Big Data system for news article recommendation

Arturi Martina[1], Meloni Alessia[2], Peracchio Roberta[3]

[1]martina.arturi@sudenti.unitn.it

[2]alessia.meloni@studenti.unitn.it

[3]roberta.peracchio@studenti.unitn.it

Abstract **- This big data project aims at producing a list of suggested articles for users, based on their past behavior on a news website. The task was dealt with by simulating a stream of clicks, with fake users randomly viewing articles present in a database. The system produces and sends back a list of suggested articles for a given user, based on preferred categories, new and popular unread articles.**

Keywords **- Clickstream, Recommendation Big Data System, Kafka, Spark, MongoDB.**

I. Introduction

The aim of this project is to design and implement a big data system for articles recommendation on online news, based on the users' past online activity.

At first, it was important to define what online activity could be registered and gathered, since the ownership of the news websites was not in the possession of the writers, and tracking cookies as well were not legally accessible. Moreover, using real life data would raise issues regarding the privacy of the users of a news website, needing to request permission for data usage.

Another issue arises regarding the news website to access. Most news websites have a monthly limited amount of free articles; after having viewed such an amount, the website blocks the user from reading more articles with a paywall.

For this project, clicks on previous articles are considered as the past activity, from which the big data system takes information on what a user usually reads, then sends back articles which a user might like.

A clickstream is generated randomly, just for the purpose of demonstrating the work of the app and gathering information. In this manner, privacy regulations will not be involved and the project was able to be concluded safely.

II. System Model

A. System architecture

The database is composed of four collections, holding information about the articles, the clicks, the users and the suggestions.

The "articles" collection has as keys the category, the headline, the authors, the link, a short description, the date and an index; the index holds information on the category of the article and a number, to differentiate it from the others.

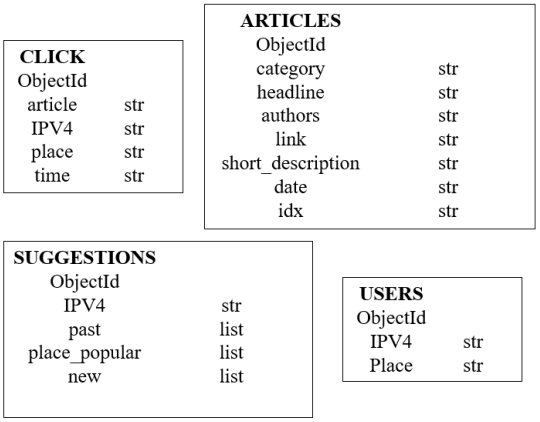

Fig 1. Database collections

The "users" collection holds information on the IPV4 and the postal code, identifying from which a user is producing the click. In this case, the IP also works as the ObjectId of the collection, as it is unique.

The "click" collection holds information on all the clicks generated. Particularly, it has the article's link, the IP and place of origin of the users and the time at which the click was made.

Finally, the Suggestion collection includes the user's IP, a list of the current week's most read articles, a list of popular ones based on location, and a list of the newest articles on the website.

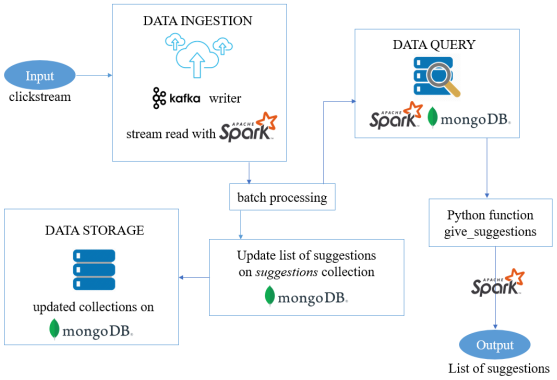

Fig. 2 System pipeline

The suggestions are computed once a day. In that instance, the system detects the clicks and updates the suggestions made for each user.

It is checked through Spark whether a user already exists in the database (in our case, MongoDB was the chosen service), by comparing the user's IP with the ones already present.

