**GitHub: https://github.com/shailja-somani-0/ADS-507-Team-4**

**System Architecture: San Diego, California Weather Data and Alerts Pipeline**

Shailja Somani, Tara Dehdari, and John Vincent Deniega

Shiley-Marcos School of Engineering, University of San Diego

ADS 507: Practical Data Engineering

Professors Jonathan Sixt and Dillon Orr

February 24, 2024

**Table of Contents**

**Introduction**

Meteorology and weather – more broadly – can be observed in real-time or over specific periods of interest. In the case of significant weather events, these observations may prove vital to understanding the breadth of data applications ranging from understanding climate change to public safety (“Climate Models,” 2023). For example, in February 2024, California experienced powerful storms that caused major flooding, mudslides, and catastrophic damage (Bravo, 2024). As such, it is incumbent on domains including data science to utilize multiple sources of data, whether from data streaming sources such as thermometers and barometers or point observations recorded at hourly or daily intervals and to engineer data pipelines that provide useable data for end-users to analyze and make critical decisions. Well-established sources such the National Weather Service [NWS], the University of California San Diego [UCSD], and OpenWeather together can inform the casual user and the data scientist with historical data analysis and weather and climate projections. So long as the data is thoughtfully treated and transformed to be compatible with each other and so long as they are standardized into a common schema, proper data engineering becomes the necessary conduit between these various weather data sources and the climate scientists and public safety officials seeking to find more effective solutions.

**Methods**

**Initial Configuration**

First, the ‘pymysql’ Python package allowed for an object to be created to take in network socket information as arguments to be passed to a MySQL server. This object acts as the interface in which embedded MySQL commands may be passed over Python such as selecting a database, passing MySQL syntax operations, or triggering execution of specific commands to the server. For this project, we connected to the local MySQL server on the machine of whomever executes the Python code. The server is directed to create a database called “WeatherDatabase” if it does not already exist, then create three tables if they do not exist: “DailyWeatherConditions,” “DailyWeather Alerts,” and “HistoricalWeatherData.” The day, month, and year columns for each table are integer values that cannot contain null values as the combination of these three fields will serve as the primary key for each table. However, we elected to keep the three columns as individual integers rather than only having a date primary key for easier joining across the two current daily tables and the historical table. By being able to join on just day and month across multiple years, that facilitates comparison across historical data. Throughout the tables, generally, any lists of geographic areas are stored as strings of VARCHAR of length 5000 as these lengths may vary depending on the quantity of alerts upon API call. Headline information is stored as a string of VARCHAR of length 150 as a reasonable length to capture the topic of the descriptive information. Description of the weather alert is stored as strings of TEXT as this narrative information contains verbose and extended data pertaining to the weather alert. Values such as humidity are stored as integer values as decimal values are not observed.

To the end of creating a singular view that contains daily (current day) data, as well as historical comparisons, the server is then directed to create a view called “DailyWeatherView.” This view combines the daily information from the “DailyWeatherConditions” and “DailyWeather Alerts” tables by joining on the entirety of the date (day, month, and year). Then, the daily data is joined to metrics from five and ten years ago, by joining to the “HistoricalWeatherData” table on day and month, as well as year minus five and year minus 10 (two joins are done in succession). The resulting view contains one row per day of data in the current year with columns for that day’s weather conditions and alerts, as well as historical weather conditions for five and ten years prior to that date. This view is very helpful for a daily snapshot tying together all three data sources, as well as to feed into visualizations, which we discuss further in the Results section.

With the MySQL database configured, data was extracted from three different sources. The ‘nwsapy’ Python package allowed interfacing with the official NWS API via Python language and associated methods and attributes. A user agent method established a connection to the NWS server by passing API and identification information as arguments to the user agent object (“API Web Service,” n.d.). Once the connection is established, the API is called to extract weather alerts for a given location. The “get\_alert\_by\_zone” method was called on the user agent object and was passed a six-character argument. This argument consisted of three A through Z characters followed by three zero through nine characters as a string value corresponding with the NWS’s identified geographic zones. An “AlertByZone” object is returned, which may be printed to standard output as either a dictionary or a Pandas DataFrame using the returned object’s respective methods.

