Jobsity Challenge

This challenge was provided by Jobsity for the Data Engineering position testing candidates on Data Engineering and Software Engineering.

Problem Statement

Your task is to build an automatic process to ingest data on an on-demand basis. The data represents trips taken by different vehicles, and include a city, a point of origin and a destination.

This CSV file gives you a small sample of the data your solution will have to handle. We would like to have some visual reports of this data, but in order to do that, we need the following features.

We do not need a graphical interface. Your application should preferably be a REST API, or a console application.

Overview

This is a Python application written using Flask at backend and PySpark for data processing. It was developed for running locally however I will give instructions on how to setup it on cloud environments.

Directory Structure

```
src/
   migrations/  - migrations for creating tables and seed data
   models/      - the ORM models used for querying data
   pipelines/   - the ETL pipelines scripts written using PySpark
   utils/       - misc. functions
```

Pipelines

```
ingestion.py              - Used for ingest from the SQL database using JDBC driver
transformation.py         - Used for transforming ingested data in a ready-for-use
format
group_trips_by_route.py   - Uses a K-means ML model to group trips into routes
report_weekly_average.py  - Creates the data for the weekly average number of trips
                                  report and sends it to SQL database
```

Instructions

This is a python application that requires the following dependencies installed on your enviroment:

```
Spark and PySpark
MySQL database running locally
Flask
```

Other libraries can be installed using the requirements.txt.

Once you install all dependencies, please create a mysql database called **jobsity_db** with the following credentials:

```
user: jobsity_user
password: password
```

If needed, you can change these values on the yaml configurations files.

After that, please run the migrations scripts at **src/migrations** for a first charge and, only then, run the pipelines in this respective order:

```
ingestion -> transformation -> group_trips_by_route -> report_weekly_average
```

Mandatory Features

Use a SQL database.

The application uses a MySQL database instance running locally and also provides the yaml configuration file in order to deploy it on a kubernetes cluster. Also, the tables used are the following:

```
trips
    - id
    - region
    - origin_x,y
    - destination_x,y
    - datetime
    - datasource

bouding_boxes
    - id
    - x_min,y_min
    - x_max,y_max

report_weekly_average
    - id
    - region
    - route
    - bouding_box_id
    - weekly_average
```

Migrations

There are 2 python scripts that seed the tables so the application can start working with some sample data.

```
src/migrations/
    001_create_trips_table.py
    002_create_bouding_boxes_table.py
```

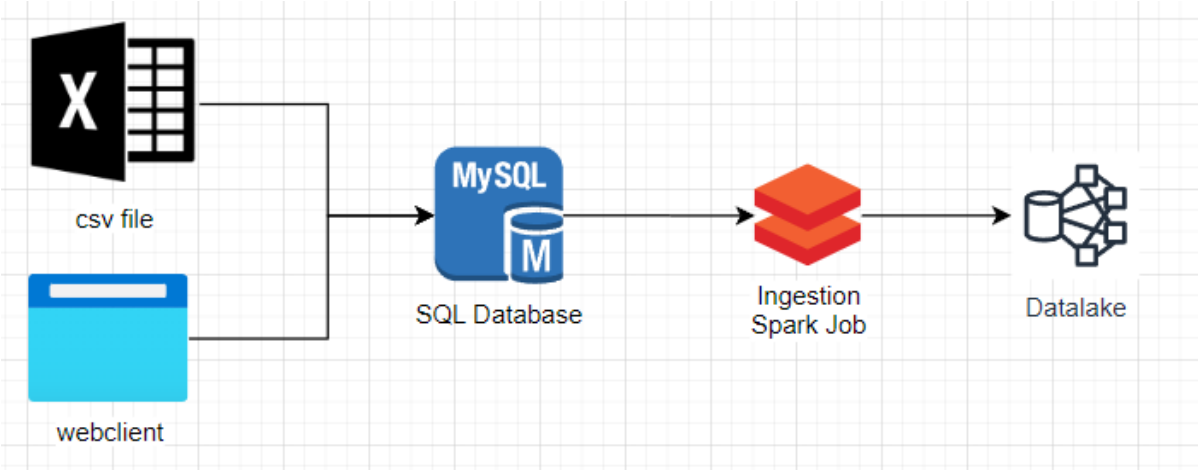
You should run them before running anything else.

