Realtime Stream Processing for Twitter Analysis

ELEN 6889 Large Data Stream Processing (2019 Spring)

Penghe Zhang (pz2244), Linnan Li (ll3235), Zhicheng Wu (zw2497), Boyuan Sun (bs3113)
Group 16
Columbia University

Abstract

Learning from the success of Facebook's streaming system in '*Realtime Data Processing at Facebook*'[1]. We implemented our streaming system shown in the *figure 1*. Introducing the *Spark Structured Streaming* to work along with *Kafka* and *Google BigQuery*, we construct main body of our streaming system to process twitter streaming data. We developed Realtime *Top k hot topics*, *Topic trends and Sentiment Analysis*. On the other hand, the third visualization application, Heat map with static data, is built on Google BigQuery data warehouse. To assure the full throughput of our system, both structured streaming and data warehouse servers are connected to Kafka as consumers.

1. Introduction

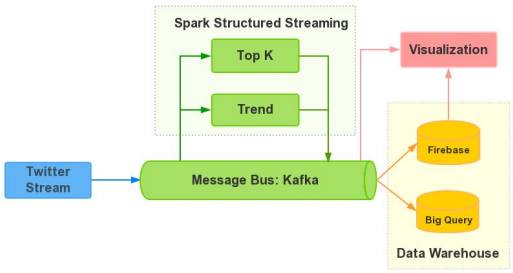

figure 1. Twitter streaming system

We introduce our Twitter streaming system showing possibility of providing real time update information for over 126 million Twitter active users every day. We import our real time data from twitter streaming API and feed such data stream into Kafka as producer. The acquire data is pulled in live as stream and stored in different topics of Kafka. Both data warehouse and Spark are treated as consumers, retrieving data from topics to which they subscribe.

2. Twitter Stream Data Source[2]

First, we are going to talk about our data source, the Twitter Stream API. We set the Twitter Stream API (by Tweepy library[3] in python) as the producer of our application, it will send very roughly 60 tweet json per second, which is also nearly 1% of the all tweets being tweeted at the moment. From all these attributes, what we use are the `created_at`, `entities`, `text`, `place`. In the `entities` attribute, we can get to know the `hashtags` of this specific tweet (will analysis further in *Section 4*); and for the `place` attribute, we will use the coordinates it provides in the batch processing visualization (will analysis further in *Section 5*). The Twitter Stream API will send its stream directly to the Kafka message bus, which will demonstrate in the next section.

3. Kafka[4]

In this section, we will discuss about functionality of Kafka. As we said in the previous section, all the data stream will send directly to the Kafka message bus, and we will demonstrate the advantages of using a Kafka as message bus below.

3.1 Why we use message bus?

Learned from the Facebook's paper[1], we knew that by using a message bus, we can benefit a lot on this middleware. For example, we can reuse the data in the message bus, and it will support multiple clients, also it allows asynchronous producers and consumers.

3.1.1 Reuse data

As the structure shown in *Figure 1,* we know that we will user the same data stream in the Kafka twice, one is for getting Top K Hashtags, one is for calculating the trend of the Twitter. If we are not using Kafka, then we cannot do analysis on the same data.

3.1.2 Asynchronous producers and consumers

By using Kafka, we can keep producers working regardless of the condition of the consumer. Whenever a consumer need to consume data, just read from the Kafka, he/she will get the stream. In this way, we can avoid delay in the Twitter streaming API delay as the only job for the producer is just push the stream to the Kafka rather than wait until consumer finish his/her job.

3.1.3 Multiple clients

This is also an advantage of Kafka, we can serve multiple clients in the same time, because different clients may have different group ids, and different group ids bring different offsets, different offsets assure different clients consumer their own data.

3.2 Why we choose Kafka

The reasons we choose Kafka as the broker is basically as below: On one hand, it is open source and an Apache application, which means it has strong compatibility with other Apache things, like Spark. On the other hand, traditional message brokers provides publish options like publish-subscribe and point-to-point mode, while Kafka provide us topic mode. Also , traditional message brokers mainly support non-persistent messaging, it means no message-replay support, which is also not what we want. Thus, we decide to use Kafka as our broker.

4. Spark Structured Streaming and Algorithms[5]

Instead of Spark Streaming, we use spark structured streaming as our stream processing engine. Spark Structured Streaming is built on Spark SQL. Therefore we can use SQL like paradigm to write applications.

There are two ways to interact with Structured Streaming API including writing SQL programmingly, DataFrames and DataSets. Python does not fully support for the Dataset API. So we use SQl and DataFrame API. It is easy to switch between these two API, even combine them in the same application.

4.1 Kafka integration and Input

Comparing to Spark Streaming, Spark Structured Streaming is a new way to interact with streaming. Currently, Spark Streaming doesn't support Kafka over 0.10 version using Python while Structured Streaming support Kafka broker version 0.10.0 or higher.