If a user is not already registered in the database, a list of articles based on the location and the newest posts is sent. The computation of the suggestions is made once a day.

If a user is in the database, the suggestions also include unread articles based on past behavior; that is, the most read categories are computed and the two most preferred are used in the suggestions.

B. Technologies

As it has been already stated, for our project we used three main technologies: Kafka, Spark, and MongoDB. We also investigated other programs like Redis, that would have doubled as a pub/sub and a database, but we thought that this double feature would have resulted in many lost messages. After careful research we decided to discard this idea and go with the three aforementioned ones. In order of our pipeline, we will explain the motives behind our choices.

As the original use case for Kafka was Website Activity Tracking, we thought it was the best fit for our project, as it is about making suggestions based on the user's previous activity on a website. Kafka is indeed focused on ingesting massive flow of data, providing a flexible, scalable, and reliable method to distribute streams of event data from one or more producers to one or more consumers. These streams are basically organized into topics, where a producer chooses to send the message to; consumers typically receive streamed events in real-time but, and this is one of the many reasons we choose Kafka, the program persists data internally allowing consumers to disconnect and then catch up once they are online again. The consumer can also request the full stream of events. Unfortunately, we must be aware of some of the limitations. The major problem is that the replication of the messages is asynchronous and can result in lost messages.

To limit this problem, we used Apache Spark to read all the Kafka streams, to be later written to MongoDB. We specifically choose Spark because it provides a faster and more general data processing platform and lets you run programs up to 100x faster in memory, or 10x faster on disk, than Hadoop; it also comes with packaged with support for Machine Learning, SQL, R and streaming and supports a variety of popular development languages (we wrote our code only using Python).

As for the database the choice of using MongoDB came naturally, as the native MongoDB connector for Apache Spark provides higher performance, greater ease of use, and access to more advanced Apache Spark functionality. The Spark engine can extend the real-time processing of operational data managed by MongoDB, while the Spark connector can take advantage of MongoDB's aggregation pipeline to extract, filter, and process only the range of data it needs, as we are going to need this exact function to give suggestions back to the user.

III. Implementation

We decided it would be useful for anyone and for future references to have a way to access and look at our implementation of the project, so we uploaded it on GitHub at this link [https://github.com/alessia96/BDT2022-group5]. We will, briefly, explain here how the code is structured. All of the technologies needed can be used with Python, so, apart from the specific packages to connect our programs with Python, the execution is not difficult as the code is written in only one programming language. For a more complete guide please refer to the documentation attached within the GitHub link.

The first thing to do was necessarily creating users, clearly one cannot use real people's data as they are sensitive data. So, we decided that

the Python Faker package would be ideal for creating our users. In a dictionary called 'users', through a 'for loop', we saved the ipv4 and the place (as postal code) for each of them, storing them in a MongoDB collection. This part was run only once to create the fake users.

After the creation of our users, it is necessary to create a message producer to publish the stream of clicks to a topic, so we decided to use Kafka and CloudKarafka for this part. In our case our code is a simulation of a newspaper's website sending us users' clicks via message. After that, once the click message is published, it is read through Spark which simultaneously saves the information contained in it in a MongoDB collection ("click"), and with the use of Kafka send a message containing a list of suggested articles for a given user. These two parts of our implementation, as they are streams of data, are always running.

Clearly it is also necessary to find a way that generates suggestions for users. We think that the most efficient way to achieve this result is to produce a collection with lists of suggested articles for each user and place (in order to suggest articles to new user that don't already exists in our database); if the user exists we want to suggests articles based on his/hers past behavior, the most popular ones based on time and place and the newest. On the contrary, if the user isn't already present in the database we are going to recommend only the most popular based on time and place and newest. To obtain the most popular articles in the last week we must take the last week's clicks and then count the frequency each article has been read. The "time_popular" suggestion list will contain the 30 links that have been clicked most. To get "place_popular" we consider the nearest postcodes, initialize an empty dictionary in order to save the 30 most clicked articles for each base postal code (postal code with a step of 100, e.g., 80100, 8200, etc.). The last step is to create a function to get the most read category of a user in order to suggest new articles based on that topic. We initialize an empty dictionary where we'll save categories and count, sorting the categories to see which ones were the most read and adding 15 article links for the first most read category and 15 other for the second one to a new list; for each user we will automatically create the suggestion lists and write it on the mongo collection "suggestion". We will update this every day.