There must be an automated process to ingest and store the data.

The automated process to ingest data is based upon pipelines in python scripts that could be orchestrated using something like airflow.

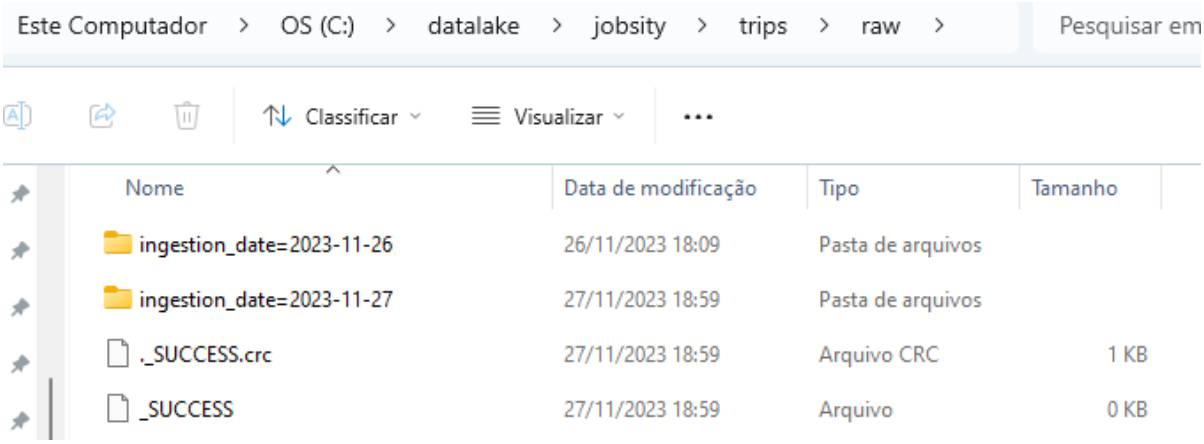
Ingestion

The ingestion is made by reading tables from a MySQL database using jdbc driver on Spark and saving it on a datalake by appeding data so if there are many runs of the pipeline none will be lost. Here is the architecture:



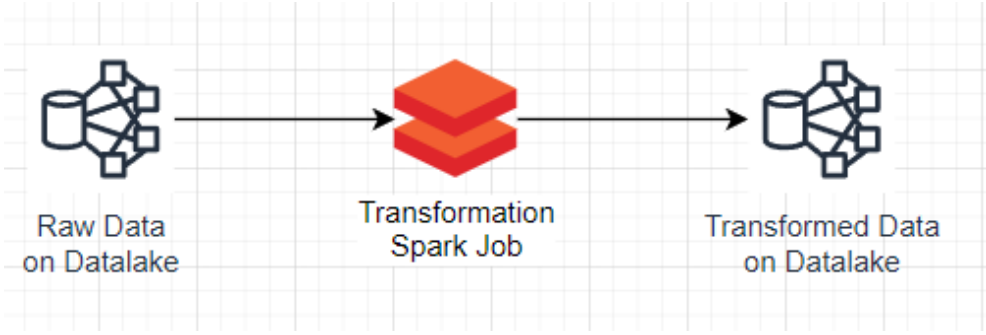
The SQL database was chosen as entry point because it could be easily accessible by any backend apllication or data job. Also, it could be scalable by replicating on the kubernetes cluster or easily choosing a managed solution from any cloud provider.

The data saved on the datalake is saved with a timestamp called **ingestion_timestamp** and **ingetion_date** so it can be used as partitions in order to improve the further processing.

