RouteService

Munkhdelger Bayanjargal[1], Alice Madama [2]

[1] m.bayanjargal@studenti.unitn.it

[2] alice.madama@studenti.unitn.it

https://github.com/AliceMadama/BDTProject2023

***Abstract*— RouteService is a user-centric and efficient service designed to assist commuters in selecting optimal routes to avoid traffic congestion in the area of New York City.**

***Keywords*—**Routing Optimization, Kafka, Docker Compose, Sparked Structured Streaming

I. Introduction

Our project involves the development of a comprehensive software solution that uses both user route request data and traffic volume sensors strategically positioned throughout the urban area of New York. Our solution will employ a Long-Short Term Memory Network to forecast traffic volumes. It will analyze the collected data and provide route recommendations to commuters based on efficiency and suggest the top three optimal routes, considering factors such as distance, estimated travel time, current traffic conditions, and historical data. Additionally, the system will proactively notify commuters about traffic volumes, allowing them to adjust their choices accordingly.

II. System Model

A. *System architecture*

In its simplest form, our system aims to be a request-based solution that warns users about traffic events and returns the top three best routes (Figure 1.1).

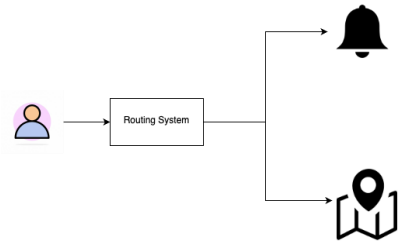

Figure 1.1: A Basic View of Our System

Our architecture is essential and leverages Kafka as its core technology. (Figure 1.2)

Three types of data are inputted in the architecture: weather data, extracted from the Iowa State University database (akrherz@iastate.edu, n.d.), User Request Data, and Traffic volumes, from the New York Open Data website (City, 2016).

In the first stage, users are invited to interact with a TelegramBot by inputting the location of their departure spot, and the desired destination. They are led throughout this process by a welcoming message, which helps them structure their request.

Weather information and Traffic information, Producers for Kafka, are ingested by the Kafka broker, which acts as a communication vase for both the data storage and the near real-time processor. Moreover, User Request Data, also a Producer, is stored in MongoDB to ensure fault resistance. The Traffic Information and Weather information are first cached into Redis for easy access and then stored in MongoDB for long-term conservation.

To conduct the near real-time processing, and define the most efficient route, we used Spark Structured Streaming. Spark Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. It takes care of running the streaming computation incrementally and continuously and updating the final result as streaming data continues to arrive, which is a perfect fit for our traffic data.

Within Spark Structured Streaming (SSS), we can call upon the locally hosted OpenRouteMap API and match the incoming requests with efficient routes and provide possible congestion alerts. The latter is made possible by the implementation of Machine Learning techniques, which forecast the hourly traffic volumes of New York Streets. SSS acts as both a producer and consumer of the Kafka architecture.

The results are resent to the Kafka broker, which stores them in MongoDB for future reference.

In the final stage, the top three optimal routes, and possible alerts are sent to the original TelegramBot to be communicated to the user.

B. *Technologies*

Our project required several technologies to be implemented. To choose the best ones to solve our case, we relied on the vast literature regarding route optimization and real-time systems.

The system is heavily reliant on Kafka as a message broker to ensure proper communication between each stage. Moreover, Spark Structured Streaming plays an important role in the processing stage of our system, ensuring efficient and seamless computations.

To begin with, we decided to leverage **Docker** technologies to encapsulate most of the backend computations. This choice was guided by the necessity to make our system portable and easily scalable, which is fundamental to applying our routing service to larger groups of users (Ferme and Gall, 2016).

We relied on **Apache Kafka** to handle requests, ingest data and distribute the information. Its design it's suited for high volumes of real-time data, as it is described as a distributed streaming platform. Once again this is catering to our necessity of making the pipeline able to deal with higher dimensionality once applied to a larger user base. Most importantly Kafka is a fault-tolerant and durable solution for publishing, subscribing, and processing data streams, since at its core, it is a distributed messaging system that allows for reliable and efficient transfer of data between applications or systems, thus favoring a real-time system.

To cache the data regarding the Weather Information and Traffic Volumes we relied on **Redis.** Redis data resides in memory, which enables low latency and high throughput data access. Unlike traditional databases, In-memory data stores don't require a trip to disk, reducing engine latency to microseconds. Because of this, in-memory data stores can support an order of magnitude more operations and faster response times (Amazon Web Services, Inc., n.d.). This is another fundamental tassel towards defining a well-structured real-time system, that will run smoothly, and reliably.

