Interview Assessment Task

Offerista Group Bulgaria

OpenWeather API

Dimitar Petkov

March 21, 2024

Table of Contents

Chapter 1: **Requirements**

Assessment test: Data Engineering

This test is intended to verify basic skills in Data Engineering (data extraction, modelling, critical reasoning) for candidates to the role of Data Engineer in ShopFully.

The deadline for returning the output is specified to the candidate when sharing the test. The results will then be used as a topic for discussion in the subsequent interview.

The candidate is left free to decide the preferred tools with which to carry out the test, it is sufficient that all the results and the coding are then shared through a text document for further discussion.

Project description

For the ShopFully team it is very important to understand the weather conditions for certain locations.

Use the OpenWeatherMap API (use a free tier subscription) to get the current weather conditions for 3 cities where ShopFully has offices: *Milano, Bologna, Cagliari.*

The task has two main parts, one focuses on the modelling while the other one is centered on script writing.

Part 1 - data modelling

1. Look at the data structure provided by the API documentation.
2. Decide which data could be considered important and bring value and discard the data which looks less relevant.
3. The data granularity should be 1-hour (we want to have hourly temperature to be able to analyse historical data in the future).
4. Create a logical and physical model for this data having the following questions in mind:
    - 4.1 How many distinct weather conditions were observed (rain/snow/clear/…) in a certain period?
    - 4.2 Rank the most common weather conditions in a certain period of time per city?
    - 4.3 What are the temperature averages observed in a certain period per city?
    - 4.4 What city had the highest absolute temperature in a certain period of time?
    - 4.5 Which city had the highest daily temperature variation in a certain period of time?
    - 4.6 What city had the strongest wing in a certain period of time?
5. Deliverable:
    - 5.1 Visualized logical schema.
    - 5.2 Complete DDL for physical database implementation.

5.3 SQL queries.

Part 2 - script writing

6. Automate the data download process.
7. Store the raw (response) data in the format you find the most suitable.
8. Identify the information you find useful and create a dataframe with it.
9. Write the data into the table(s) you identified in the modelling process.
10. Write the data to a relational database. You have the freedom to decide how to organize your data/relationships, data types, primary, foreign keys, indices, etc.
11. Could you answer the questions from the previous section using aggregations in python applied on the denormalized dataframe?

Note

12. If you are not able to provide a script, you can concentrate on the modelling part of the data by looking at the expected API output/response in the documentation.
13. Organize the project taking into consideration that data needs could grow and involve:

    13.1 All Italian municipalities and beyond.

    13.2 Different ways to get the information and consequently different API calls (always the same endpoint)

Chapter 2: **Chosen tech stack**

After analysing the project requirements, the chosen technologies to complete the task at hand are:

- Coding language: Python, SQL

- Azure Databricks: for ingesting and transforming the data.

- Azure PostgreSQL database though PgAdmin4: for storing data, both raw and processed.

- Azure Key Vault: for storing Azure keys and passwords.

Chapter 3: **API information**

The API can be found HERE. The chosen tier for the task is the Current Weather Data Free tier. It offers access current weather data for any location, format in JSON, XML, and HTML. With this free data we can set-up an hourly schedule get data from the API and house a database with the hourly forecast.

By providing the API with city names (*Milano, Bologna, Cagliari*), preferred units (metric) and default language as English we can get the following information:

1. coord
    - 1.1 coord.lon - Longitude of the location
    - 1.2 coord.lat - Latitude of the location
2. weather
    - 2.1 weather.id - Weather condition id
    - 2.2 weather.main - Group of weather parameters (Rain, Snow, Clouds etc.)
    - 2.3 weather.description - Weather condition within the group. Please find more here. You can get the output in your language. Learn more
    - 2.4 weather.icon - Weather icon id
3. base Internal parameter
4. main
    - 4.1 main.temp - Temperature. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit
    - 4.2 main.feels_like - Temperature. This temperature parameter accounts for the human perception of weather. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit
    - 4.3 main.pressure - Atmospheric pressure on the sea level, hPa
    - 4.4 main.humidity - Humidity, %
    - 4.5 main.temp_min - Minimum temperature at the moment. This is minimal currently observed temperature (within large megalopolises and urban areas).
    - 4.6 main.temp_max - Maximum temperature at the moment. This is maximal currently observed temperature (within large megalopolises and urban areas).
5. visibility - Visibility, meter. The maximum value of the visibility is 10 km
6. wind
    - 6.1 wind.speed - Wind speed. Unit Default: meter/sec, Metric: meter/sec, Imperial: miles/hour
    - 6.2 wind.deg - Wind direction, degrees (meteorological)