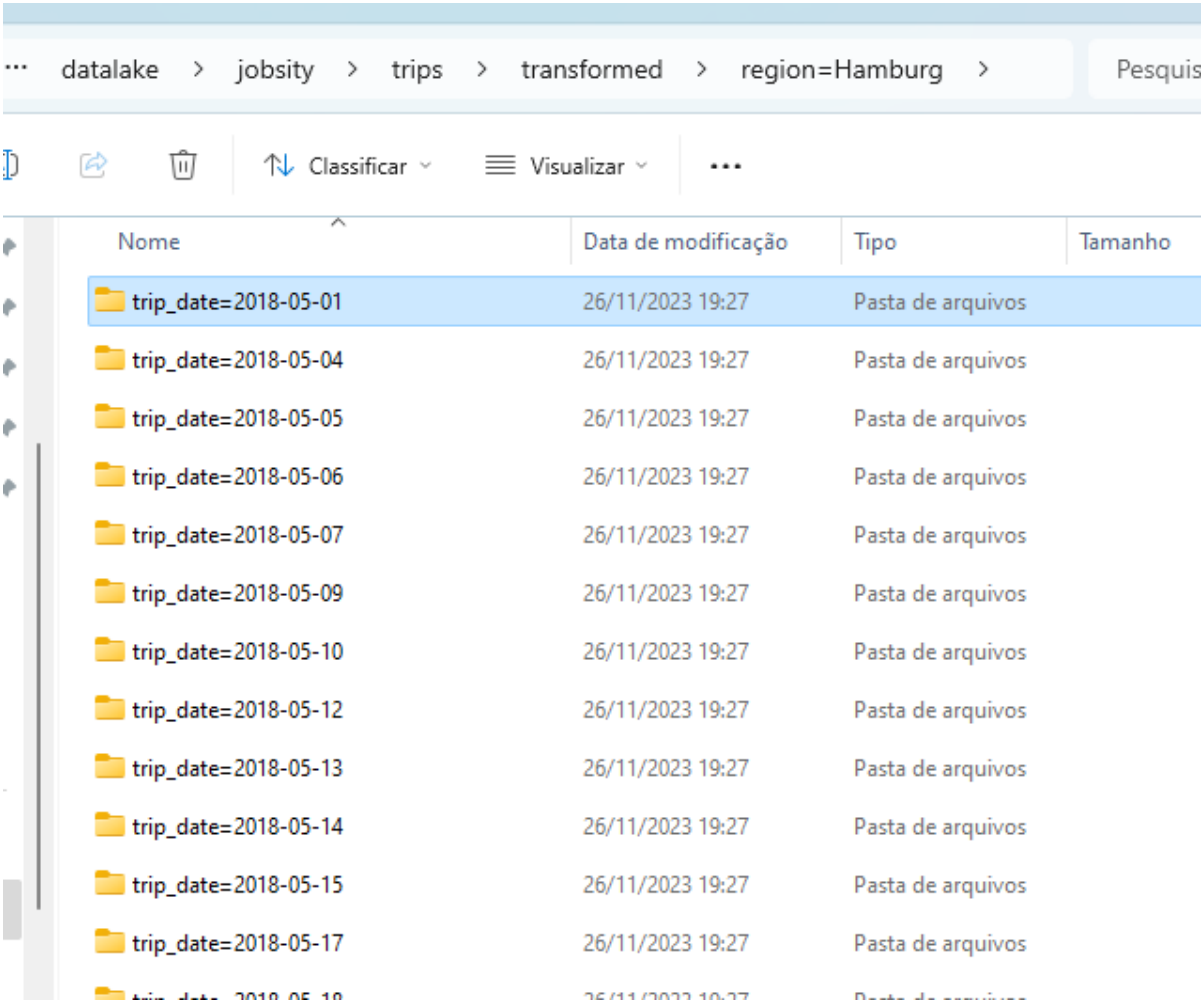


Transformation

The transformation occurs entirely on the cluster side by running a spark job after the ingestion in order to read the entire raw ingested data and save it de-duplicated and properly partitioned for improving performance on a another location at the datalake so this way only the latest data will be available for the following processes.

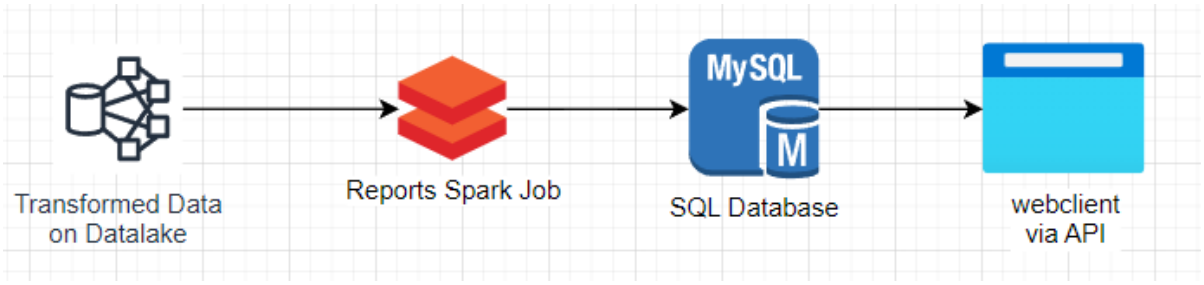


The data is saved partitioned so trips with same regions and date are saved together to improve future uses and queries.