As per the last step we must create a consumer through Kafka to receive suggestions. The consumer is always running, as its function is pivotal to be able to send the suggestion back.

IV. Results

Following our implementation and after the initialization of our various scripts; we have at the end, as a result a list of suggestions to send back to the website of the newspaper that requested them for their user in the first place. Clearly having several scripts, we have more output that now, in almost a chronological order, we are going to analyze.

Fig. 3 Snippet of collection articles of BDT database

We decided to include also the first screenshot (Figure 3) for completeness, as it is not a real result, but it is the first line of our database in which our articles are stored (based on the Huffington Post's dataset). In there, one can find the id of each article, the category to which it belongs, the title, the authors, the link, a brief description, the publication date, and the index (composed of the category and a code)

Fig. 4 Snippet of collection click of BDT database

The first actual result is included in figure 4 and is the output we have following the

implementation of our "click_streaming.py". Here one can find the id of the article clicked by the user, the link of the article, the ipv4 of the reader, the place, the date and time of when the click was made

Fig. 5 Snippet of collection suggestions of BDT database

In figure 5 one can see the output (not the complete one as it would have been too long) of the suggestion generator. There is the ipv4 of the user and 4 lists (in order: the one related to past behaviors, the most popular at that time, the most popular based on the user's postal code and newest one, in which there are 30 links of articles to be suggested to the user.

As for the last (figure 6), one could say actual result, is the output of the *consumer.py*, that sends the suggestion back to the newspaper, where one can find the ipv4 and the suggestion link.

Fig. 6 Consumer output

From these results we can confidently say that the choice to use Kafka to stream, Spark to write messages on the database and MongoDB as a database concurrently was the best choice. Everything worked, indeed, without too many hitches in a very smooth and linear way.

V. Conclusion

During the project we faced several problems. First of all, how it was possible to create this fake stream of data without having a website available from where to receive and save the clicks, we tried to solve it by pretending that we were asked by a third party (who provided us with the data) to create suggestions that they would have sent back to their users. One of the possible improvements for the project is the creation of an actual newspaper website knowing that that would make everything more truthful and closer to the real world. Unfortunately, due to the time limits we weren't able to implement it.

We also believe that not having a real site that makes the request we cannot be 100% sure of the functionality; we suppose that in the real world it is more complex to manage this continuous stream of data making it necessary to implement something more robust.

Another problem we ran into was the fact that we didn't want to save Kafka's message to later look it up on MongoDB. We thought that skipping this step would save us both space and time at computational level. We, after some careful consideration and research, managed to solve this problem. Through the "writeStream" function, we load the messages on Spark dividing them into batches and for each batch we write on MongoDB getting the relative suggestions.

Despite the various problems that emerged through the course of the project and the possible improvement, we are certain of the functionality of our program and that Kafka, Spark and MongoDB are the best technologies to use together for "Website Activity Tracking" having led us to an excellent performance.

References

[1] MongoDB. (November 2017). *Data Streaming with Apache Kafka & MongoDB*. [White paper]. https://www.mongodb.com/collateral/data-streaming-with-apache-kafka-and-mongodb

[2] MongoDB. (October 2018). *Unlocking Operational Intelligence from the Data Lake*. [White paper].

https://www.mongodb.com/collateral/unlocking-operational-intelligence-from-the-data-lake

[3] MongoDB. (November 2017). *Apache Spark and MongoDB - Turning Analytics into Real-Time Action.* [White paper]. https://www.mongodb.com/collateral/apache-spark-and-mongodb-turning-analytics-into-real-time-action

[4] Kafka Official Documentation - https://kafka.apache.org/documentation/

[5] Spark Official Documentation - https://spark.apache.org/docs/latest/

[6] MongoDB Official Documentation - https://www.mongodb.com/docs/