7. clouds
    7.1 clouds.all - Cloudiness, %
8. dt - Time of data calculation, unix, UTC
9. sys
    9.1 sys.type - Internal parameter
    9.2 sys.id - Internal parameter
    9.3 sys.country - Country code (GB, JP etc.)
    9.4 sys.sunrise - Sunrise time, unix, UTC
    9.5 sys.sunset - Sunset time, unix, UTC
    9.6 timezone - Shift in seconds from UTC
10. id - City ID. Please note that built-in geocoder functionality has been deprecated. Learn more here
11. name - City name. Please note that built-in geocoder functionality has been deprecated. Learn more here
12. cod - Internal parameter

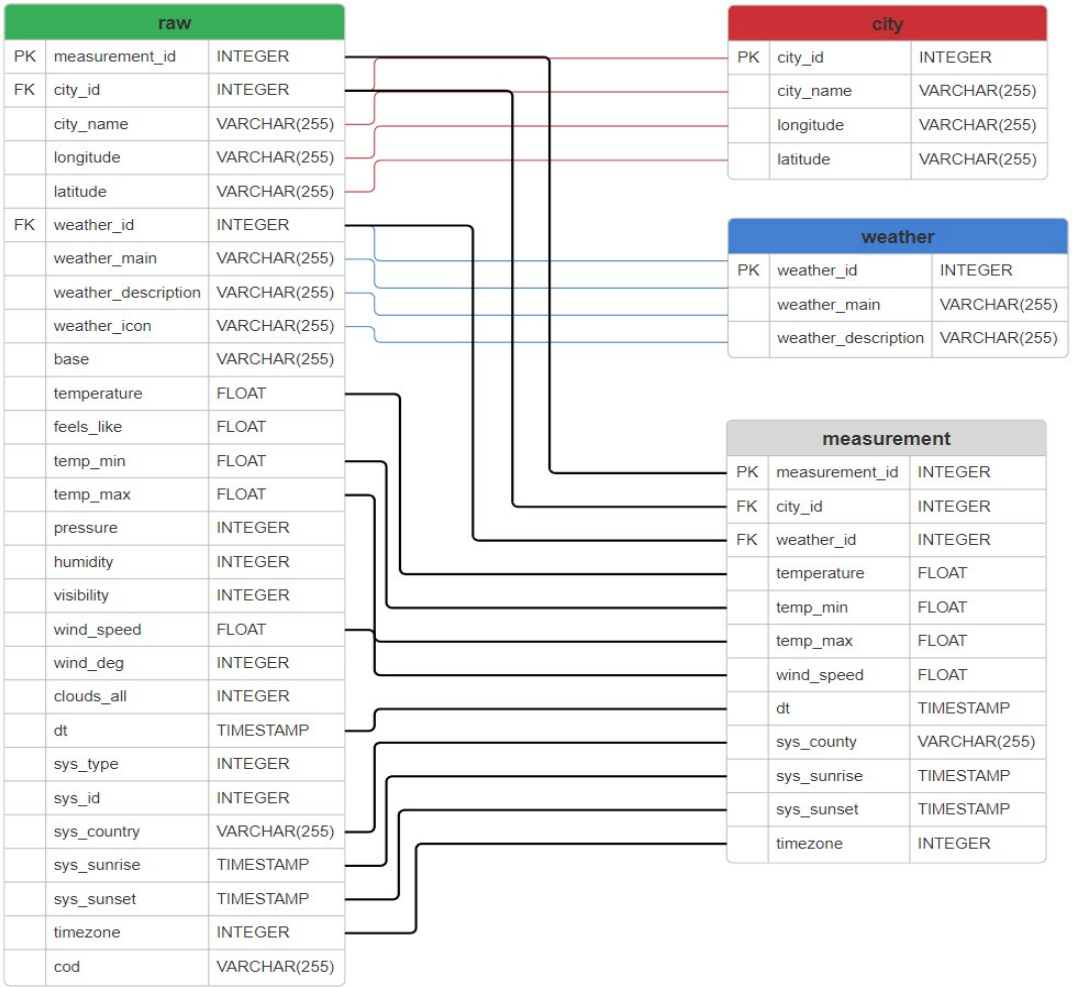
Chapter 4: **Data model**

I have design the data model with first having a table with all of the raw data from the API. We save all of the data for future requirements. Even thou columns like longitude, latitude, weather_description, weather_icon, base, feels_like, pressure, humidity, visibility, wind_deg, clouds_all, sys_type, sys_id, timezone and cod. Then we create the table measurement which will store all the needed data for our needs with the "timezone" column not being utilized but only for reference information. The "sys_country" represents a country code, while it is not needed it will be useful if we where to query information for the whole country for point 13.1 of Chapter 1.