Reports and ML Models

The ML K-means model that cluster similar trips as routes is run as a Spark Job and saves it on the datalake as datasets. Also, the report used to answer the question proposed about week



Trips with similar origin, destination, and time of day should be grouped together.

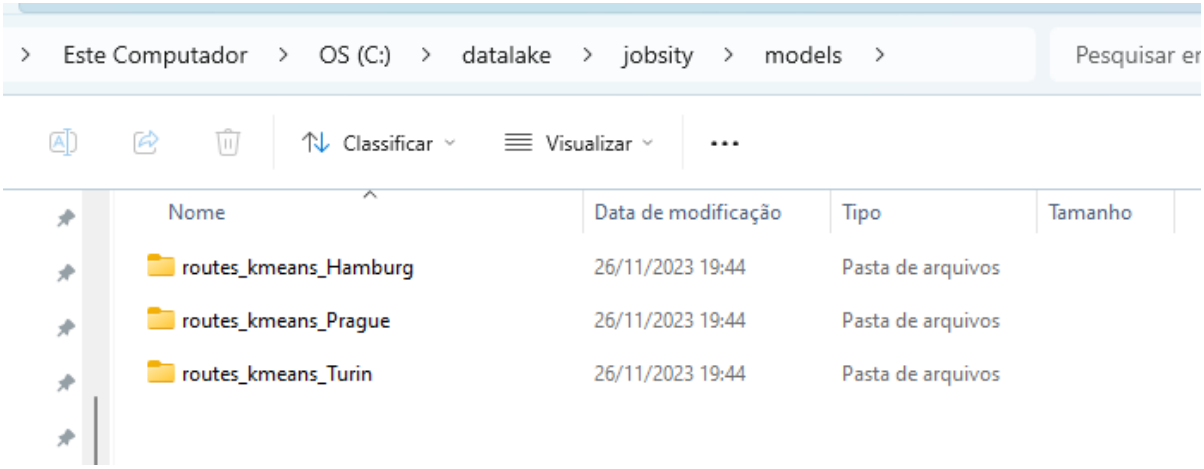
In order to group trips with similar origin, destination and time of day, I used a ML Model called K-means which given a number of clusters (k = 5) groups the data points by similarity. In addition, in order to improve the accuracy of the model predictions, I created one for each region because of the nature of the data which is location specific. This way we can analyze the statistics from the similar locations in each city and use this insights to improve the service.

As we can see on the image below, the trips are clustered by origin and destination coordinates which are passed on to the algorithm achieving the following predictions.

The model is created on the pipeline script and saved on disk at the cluster location like the following:

```
for region in regions:
    df_region = df.filter(col("region") == region)

    assembler = VectorAssembler(
        inputCols=["origin_x", "origin_y", "destination_x", "destination_y"],
        outputCol="features"
    )

    kmeans = KMeans().setK(5).setSeed(42)

    pipeline = Pipeline(stages=[assembler, kmeans])

    model = pipeline.fit(df_region)

    model_path = f"{DATALAKE_PATH}/jobsity/models/routes_kmeans_{region}"

    save_model(model, model_path)

    predictions = model.transform(df_region)
```

As you can see, the models are saved to disk so they can be used later without having to training them again.

