Hypothetical Risk-Alert System

Astante Giovanni[1], Fabbri Letizia[2], Vandelli Davide[3]
giovanni.astante@studenti.unitn.it
letizia.fabbri@studenti.unitn.it
davide.vandelli@studenti.unitn.it

***Abstract*— The aim of the project was to compute the geophysical risk associated with a specific location in the United Kingdom. A toy example has been developed to show how even simple computations in Apache Spark can provide a useful output for an hypothetical risk-alert service which, in our view, could be implemented by fetching the extracted information from a Redis database.**

***Keywords* — risk-alert, Apache Spark, Redis**

I. Introduction

The task was about computing online geophysical risk given a location. What follows was our rationale:

- finding **relevant raw data** that could've been ingested and stored in batches, in a scalable and spike-tolerant way. Optimizing for fast write operations.
- Applying computations in a **scalable** way to extract information. Possibly operating on data structures that are somewhat 'compatible' with the raw data format.
- Storing the final information for each location where it is **optimized-for-lookup**.
- Fetching from the final storage the **possible risks** for the current day for any location. Making it available for an hypothetical risk-alert system.

In the first phase, data from weather forecast services and environmental agencies were collected. All of them were semi-structured (JSON or XML format). The variables that were most significant for our purpose were identified.

II. System Model

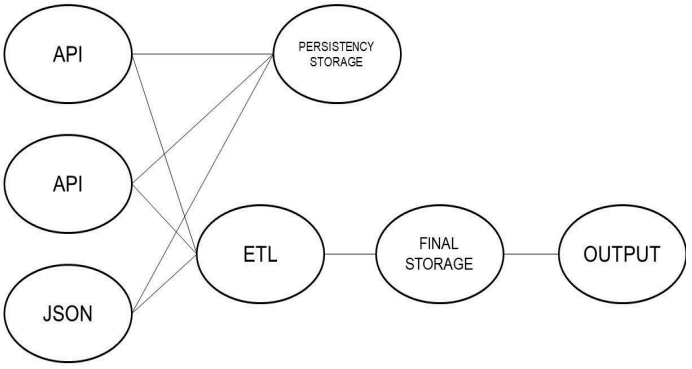
A. System architecture

In the ingestion phase, two data sources are APIs: 3B-Meteo API[1], flood monitoring API from the UK Environmental Agency[2]. The last source is the British Geological Survey website[3] (XML scraped data regarding most recent earthquakes).

All the data is then both persistently stored in a data lake and sent to an ETL pipeline through a fanout architecture.

The ETL pipeline has a dataframe as output: each row represents a city and its risks for the current day. The challenge here was to put the data together in this format since the 3B-Meteo data are organized city-wise and instead the other data are organized region-wise. This issue has room for future development by using location data.

Finally, the dataframe is then converted to a JSON file, stored in a database from which data for a single row can be fetched to implement a risk-alert system (leaving room for implementing dashboards).



B. Technologies

This paragraph goes over which technologies, and in which stage of the pipeline they have been used and why, summed in the table below.

TABLE I
TECHNOLOGIES USAGE IN THE PROJECT

Technology	Stage	Purpose
Apache Kafka	Ingestion	Fanout architectures avoid coupling between data coming in and what is passed to other stages of the pipeline. Optimized

		for fast writing.
Technology	Stage	Purpose
Data Lake	Persistent storage	All the data is semi-structured and a data lake is more scalable than any NO-SQL database. Optimized for fast writing.
Apache Spark	ETL	It's scalable and supports semi-structured data through the data frame data structure. Optimized for batch processing.
Redis	Final Storage	Scalable and optimized for fast writing and lookup operations.

III. Implementation

The implementation is a toy example that takes into account the ingestion without using Kafka, the ETL pipeline using Spark and the final storage into Redis. At the second stage we had to resort to python native methods as Spark was too challenging. Moreover, only the 3B-Meteo API was used for lack of time, and the implementation **lacks a system** to fetch data periodically from it. The code is available on our Github page[4] and is properly commented.

The only script that should be run to execute the code is `main.py`. The other scripts contain the classes imported by the main script for it to work.

In connectors/weather.py the defined class calls the 3B-Meteo API and organizes the relevant data as a dictionary. In `data_model/spark_setup.py` the defined class is designed to transform the dictionary data into a Spark dataframe associating cities and risks. The values are **dichotomized**: if extreme, then a risk is set as True. In `redis_configuration/data_conversion.py` the defined class wraps methods from previously cited classes to fetch data and elaborate them in Spark. It also has a method that converts the Spark dataframe to a dictionary that will be stored into Redis. In the 'redis_operations' – 'redis_configuration' folder – the defined class instantiates a Redis connection and provides methods to store data into Redis and perform a lookup operation. In `redis_setup/query_retrieval.py` folder – the defined class wraps some methods from the classes defined in the scripts in the `/redis_configuration` folder in order to let a user finally perform a lookup operation. Then, the user needs to interact with the program, and input has to be inserted or the program can be exited. See more here.

IV. Results

We've learned how important it is to have a clear idea of the big picture of a project like this from the beginning. Even if we did not have years of programming skills and technical know-how to implement all the stages and technologies in the most appropriate way, we were able to create an example that pursued assumptions for what we studied during the Big Data Technologies course. We could maintain a linear order in the implementation and use two out of the three technologies we wanted nonetheless, which could be regarded as an achievement.

V. Conclusions

Our system has several limitations and there is room for enhancement at each step of the pipeline. A few from a long list we made:

- Ingesting data with Kafka **will be necessary** in order for the system not to break with the first spike in data velocity. Furthermore, time batch for the ingestion should be defined to make the most of it.
- Storing raw data in a data lake should be implemented to provide fault tolerance.
- The calculations in the ETL pipeline should be performed without recurring to python native methods. The system may break if performing over big amounts of data.
- In case of big data usage, a **pub-sub architecture** could be implemented to connect the ETL output to the final storage.

References

[1] https://www.3bmeteo.com/faq/webservice
[2] https://environment.data.gov.uk/flood-monitoring/doc/reference
[3] https://earthquakes.bgs.ac.uk/earthquakes/recent_uk_events.html
[4] https://github.com/FluveV/BDT_code