However, not all geographic areas may have an active alert at the time the API is called, so the object that is returned to the user may contain a null value. In order to avoid inserting malformed null data into the MySQL database, this object is checked if the value is null, which if it is will set values in accordance with the target schema containing the primary keys used across all three sources: Date, Month, and Year. For this and the remaining values in accordance with the schema, null values and a headline of “No Alert Today” are staged in a string variable, which will be passed to the MySQL connection object for execution as an “INSERT INTO” statement.

The pipeline takes in a second data source, the UCSD historical weather dataset, in the form of a static comma-separated value [CSV] file. Data from this source contains daily observations of weather data from the period year 1939 to 2019, which was originally collected by the National Oceanic and Atmospheric Administration (“San Diego Weather,” n.d). Temperature data was transformed into the same units per the schema and dates were parsed into their respective day, month, and year values for primary key purposes. Columns that were valid, but did not have significant overlap or contextual importance with the other data sources were dropped, primarily consisting of temperature and wind speed flag information, of which the data was sparse in density. The transformed data was parsed into their respective schema variables and staged into a string variable containing the MySQL statement to be executed as an “INSERT INTO” statement.

The pipeline takes in the final data source, the OpenWeather API, which utilizes the ‘requests’ Python package to extract data via a uniform resource locator with an embedded latitude and longitude for San Diego, California as well as an assigned application identifier (OpenWeather API, n.d.). Similarly to the two sources above, data only from the current day is extracted, which is returned in a dictionary object. Data is identified in accordance with the target schema and staged into a string variable that is to be executed as an “INSERT INTO” statement, much like the previous sources.

**Daily API Calls**

The initial configuration involved building the “WeatherDatabase” with three tables and one view, as well as dumping all the historical data into the “HistoricalWeatherData” table. It also involved two API calls to the NWS and OpenWeather APIs to load in data for the day the tables were all created. However, because that weather data is frequently updated and published, we elected to call both those APIs daily at 10am to insert the current day’s weather conditions and alerts into our database in real-time.

In order to automate the daily API calls, a separate Python file was created that first connects to the local MySQL server and selects the “WeatherDatabase.” Then, it calls both APIs and inserts the daily data isolated from them (as defined by what is in line with the schema) into the appropriate tables. This Python file is called by a daily CronJob (using Terminal on a Mac) that is set to run at 10am daily. As such, the pipeline extracts, by calling the respective API procedures, executes the transformation in the provided schema, and loads the data into the MySQL database using the embedded INSERT INTO statement, without any additional user effort.

**Results**

**Limitations and Next Steps**

**References**

*API Web Service.* (n.d.) National Weather Service (NWS) National Oceanic and Atmospheric Administration (NOAA). Retrieved February 1, 2024, from https://www.weather.gov/documentation/services-web-api

Bravo, Christina. (2024, February 21). *San Diego weather: Pacific storm brings heavy rain overnight, clear skies ahead*. MSN. Retrieved February 23, 2024 from https://www.msn.com/en-us/weather/topstories/san-diego-weather-pacific-storm-brings-heavy-rain-overnight-clear-skies-ahead/ar-BB1iuha2

*Climate Models*. (2023, August 25). NOAA. Retrieved February 23, 2024 from https://www.climate.gov/maps-data/climate-data-primer/predicting-climate/climate-models

*San Diego Weather - GHCN Daily.* (n.d.) The Scripps Institution of Oceanography at University of California, San Diego (UCSD). Retrieved February 1, 2024, from https://oceaninformatics.ucsd.edu/datazoo/catalogs/ccelter/datasets/13.

*Weather API.* (n.d.) OpenWeather. Retrieved February 1, 2024, from https://openweathermap.org/api.