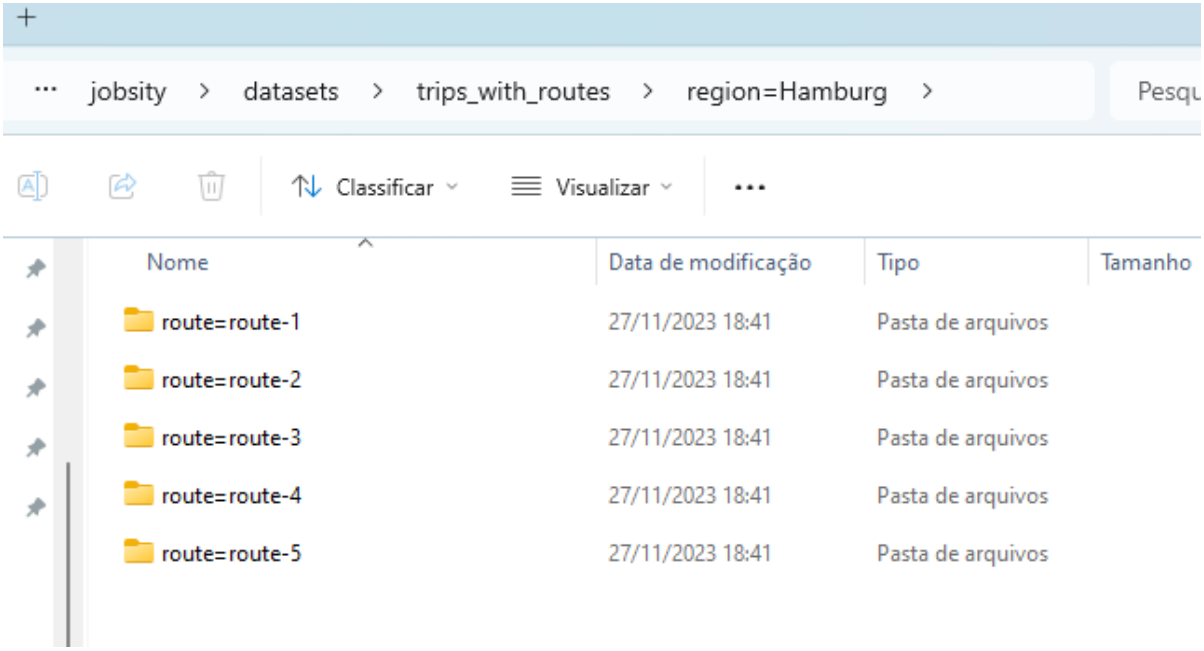


The resulting data should look like this:

