Technical Documentation

Author: Alexandre Otávio Guedes Drummond

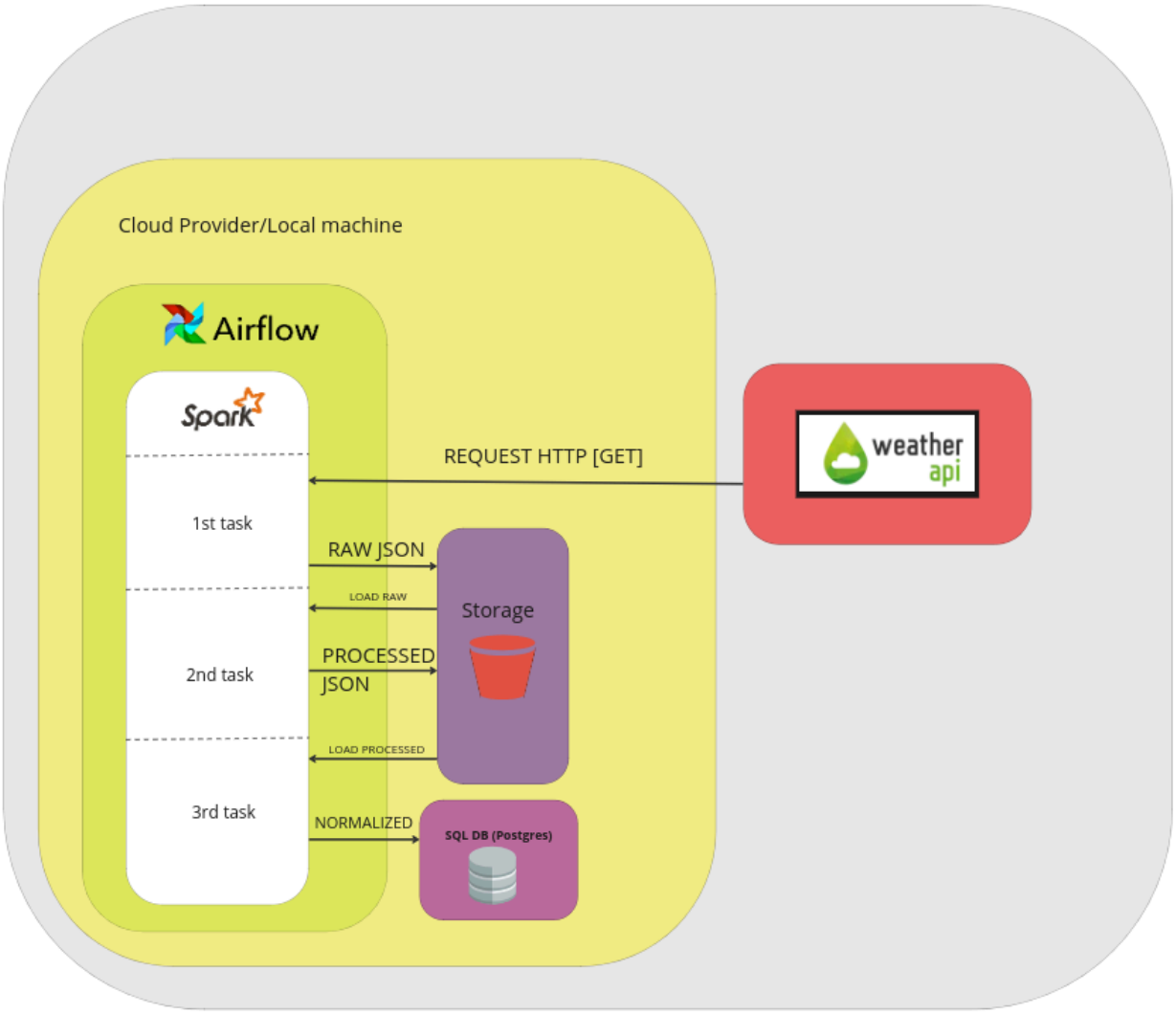
Software Name: Weather Data Warehouse

Software Version: 1.0

General Information

This project is under development aiming to present and employ several modern concepts adopted in Data Engineering, intending to discuss the desired stages and components in a current Data Warehouse. Those topics include, but are not limited to: data extraction (through API or local/cloud storage), local processing, distributed processing, data governance, software architecture, database architecture, data analysis, and design. It is not a restricted project, therefore as it becomes appropriate new topics and functionalities may be inserted.

Project Workflow



Weather Data Warehouse Execution Flow

Weather API: External source for weather data.

Spark: Processing tool used to manipulate data in a distributed manner, leveraging a cluster of machines.

Storage: Module responsible for semi-structured data, corresponding to the project's "Data Lake" component.

SQL Database: Module responsible for storing normalized data, corresponding to the project's "Data Warehouse" component. This is where data is accessed for consumption.