Then we have a one-to-many relationship between tables:

- city > measurement
- weather > measurement

Because we have multiple measurements per city and weather types. With each city and weather information being stored in their own tables and linked to the measurements table via id keys. This layout is designed as such with the idea that every hour data will be appended into the "raw" table. Then before querying or at a certain time period i.e., once per day or once per week the process notebooks can be ran and the data will be populated in the other tables for query execution.



Chapter 5: **Data flow**

The code for the notebooks is stored in GitHub via LINK.

With Project layout being:

1. Ingestion – here we set-up connection and store the raw data.

    1.1 Notebook "1. data_ingestion_from_api.py" it sets up the API keys, PostgreSQL database connection, it fetches the data from the API and formats it according to the DB schema and then writes it to the DB "raw" table. For practical uses some of the data/columns are not needed for the final goal but are stored for testing purposes. Data/Columns like longitude, latitude, weather_description, weather_icon, base, feels_like, pressure, humidity, visibility, wind_deg, clouds_all, sys_type, sys_id, timezone and cod.

2. Processed – here we distribute the relative data to the final tables that we will query.

    2.1 Notebook "1. processing_raw_data_to_populate_city_table.py" here we take any new cities and populate their data into the "city" table.

    2.2 Notebook "2. processing_raw_data_to_populate_weather_table.py" here we take any new weather types and populate their data into the "weather" table.

    2.3 Notebook "3. processing_raw_data_to_populate_measurement_table.py" here we take any new measurements and populate their data into the "measurement" table.

3. Presentation – here we have the one of the deliverables the final queries.

    3.1 Notebook "1. required_queries_for_task.py" contains all that queries for execution to achieve the results defined in point 2. Of the Requirements

4. Data_save_to_dbfs – here we have the notebook that writes the data from the PostgreSQL database to csv in DBFS.

    4.1 Notebook "1. data_download_as_csv_to_dbfs.py" writes the data to csv as mentioned above.

Chapter 6: **Deliverables**

The deliverables are as described in point 3 of the Requirements. DDL's and queries will be provided in .txt document in the GitHub repo:

1. Visualized logical schema – can be found in Chapter 4

2. Complete DDL's for physical database implementation:

   2.1 Table "raw"

```
CREATE TABLE raw (
    measurement_id SERIAL PRIMARY KEY,
    city_id INTEGER,
    city_name VARCHAR(255),
    longitude VARCHAR(255),
    latitude VARCHAR(255),
    weather_id INTEGER,
    weather_main VARCHAR(255),
    weather_description VARCHAR(255),
    weather_icon VARCHAR(255),
    base VARCHAR(255),
    temperature FLOAT,
    feels_like FLOAT,
    temp_min FLOAT,
    temp_max FLOAT,
    pressure INTEGER,
    humidity INTEGER,
    visibility INTEGER,
    wind_speed FLOAT,
    wind_deg INTEGER,
    clouds_all INTEGER,
    dt TIMEZONE,
    sys_type INTEGER,
    sys_id INTEGER,
    sys_country VARCHAR(255),
    sys_sunrise TIMEZONE,
    sys_sunset TIMEZONE,
    timezone INTEGER,
    cod VARCHAR(255)
);
```

   2.2 Table "city"

```
CREATE TABLE city (
    city_id INTEGER PRIMARY KEY,
    city_name VARCHAR(255),
    longitude VARCHAR(255),
    latitude VARCHAR(255)
);
```

   2.3 Table "weather"

```
CREATE TABLE weather (
    weather_id INTEGER PRIMARY KEY,
    weather_main VARCHAR(255),
    weather_description VARCHAR(255)
);
```

2.4 Table "measurement"

```
CREATE TABLE measurement (
    measurement_id INTEGER PRIMARY KEY,
    city_id INTEGER,
    weather_id INTEGER,
    temperature FLOAT,
    temp_min FLOAT,
    temp_max FLOAT,
    wind_speed FLOAT,
    dt TIMEZONE,
    sys_country VARCHAR(255),
    sys_sunrise TIMEZONE,
    sys_sunset TIMEZONE,
    timezone INTEGER,
    FOREIGN KEY (city_id) REFERENCES city(city_id),
    FOREIGN KEY (weather_id) REFERENCES weather(weather_id)
);
```

3. SQL queries for the results from point 2 of the Requirements. These queries can also be viewed in folder/notebook *3.presentation/ 1. required_queries_for_task.py*

3.1 Distinct values of conditions (rain/snow/clear/…) for a given period.

We select first the only the measurements in the date period to narrow down the join operation. Then we join and do a distinct clause.

```
SELECT DISTINCT w.weather_main
        FROM (
            SELECT *
            FROM measurement
            WHERE dt BETWEEN '2024-03-18 10:10:10' AND '2024-03-26 10:10:10'
            ) m
        JOIN weather w ON m.weather_id = w.weather_id
        WHERE m.dt BETWEEN '2024-03-18 10:10:10' AND '2024-03-26 10:10:10';
```

3.2 Most common weather conditions in a certain period of time per city.

Here I have two solutions. First one is the answer to the task and the second being the most common weather condition overall.