id	origin_x	origin_y	destination_x	destination_y	datetime	datasource	ingestion_timestamp	trip_week	region	trip_date	features	prediction	route
29	7.69123	45.0691	7.52636	44.9817	2018-05-26 04:25:56	cheap_mobile	2023-11-26 18:09:...	21	Turin	2018-05-26	[7.69123,45.0691,...	0	route-1
38	7.62504	44.9818	7.73895	45.0373	2018-05-26 12:14:42	bad_diesel_vehicles	2023-11-26 18:09:...	21	Turin	2018-05-26	[7.62504,44.9818,...	3	route-4
46	7.73966	45.101	7.59712	45.1162	2018-05-26 05:15:45	cheap_mobile	2023-11-26 18:09:...	21	Turin	2018-05-26	[7.73966,45.101,7...	1	route-2
63	7.73283	45.0975	7.66251	45.0647	2018-05-26 06:04:17	bad_diesel_vehicles	2023-11-26 18:09:...	21	Turin	2018-05-26	[7.73283,45.0975,...	1	route-2
25	7.60311	45.0575	7.63958	45.1002	2018-05-13 12:00:21	funny_car	2023-11-26 18:09:...	19	Turin	2018-05-13	[7.60311,45.0575,...	2	route-3
57	7.56348	44.98	7.51713	45.0648	2018-05-13 22:16:27	pt_search_app	2023-11-26 18:09:...	19	Turin	2018-05-13	[7.56348,44.98,7,...	4	route-5
79	7.51313	45.0442	7.64359	45.0213	2018-05-13 15:37:37	bad_diesel_vehicles	2023-11-26 18:09:...	19	Turin	2018-05-13	[7.51313,45.0442,...	2	route-3
2	7.67284	44.9957	7.72037	45.0678	2018-05-21 02:54:04	baba_car	2023-11-26 18:09:...	21	Turin	2018-05-21	[7.67284,44.9957,...	1	route-2
75	7.53574	45.0923	7.57971	45.1043	2018-05-21 15:36:26	cheap_mobile	2023-11-26 18:09:...	21	Turin	2018-05-21	[7.53574,45.0923,...	2	route-3
87	7.67669	45.0625	7.68696	45.0971	2018-05-21 21:32:17	baba_car	2023-11-26 18:09:...	21	Turin	2018-05-21	[7.67669,45.0625,...	1	route-2
59	7.59273	45.11	7.71493	45.0773	2018-05-18 19:29:56	bad_diesel_vehicles	2023-11-26 18:09:...	20	Turin	2018-05-18	[7.59273,45.11,7,...	3	route-4
19	7.70322	45.1204	7.71323	45.0873	2018-05-29 15:58:39	cheap_mobile	2023-11-26 18:09:...	22	Turin	2018-05-29	[7.70322,45.1204,...	1	route-2
32	7.6859	45.0757	7.7156	45.0547	2018-05-29 13:42:34	bad_diesel_vehicles	2023-11-26 18:09:...	22	Turin	2018-05-29	[7.6859,45.0757,7...	1	route-2
4	7.54151	45.0916	7.74529	45.0263	2018-05-06 09:49:16	bad_diesel_vehicles	2023-11-26 18:09:...	18	Turin	2018-05-06	[7.54151,45.0916,...	3	route-4
8	7.56079	45.019	7.58357	45.1053	2018-05-06 00:00:44	cheap_mobile	2023-11-26 18:09:...	18	Turin	2018-05-06	[7.56079,45.019,7...	4	route-5
34	7.66261	45.0944	7.72429	45.0738	2018-05-08 15:11:01	bad_diesel_vehicles	2023-11-26 18:09:...	19	Turin	2018-05-08	[7.66261,45.0944,...	1	route-2
69	7.57917	45.0706	7.61348	45.0609	2018-05-08 04:29:25	funny_car	2023-11-26 18:09:...	19	Turin	2018-05-08	[7.57917,45.0706,...	2	route-3
72	7.72038	45.1044	7.69505	45.1283	2018-05-05 07:05:01	baba_car	2023-11-26 18:09:...	18	Turin	2018-05-05	[7.72038,45.1044,...	1	route-2
88	7.54639	45.0875	7.55594	44.9975	2018-05-05 06:08:53	bad_diesel_vehicles	2023-11-26 18:09:...	18	Turin	2018-05-05	[7.54639,45.0875,...	2	route-3
64	7.68875	44.9889	7.76174	44.9867	2018-05-27 09:49:16	cheap_mobile	2023-11-26 18:09:...	21	Turin	2018-05-27	[7.68875,44.9889,...	1	route-2

only showing top 20 rows

The data is then saved partitioned on the dataset at the datalake by region, route and date.



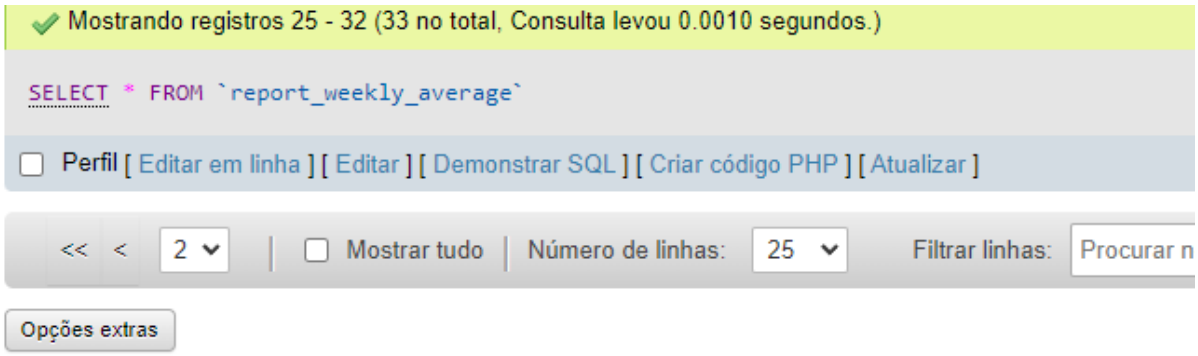
Develop a way to obtain the weekly average number of trips for an area, defined by a bounding box (given by coordinates) or by a region.

Report Weekly Average

In order to further automate the process of obtaining this insight I created an Spark Job that performs all the calculations necessary and save it into a SQL database that then can be access by backend applications and analytics users. Also, the bouding boxes with the coordinates are on a SQL table called **bouding_boxes** and contain coordinates that will be used to filter the trips occurred inside them and calculate the weekly average.