MongoDB was used to store User Requests data, as a fault-tolerant solution to conserve important information that could be later re-used by the system. MongoDB is built on a scale-out architecture, thus once again perfect for future enlargements of the users' pool. As a document database, MongoDB makes storing structured or unstructured data easy, allowing for more complex relations.

Spark Structured Streaming was utilized for processing and producing the output file. Not only is spark structured streaming a scalable, high-throughput, fault-tolerant stream processing tool for live data streams but it can also be easily applied to Machine Learning. This is possible because Spark Structured Streaming lets you express computation on streaming data in the same way you express batch computation on static data. The Structured Streaming engine performs the computation incrementally and continuously updates the result as streaming data arrives, making it perfectly suitable for real-time operations.

To conclude **a Telegram Bot** was created to implement a proper visualization and interactive interface for the user. We leveraged the library "telebot" in python3. The reason a telegram bot was chosen is both due to practicality and the need for simplicity: this makes our service easy to use for the end customer, and simple to interact with, as all it requires is an internet connection and a device that can operate telegram.

III. Implementation

The implementation of our system has been designed with a focus on simplicity and efficiency, placing the utmost importance on minimizing user response times. (https://github.com/AliceMadama/BDTProject2023) .

To achieve this, we followed a structured approach. First, we modeled the data, taking into account the various variables and constraints that needed to be considered to produce a working prototype. Once the data model was established, we proceeded with data collection and simulation. This step involved generating synthetic data, like userID, username, email, etc., using appropriate libraries, such as Faker. Next, the collected data was ingested into Apache Kafka, which served as the central streaming platform for our system. Simultaneously, parts of the data were also stored both in Redis and MongoDB. Redis acted as a cache for user request information, enabling quick retrieval shortly, while MongoDB served as a fault-tolerant database for storing all system data. The user requests, after being ingested by Kafka, were processed using Spark Structured Streaming. This allowed for real-time analysis and transformation of the data. The resulting output from Spark Structured Streaming was sent to the user through a Telegram Bot, after being saved in MongoDB. Overall, this implementation strategy ensures that user requests are processed promptly, minimizing waiting times and providing a seamless user experience.

A. Data Modeling

The first step of our implementation process was to define the data modeling for the project. To begin with, we defined a **Conceptual Model**, which highlights the entities, and their relationships. Secondly, we outlined the **Logical Data Model.** In this stage, we focused on tabulating data, defining the attributes and the relationships between each table of attributes**.** To conclude, we designed the **Physical Model,** defining all the variable types, allowing us to have a clearer idea of how to deal with them later on.

B. Data simulation and collection

To initialize the system, we used the Faker library in Python to simulate user data. This included generating details such as names, usernames, emails, and IDs for the users. We also captured their routing requests, specifying the longitude and latitude coordinates for departure and arrival locations. Additionally, request times were generated to facilitate the communication of alerts. Simultaneously, we collected data on traffic volumes and weather conditions within the New York urban area. Although our prototype system focuses on functionality rather than broad coverage, scalability was a key factor in our technology choices.

C. Apache Kafka Ingestion

Apache Kafka is a crucial component within our system as it serves as a distributed streaming platform. It facilitates the flow of simulated and collected data from Producers to the relevant Kafka topics, ensuring that it reaches the appropriate Consumers. Our system comprises three Producers: Spark Structured Streaming, User Data Requests, Weather CSV, and Traffic Volumes CSV. User Data Requests, as mentioned before are stored in MongoDB, while Traffic and Weather information are first cached in Redis. It serves as a short-term database, enabling us to swiftly retrieve information shortly. By utilizing Redis, we can avoid the need to generate new information every time, thereby significantly reducing computing time. However, to ensure resilience and fault tolerance, we have implemented MongoDB as a reliable database solution. It serves as a comprehensive repository for all the data inputted into our system. This ensures that we can retrieve information whenever necessary. By having access to the database instead of relying solely on new data products, we can achieve faster computing times and enhance overall system performance.

It is important to note that Spark Structured Streaming acts as a Producer for Apache Kafka, within our system, but also as a Consumer.

D. Route Optimization process

Our system's route optimization process relies on Spark Structured Streaming's capabilities. The Spark Streaming application consists of three key components: the source, the business logic or process engine, and the sink or output. In our case, the input source directly takes the User Request Data and the CSV data. This is then processed by the business logic component, which calls upon OpenRouteService REST API. To enhance the efficiency and seamlessness of the experience, the API is locally hosted, which avoids long waiting times during calls. OpenRouteService returns the fastest, shortest paths between the locations chosen by the user. The results of this process are first saved in MongoDB and also sent back to Kafka for outputting.