Airflow: Module for orchestration or the pipeline, managing the workflow to ensure data is consumed and processed at the correct moment and sequence, in an automated manner.

Repository Structure

The repository has been organized to support both the expansion and maintenance of the project, while also reflecting the relationships between various tools. The ”dags” directory, under “external”, holds scripts containing workflow definitions (referred to as DAGs in Airflow) and their dependencies. These dependencies encompass all the schemas and source codes required to execute a PySpark process (located in the “src” directory), along with scripts for establishing database connections.

API’s

Weather API

For this project, the chosen data source is the API available at https://www.weatherapi.com/. Weather information was selected due to its abundance, easy accessibility, and free availability, aligning with the project's goal of simulating an application that tackles Big Data Problems.

Accessing the API is straightforward, requiring only a sign-up on the platform to generate a key. This key is then inserted into the route of the GET request, enabling immediate use of the tool. The API allows up to 1 million calls per month, a limit well-suited for the project's requirements.

In addition to climate data, the API offers methods to retrieve various other information, such as marine and astrological conditions and sports events. This diverse information will be collected to enrich the project's database.

It's worth noting that while bulk requests are possible for obtaining data for multiple cities in a single call, the project uses simple requests. This decision is intentional to simulate Big Data conditions, generating a higher demand for processing and creating simulations closer to real-world scenarios.

ACCESS POINTS FOR THE API		
Root URL	http://api.weatherapi.com/v1/{endpoint}.json?key={API_KEY}&q={city}	
FUNCTION	**METHOD**	**ENDPOINT**
Live weather	GET	/current
Astrology conditions	GET	/astrology
Marine conditions	GET	/marine
Sport events today	GET	/sports

REST Countries API

To automatically populate countries' information for the locations stored in the database, we utilized an API with the following characteristics:

ACCESS POINTS FOR THE API		
Root URL	https://restcountries.com/v3.1/	
FUNCTION	**METHOD**	**ENDPOINT**
Country data by name	GET	/name/{country_name}

PySpark

The PySpark framework enables users to harness the power of Apache Spark using Python as the programming language. This widely adopted open-source tool is designed for working with Big Data, facilitating parallelism during data processing. This feature allows application resources to be horizontally scaled with minimal changes to the source code—a crucial capability for Big Data applications where the volume of newly generated data may exceed the system's processing capacity. This scalability enables a computer cluster to function as an integrated device, with the flexibility to automatically or manually adjust cluster resources based on demand.

Apache Spark is engineered to optimize user-executed commands, offering internal mechanisms that support various programming techniques and paradigms without compromising processing efficiency. This flexibility allows users to accomplish the same operation in several ways, each with identical performance. In this project, we chose to explore different approaches to similar operations, presenting options for utilizing PySpark. This exploration includes leveraging built-in functions for tasks that can be solved simply and concisely, as well as employing custom functions that involve advanced commands for a more customized and flexible behavior.

Data storage Architecture

It was chosen a hybrid storage model based on the “Data Lakehouse” modern concept for structure. In this model, raw data is first extracted and stored in a semi-structured format (locally or in the cloud) in its original form. It is then transformed and saved in a processed format in a separate directory. Subsequently, the processed information is loaded back and

inserted into the SQL database, enabling the generation of business intelligence and insights through analytics by querying data directly from the database.

This approach combines the robustness of a Data Warehouse, ensuring transformed data adheres to the expected schema, with the flexibility of a Data Lake in the initial stages. Raw information is always initially inserted into the raw data directory. If it adheres to the expected schema, signifying quality, it is processed and inserted into the Data Warehouse for consumption. If there's a mismatch in the schema, the transformation is halted, and inconsistent data is not inserted. A warning can be triggered to notify the responsible party. Once the new structure is ready, rejected data can be ingested again directly from the Data Lake, ensuring consistent insertion into the database without data lost.