To process stream data, Spark need to use `writeStream` to start processing. It will process stream data as small batch which we can decide the length of it. On default it is processed by micro-batch. The system will check for availability of new data as soon as the previous processing has completed. (*figure 2*)

```
df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "35.243.144.79:9092") \
    .option("subscribe", "tweepyv1") \
    .option("startingOffsets", "latest") \
    .option("failOnDataLoss", "false") \
    .load()

    .selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)") \
    .writeStream \
    .format("kafka") \
    .outputMode("update") \
    .option("kafka.bootstrap.servers", "35.243.144.79:9092") \
    .option("topic", "slow") \
    .option("checkpointLocation", "./logslow") \
    .trigger(processingTime='60 seconds') \
    .start()
```

figure 2. Kafka integration and input code example

In our Topk application, it only needs to be triggered as the length of the sliding window. But for other applications if they need to check result of each input as soon as possible, it need to be set as continuous processing or micro-batch.

4.2 TopK Application

Using SQL to write TopK application is really simple. The only thing we need to do is groupby window, hashtag and count. This is convenient for who are familiar with SQL syntax. (*figure 3*)

```
TopK   =  df.withWatermark("timestamp", "10 seconds")\
          .groupby(df.hashtag, window("timestamp", "500 seconds", "60 seconds"))\
          .agg(count('hashtag'), avg('sentiment'))
```

figure 3. TopK application code example

4.3 User Defined Function[6]

It is an important update for Spark to support Pandas in Python API. Because it can take the advantage of Numpy as the compute engine of doing matrix computation. Spark use Apache Arrow to convert data from JVM to Python. When we register a Pandas UDF, it can automatically transfer to a column-based vectorized data. It is easy to use this as aggregation function in Group By.

The problem of this method is it is hard to debug the pandas UDF. Due to the implementation of Spark, the function using the @pandas_udf decorator can not print any middle processes. It is hard to decide the shape and type of data. To get the trend of each topic, we apply window function on window timestamp. Our goal is to calculate the difference between several windows and then use linear regression to get the slope of the topic count. This slope can be directly used as the trend of a topic within a specific window. (*figure 4*)

```
from sklearn import linear_model

@pandas_udf("key string, value double", PandasUDFType.GROUPED_MAP)
def trend_udf(key, pdf):
    reg = linear_model.LinearRegression()
    reg.fit(np.array(pd.to_datetime(pdf.timestamp).astype('int')).reshape(-1,1),
            np.array(pdf.count_num).reshape(-1,1))
    return pd.DataFrame([key + (reg.coef_[0][0],)])

dftrend = dftrend.groupby(window("timestamp", "500 seconds", "60 seconds"),
                          'hashtag').apply(trend_udf)
```

figure 4. UDF(Pandas) code example

Spark also support another kind of UDF which does not use Pandas. This is more simple to use. We use pandas_udf as aggregation function while the simple udf can be used as individual data interaction. Our sentiment Analysis is based on this kind of UDF. In our application, we introduced a trained model from textblob which calculate sentiment score for each sentence. For each topic, we get the mean of this sentiment score and display it on our visualization part. *(figure 5)*

```
from textblob import TextBlob


@udf(FloatType())
def senti(x):
    blob = TextBlob(x)
    s = []
    for sentence in blob.sentences:
        s.append(sentence.sentiment.polarity)
    return sum(s)/len(s)

df = df.select('timestamp',\
               get_json_object('value') \
               senti(get_json_object('value')))
      .filter(df.hashtag.isNotNull())
```

figure 5. UDF(Spark) code example

5. Data Warehouse

The data warehouse of this system provides static data querying as well as reliable data storage for all results.

5.1 Data warehouse vs. Database

Choosing data warehouse over databases, we design our system to process huge amount of data with large variety in data types. Data warehouse as online analytical processing method is more compatible with our purpose. Like BigQuery, we choose it over many databases due to its monolithic nature. In other work, we don't need relational database which perform multiple joints for single query. This lets visualization and analysis easy and fast.

5.2 Google BigQuery[7]

Reducing from facebook's Hive-Laser data warehouse design, we implemented Google's BigQuery data warehouse along with firebase realtime database. BigQuery covers a large range of time accessing for history data querying up to years. Both of the BigQuery and Firebase databases read from kafka streaming performing as consumers. Databases can acquire the data in both batching and streaming. For the purpose of low latency we storage our data in the streaming fashion. The input data are processed as long bytes of the strings that we need to parse the timestamp and location information from original strings. Google BigQuery keeps a various source of data that are coming into the streaming system and keeps those data for later analyzing.

5.3 Google Firebase[8]

Unlike Bigquery, firebase is a data cache for some temporary datas. The living time of each batch in firebase is related smaller. In the other hand, firebase only stores post-processing data which is different from BigQuery who keeps all the raw data for the system. From another point of view, firebased can be treated like a middleware for practical applications inside or outside the system.

6. Visualization

We implemented a web application to show how our stream system works. The application is basically combined with three parts: the first part is a real-time top-k presentation which using one of the data warehouse firebase; the second part shows the trend of about 20 hashtags along with time. It is an application in out whole system that directly reads data from out message bus Kafka; the third visualization is a practical application of the whole system's data warehouse BigQuery. As BigQuery stores all the data appear in the system, it can provide past data for further usage.