E. Alerts with Machine Learning

Additionally, in parallel to the route optimization process, a Long Short-Term Memory (LSTM) model is implemented. This model is employed to predict traffic volumes for each hour. By comparing these predicted volumes to the average traffic volume in the New York urban area, the system can identify congested areas. If the traffic volumes exceed the average, alerts can be sent back to the users to inform them of potential congestion. Our alert system bases itself on a strong assumption: some elements influence traffic flows on specific dates and we expect to be able to isolate them and use them to predict future volumes, specifically the timestamp seemed to heavily influence the hourly traffic volume (Figure 3.1)

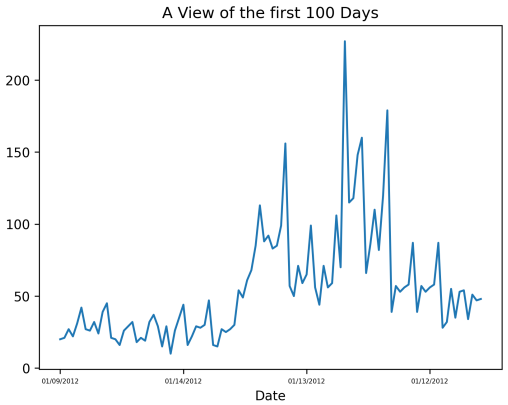

Figure 3.1: A simple view of the first 100 days in traffic volume

F. Telegram Bot

To facilitate the user-system interaction, an implementation of a Telegram Bot was incorporated, streamlining the user's access to the service. Users can conveniently utilize the bot by having a functional

device with internet connectivity and Telegram installed. The bot accepts user requests for routes, including departure and arrival locations. It also collects the connected user information, which is crucial for reconnecting the user with their initial request after the computation stages. The output provided to the user includes a comprehensive map displaying each possible route, along with congestion alerts, duration, and distance for each route.

IV. Results

The system has been successfully deployed with the utilization of over 10,000 simulated user data, containing routing coordinates limited to the New York City area. The demo showcases congestion prediction on an hourly basis and provides users with three optimal route options. One of the notable challenges faced during the project was the development and integration of a Telegram-hosted chatbot.

Additionally, we faced the ongoing task of making well-informed decisions to strike a balance between reliability and efficiency in delivering results.

V. Conclusions

Our system has encountered a few limitations: to begin with, for the alert system to be truly effective, the Machine Learning model would require to be much more complex and well-structured, such as Spatio-Temporal Graph Convolutional Networks (Yu, Yin, and Zhu, 2018). This would allow us to take into consideration many more features and provide more accurate views of the traffic situation in the area. However, due to time constraints and the necessity of focusing on system modeling, we opted for a simpler solution. At the time being our Bot doesn't display the alerts, however, we do have them implemented within the system.

As mentioned before the system currently can only work over the urban area of New York, although the pipeline was built keeping in mind scalability, this is indeed a limitation to our user's possibilities.

We also lack easily accessible diagnostics, it is possible to check for the performance time, etc, however, this is not elegantly summed up in a dashboard, rich in informative visualizations.

Being this a prototype it is lacking a lot of features that could easily improve the user experience and overall quality of the product: we are lacking a feedback system, that could let us track user preferences and potential faults; it would be good practice to have a double checking system for the traffic flows, to ensure that the model is working accurately and faithfully to reality.

Another issue we must face is the collection of user data: the Telegram username is a piece of personal information, and thus should be handled under the GDPR. We have considered implementing a disclaimer in the Bot which warns the user their userID must be collected in order to use the service, therefore to abstain from usage if not agreeing to such treatment of their data.

It would also be interesting to substitute Spark with Flink, to observe if such a change could improve our response time.

To conclude, the Telegram bot doesn't consider all of the possible exceptions, it's just a very simple form of interaction between users and machines.

Although aware of our limitations we are proud of what we have produced and hope to continue working on this in the future, to express the project's full potential. Until then "So long, and thanks for all the fish!".

References

[1] akrherz @iastate.edu, daryl herzmann (n.d.). *IEM :: Download ASOS/AWOS/METAR Data*. [online] mesonet.agron.iastate.edu. Available at: https://mesonet.agron.iastate.edu/request/download.phtml?network=CO_ASOS [Accessed 26 Jun. 2023].

[2] Amazon Web Services, Inc. (n.d.). *Redis: in-memory data store. How it works and why you should use it*. [online] Available at: https://aws.amazon.com/redis/.