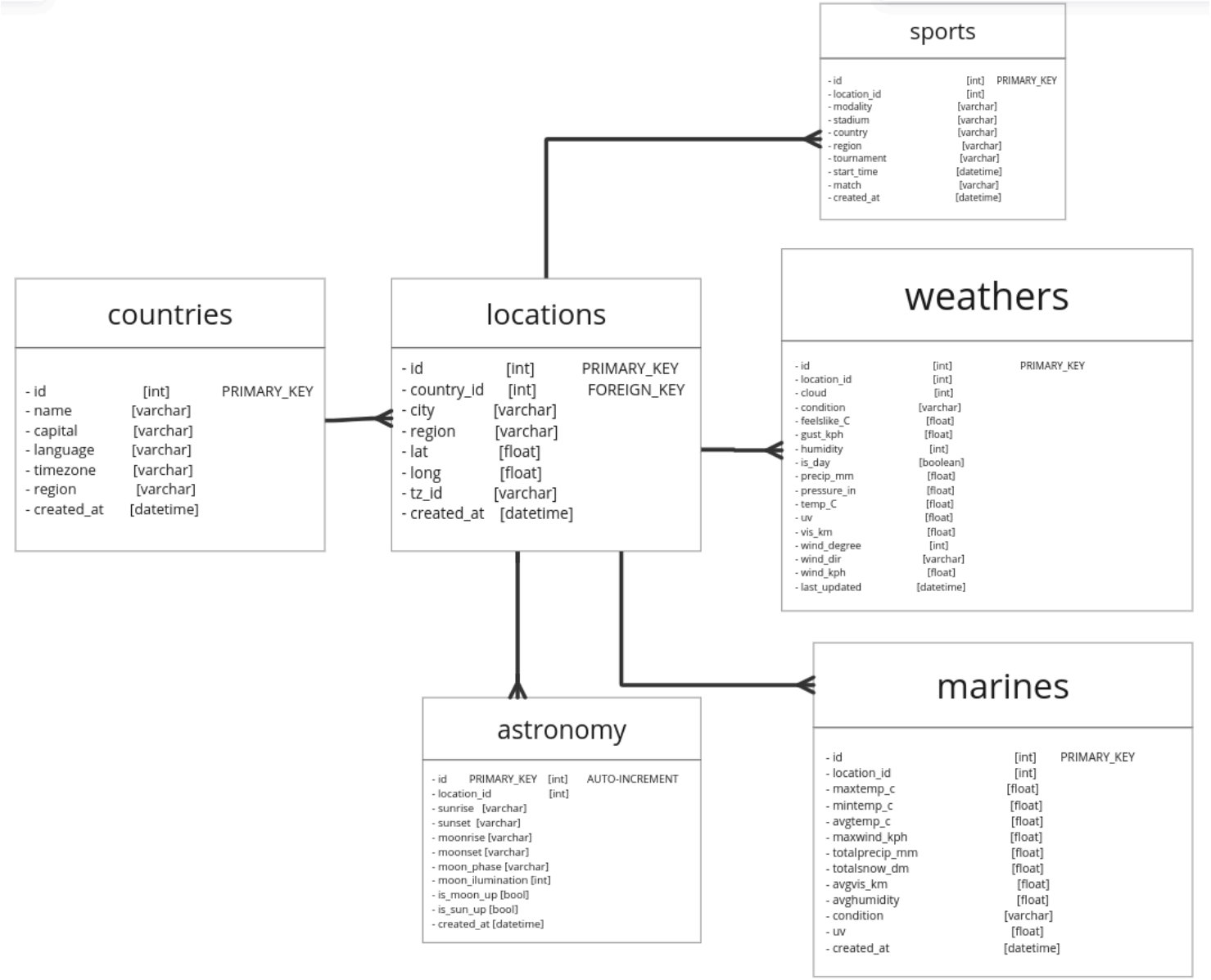
In addition to the data meant for database insertion and consumption, we also store execution logs generated during the processing stage. This helps maintain a historical record and facilitates bug or inconsistency identification during operation.

It's important to note that this strategy as a downside results in increased storage requirements due to storing data in different states (raw JSON, processed JSON, and SQL database). However, with the current low cost of storage available in cloud providers and the advantages of storing data in multiple forms, including the possibility of paying less for infrequently accessed data, our solution offers a favorable cost-benefit ratio.

Database Design

The database follows Inmon's data warehouse structure, leveraging its design principles. The initial storage stage provides high flexibility for ingesting data, and the final stage is tailored to record structured data. This allows correctly formatted information to be inserted and ready for consumption. Any data sample that doesn't meet the requirements is persisted in “raw” or “processed” stages and may later be used.

Given the dependence of all information on the associated location, we opted for implementing a “locations” table to record this data. Each location is assigned a unique ID, serving as a foreign key for other tables designed to accommodate normalized data samples.



Entity-Relation Mapping for the database designed

Airflow

The open-source software Apache Airflow provides a range of functionalities essential for orchestrating data processing workflows. When combined with Python tools, Airflow facilitates the automated execution of processing steps at scheduled intervals. Additionally, it enables specific procedures to be triggered based on the results obtained at each processing stage. Its significance becomes especially pronounced in large systems, where numerous workflows run daily. Centralizing these workflows in a single platform like Airflow becomes indispensable for efficient management in such complex scenarios.

Deployment

The ultimate objective of this project is to deploy it on a cloud environment, enabling continuous operation without relying on local machines. The plan is to containerize the project using Docker, making it compatible with AWS services such as ECS or EKS. However, to streamline the development process, the project is designed to work locally as well. As a result, both local and cloud deployments are available options.

Dependencies

Package	**Version**	**Package**	**Version**
apache-airflow	2.8.0	jsonschema	4.20.0
psycopg2_binary	2.9.9	pyspark	3.5.0
python-dotenv	1.0.0	Requests	2.31.0