6.1 Real-time top-k bar chart and wordcloud

First of all, the backend server reads data from Kafka using several Kafka consumers. These consumers will retrieve data from different topic. Generally there are three different topics in our system which are generated by the data processing part. They are "top k with 10 seconds", "top k with 1 minute" and "data trend". In this part, the server will consume the first two topics and using a transactional batch-writing to put the data into firebase. Each time the server detects a new timestamp, it will update the database with newly top hashtags. It is built in D3.js which using a "snapshot" function to monitor the firebase based on the web-socket protocol.

6.2 Real-time trend line chart

The visualization of this part is designed by Chart.js, and to send streaming data to the frontend, we implement a Flask streaming backend which serve as a Kafka consumer who will continue reading data from the topic "data trend". When the servers consumes a new batch of data with a new timestamp, it will send the data to the front end and reload the webpage to update the data.

6.3 Batch visualization: heat map

This part is built with GoogleMap.js and basically it will reflect the number of tweets related to different regions in US. A user can set a specific period of time in the webpage, then server will send a SQL query to the Google BigQuery according to the time period. When it receive the response from the data warehouse it will reflect all the changes in the web page. The google map will show a heat map according to each region's tweet counts in this specific time period.

7. Performance Analysis and Future work

According to our measurement, the speed of our tweepy producer is 8.03 item/second. No matter the implementation of our spark application, it can produce as full speed. The Shuffle Write Size / Records of spark application is 38.4 KB / 617.

Talking about future work, we decide to enrich the sentiment analysis part, not just a simple polar type judgement, we may use our own trained model in the UDF to analysis more emotions; besides, we may also implement on trend predicting based on the past topics trends in the data warehouse by Spark MLib.

8. References

1. Chen, Guoqiang Jerry, et al. "Realtime data processing at Facebook." *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016.
2. https://developer.twitter.com/en/docs/tweets/filter-realtime/overview.html
3. https://www.tweepy.org/
4. Kreps, Jay, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing." *Proceedings of the NetDB*. 2011
5. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
6. https://spark.apache.org/docs/latest/sql-pyspark-pandas-with-arrow.html
7. https://cloud.google.com/bigquery/docs
8. https://firebase.google.com/docs

9. Appendix

1. The code repo Github link (https://github.com/zw2497/Twitter_Stream_Processing)
2. The results is shown in our web visualization. Below are some demos of the webpage.

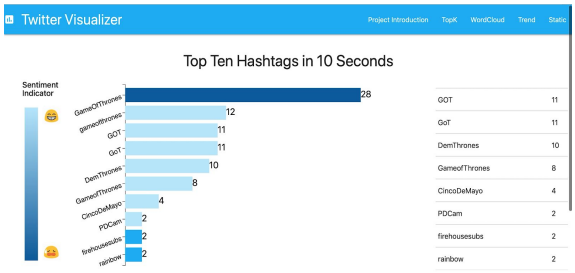

figure 6. Real-time TopK hashtags bar chart

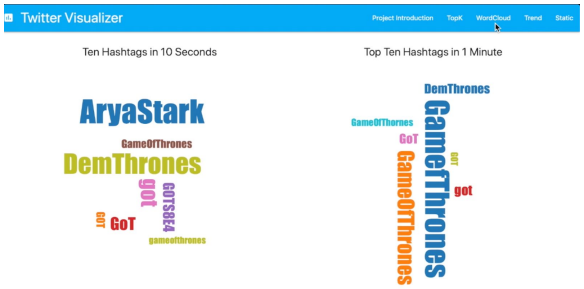

figure 7. Real-time TopK hashtags word cloud

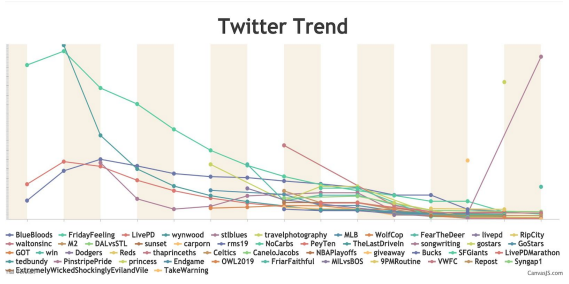

figure 8. Real-time trend chart

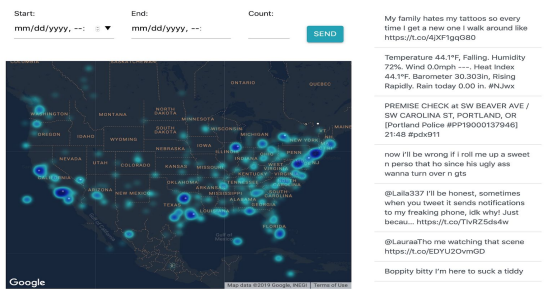

figure 9. Tweet counts heatmap