The pipeline is called **report_weekly_average** and it will calculate 3 metrics:

```
region_weekly_average       - the weekly average of each route
region_route_weekly_average - the weekly average by route (from the ML model)
bouding_box_weekly_average  - the weekly average in a bouding box defined by the
user
```

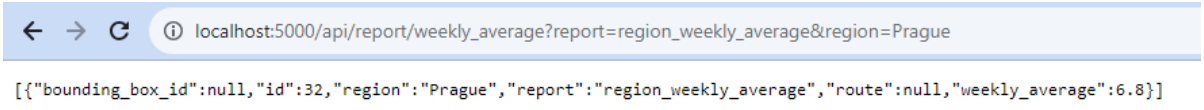


report	region	weekly_average	route	bounding_box_id	id
region_route_weekly_average	Turin	1.3333333333333333	route-1	NULL	26
region_route_weekly_average	Turin	2.8	route-2	NULL	27
region_route_weekly_average	Turin	2	route-3	NULL	28
region_route_weekly_average	Turin	2	route-4	NULL	29
region_route_weekly_average	Turin	1.3333333333333333	route-5	NULL	30
region_weekly_average	Hamburg	5.6	NULL	NULL	31
region_weekly_average	Prague	6.8	NULL	NULL	32
region_weekly_average	Turin	7.6	NULL	NULL	33

API

The data can be accessed by an API served by the Flask application with the following route:

```
GET /api/report/weekly_average
    param report  - ex: region_weekly_average
    param region  - ex: Prague
    param route   - ex: route-1
    param bbox_id - ex: 1
```



[{"bounding_box_id":null,"id":32,"region":"Prague","report":"region_weekly_average","route":null,"weekly_average":6.8}]

Develop a way to inform the user about the status of the data ingestion without using a polling solution.

In progress.

The solution should be scalable to 100 million entries. It is encouraged to simplify the data by a data model. Please add proof that the solution is scalable.

Since the SQL database is only used for receiving the data from client facing applications it could be periodically clean so it does not become a bottleneck and because the following data processing occurs in Spark jobs independently it could leverage the compute power to process large amounts of data. Also, the reports and metrics are all calculated on the data side and only served to the backend via SQL tables which are used for displaying via APIs and dashboards.

So, we have a very scalable solution that is able to handle to 100 million entries.

Bonus Features

Containerize your solution.

Both the application and the MySQL databases had YAML files created for deploying it on kubernets cluster. But first you will need to build the docker image file using:

```
docker build . -t jobsity-app
```

Sketch up how you would set up the application using any cloud provider (AWS, Google Cloud, etc).

AWS

The pipelines could be easily turned into Spark EMR jobs by saving the files into S3 and setup the jobs to execute them. Also, the S3 could be used as an datalake and we could use either the SQL managed solution or an EC2 instance with it, being the first option preferreble. The Python application since its a simple Flask one could be easily deployed into Elastic Beastalk.

Azure

The pipelines could be managed by the ADF running them on DataBricks notebooks connected to standalone clusters or spot clusters for low cost. Also, the SQL database could be migrated to its Database Service for managed solution and the data lake use the Blob Storage connected to the DataBricks for access control.

Include a .sql file with queries to answer these questions:

Both were run at the MySQL database and they are under the sql_query folder.

From the two most commonly appearing regions, which is the latest datasource?

```
WITH top_regions AS (
    SELECT
        region,
        COUNT(*) as occurrences
    FROM trips
    GROUP BY region
    ORDER BY occurrences DESC
    LIMIT 2
)
SELECT
    t.region,
    t.datasource,
    top_regions.occurrences
FROM trips t
INNER JOIN top_regions ON t.region = top_regions.region
WHERE t.datetime = ( SELECT MAX(datetime) FROM trips WHERE region = t.region );
```

region	datasource	occurrences
Turin	pt_search_app	38
Prague	cheap_mobile	34

What regions has the "cheap_mobile" datasource appeared in?

```sql
SELECT
    region,
    COUNT(*) occurrences
FROM trips
WHERE datasource = 'cheap_mobile'
GROUP BY region
ORDER BY occurrences DESC;
```

region	occurrences ▾ 1
Prague	13
Hamburg	10
Turin	10