3.2.1

```
SELECT m.city_id, c.city_name, w.weather_main, COUNT(*) AS frequency
        FROM (
            SELECT *
            FROM measurement
            WHERE dt BETWEEN '2024-03-18 10:10:10' AND '2024-03-26
          10:10:10'
          ) m
        JOIN city c ON m.city_id = c.city_id
        JOIN weather w ON m.weather_id = w.weather_id
        GROUP BY m.city_id, c.city_name, w.weather_main
```

```
ORDER BY frequency DESC;
```

3.2.2

```
SELECT m.city_id, c.city_name, w.weather_main, COUNT(*) AS frequency
FROM (
        SELECT *
        FROM measurement
        WHERE dt BETWEEN '2024-03-18 10:10:10' AND '2024-03-26 10:10:10') m
JOIN city c ON m.city_id = c.city_id
JOIN weather w ON m.weather_id = w.weather_id
GROUP BY m.city_id, c.city_name, w.weather_main
ORDER BY frequency DESC
LIMIT 1;
```

3.3 Temperature averages observed in a certain period per city.

Again, we select first the only the measurements in the date period to narrow down the join operation. We also round the float of the average to the 2nd decimal point for better visualizing the result. We achieve this with CAST AS DECIMAL.

```
SELECT m.city_id, c.city_name, CAST(AVG(m.temperature) AS
DECIMAL(10,2)) AS avg_temperature
FROM (
        SELECT *
        FROM measurement
        WHERE dt BETWEEN '2024-03-18 10:10:10' AND '2024-03-26
       10:10:10'
        ) m
JOIN city c ON m.city_id = c.city_id
GROUP BY m.city_id, c.city_name;
```

3.4 City that had the highest absolute temperature in a certain period of time.

Here we use the same logic as 3.3 but we also order in desc and limit to 1 to get the highest value.

```
SELECT m.city_id, c.city_name, CAST(MAX(m.temp_max) AS
DECIMAL(10,2)) AS max_temperature
FROM (
        SELECT *
        FROM measurement
        WHERE dt BETWEEN '2024-03-18 10:10:10' AND '2024-03-26
       10:10:10'
       ) m
JOIN city c ON m.city_id = c.city_id
GROUP BY m.city_id, c.city_name
ORDER BY max_temperature DESC
LIMIT 1;
```

3.5 City that had the highest daily temperature variation in a certain period of time.

First, we create a CTE to get the only the data in the time period and during the day so the time is sys_sunrise < = dt =< sys_sunset. Then we find the max and min. And after we order and limit to 1 to get the biggest temperature variation.

```
WITH daily_variation_cte AS (
  SELECT
        city_id,
        dt::date AS measurement_date,
        MAX(temp_max) - MIN(temp_min) AS daily_variation
  FROM measurement
  WHERE
        dt::time >= sys_sunrise::time
        AND dt::time <= sys_sunset::time
        AND dt BETWEEN '2024-03-18 10:10:10' AND '2024-03-26 10:10:10'
  GROUP BY
        city_id,
        dt::date
)
SELECT
        dvc.city_id,
        c.city_name,
        dvc.daily_variation
FROM daily_variation_cte dvc
JOIN city c ON dvc.city_id = c.city_id
ORDER BY dvc.daily_variation DESC
LIMIT 1;
```

3.6 City that had the strongest wing in a certain period of time.

We use the same approach for the float as 3.3 and we use the same approach for the filtering and ordering as 3.5.

```
SELECT m.measurement_id, m.city_id, c.city_name,
CAST(MAX(m.wind_speed) AS DECIMAL(10,2)) AS max_wind_speed
FROM (
        SELECT *
        FROM measurement
        WHERE dt BETWEEN '2024-03-18 10:10:10' AND '2024-03-26 10:10:10'
        ) m
 JOIN city c ON m.city_id = c.city_id
 GROUP BY m.measurement_id, m.city_id, c.city_name
 ORDER BY max_wind_speed DESC
 LIMIT 1;
```