[3] Apache Kafka. (n.d.). *Apache Kafka*. [online] Available at: https://kafka.apache.org/documentation/.

[4] Bajaj, G., Agarwal, R., Bouloukakis, G., Singh, P., Georgantas, N. and Issarny, V. (2016). Towards building real-time, convenient route recommendation system for public transit. *2016 IEEE International Smart Cities Conference (ISC2)*. doi:https://doi.org/10.1109/isc2.2016.7580779.

[5] City (2016). *NYC*

[6] *Open Data*. [online] Cityofnewyork.us. Available at: https://opendata.cityofnewyork.us/.

[7] Ferme, V. and Gall, H.C. (2016). *(PDF) Using Docker Containers to Improve Reproducibility in Software and Web Engineering Research*. [online] ResearchGate. Available at: https://www.researchgate.net/publication/303515069_Using_Docker_Containers_to_Improve_Reproducibility_in_Software_and_Web_Engineering_Research.

[8] https://github.com/mongodb/docs-bi-connector/blob/DOCSP-3279/source/index.txt. (n.d.). *MongoDB Documentation*. [online] Available at: https://www.mongodb.com/docs/.

[9] pytba.readthedocs.io. (n.d.). *pyTelegramBotAPI Documentation 4.7.0 documentation*. [online] Available at: https://pytba.readthedocs.io/en/latest/index.html.

[10] Redis. (n.d.). *Introduction to Redis*. [online] Available at: https://redis.io/docs/about/.

[11] Ricci, F. and Tumas, G. (2009). Personalized Mobile City Transport Advisory System Create new project 'ONE Project' View project Music recommender systems View project Personalized Mobile City Transport Advisory System. doi:https://doi.org/10.1007/978-3-211-93971-0_15.

[12] spark.apache.org. (n.d.). *Structured Streaming Programming Guide - Spark 2.4.5 Documentation*. [online] Available at: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

[13] Yu, B., Yin, H. and Zhu, Z. (2018). Spatio-Temporal Graph Convolutional Networks: A Deep Learning Framework for Traffic Forecasting. *www.ijcai.org*, [online] pp.3634–3640. Available at: https://www.ijcai.org/proceedings/2018/0505 [Accessed 26 Jun. 2023].

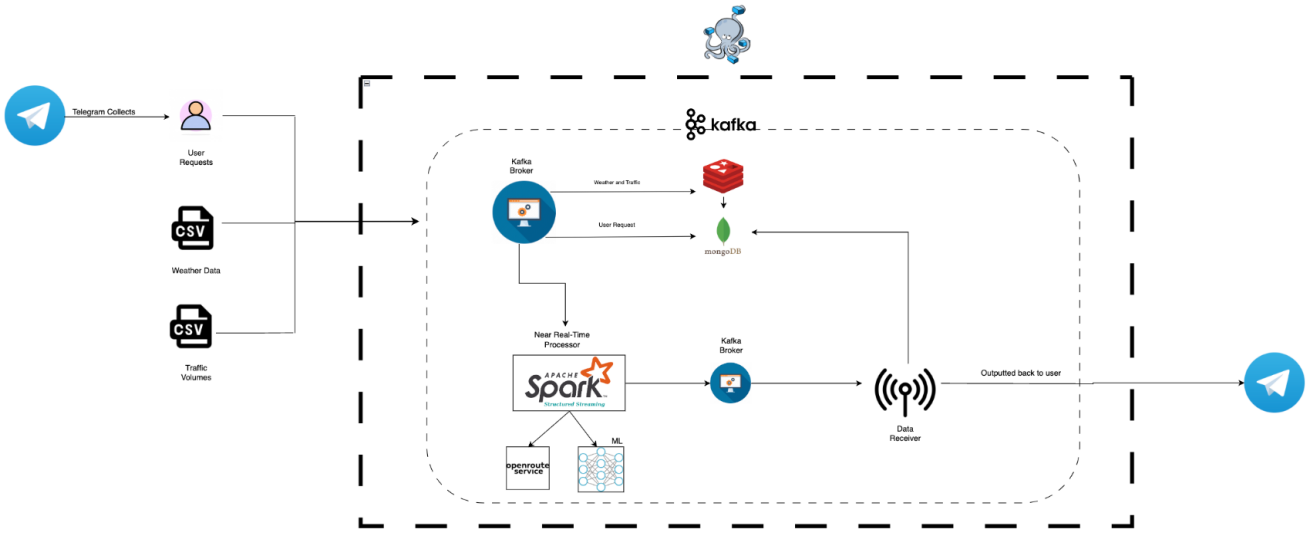

Figure 1.2: Architecture