes. If we do that some strange trips appear but still we can raise the limit if we want.

Passenger count variable study

Now to start with a range of ±1 of the std :

```
SELECT pickup_datetime,
        dropoff_datetime,
        puzone,
        dozone,
        passenger_count,
        z_passenger NOT BETWEEN -1 AND 1 AS is_outlier
FROM calculate_z_scores
```

With this range we get outliers for 3+ more passengers and less than 1, something not very accurate. If we bump the upper limit to 3 we begin to get more accurate results taking into account that supposedly the yellow cabs cannot ride with more than 4 passengers. So finally the query that gives us the outliers between different zones for the variable passenger_count is:

```
SELECT pickup_datetime,
        dropoff_datetime,
        puzone,
        dozone,
        passenger_count,
        z_passenger
FROM calculate_z_scores
WHERE z_passenger NOT BETWEEN -1 AND 3.5
```

Trip distance variable study

We can get to the next and last variable of our problem that is the trip_distance.

As we can see the average of trip_distance is 3 miles with a deviation of 3.8 miles. That is where most of the distribution values are ranged. Again let’s start with the ±1 of the std:

```
SELECT pickup_datetime,
        dropoff_datetime,
        puzone,
        dozone,
        trip_distance,
        Z_trip NOT BETWEEN -1 AND 1 AS is_outlier
FROM calculate_z_scores
```

We get lots of outliers because the upper limit is very low. Taking into account that a trip from the upper east side to JFK or

Laguardia Airport are 15+ miles, we should not discard this trips because there are mny of them. So raising the upper limit to 7 or 8 should do the trick and we can have a pretty solid outlier list with this rage:

```
SELECT pickup_datetime,
      dropoff_datetime,
      puzone,
      dozone,
      trip_distance,
      z_trip
FROM calculate_z_scores
WHERE z_trip NOT BETWEEN -1 and 7
```

Improvements and added complexity: Data Stream

There is always room for improvements. For the next iteration in the process, I would like to automate the calculation of the standard deviation and average of the series for the trip time, trip distance and passenger count variables. One way of doing this would be as described above using the* `AggregatingMergeTree` *Engine and the* `AggregateFunction()`*. I'll give it a try.

Another way would be publishing and endpoint for the materialized view and query the data from a python script that I would happily write :-D when **I have more time (working on it)**. But still because of the simplicity of the model, we would need a human operator to calculate the lower and upper bounds of the distribution.

UPDATE 21-05-2021: *Another added problem/feature that I'll be adding is: How would you manage to solve the same problem but with the added complexity of data streaming: Instead of loading the CSV, just imagine that you have a continuos stream of data and you have to detect the outliers as the data is arriving*