A Comparison Study: Spark vs Flink for Real time Geo-spatial Data Analysis

Aravind Alagiri Ramkumar

Abstract

The main goal of this project is to compare the most widely used stream processing systems in terms of various parameters such as ease of use, ease of implementation, types of queries and the performance of the systems under various use cases.

1. Introduction

Large amounts of data are generated by social media websites such as Facebook and twitter. Performing a real time data analysis on such kind of data is a good test case for comparing the stream processing systems. The project focuses on the comparison of the most widely used stream processing systems namely Apache Spark and Apache Flink.

The report is structured as follows. Section 2 provides a basic background of the stream processing system and its fundamental concepts. Section 3 explains the system architecture and workflow used for the comparison. Section 4 explains the various use cases, its implementation and performance results for both the systems. Section 5 provides a result of our analysis. Section 7 explains the implementation of the real time geo-spatial data analysis use case. Section 6 concludes the report.

2. Background and fundamental terms

This section will provide a background on the stream processing systems in question and describe a few fundamental terms that will be used later in the report.

2.1 Streaming Processing Model

There are two major types of model in which the data is processed in most of the stream processing systems. They are as follows. *Data flow model* where a processing system is continuously ingesting data that is processed at a tuple level by a DAG of operators. *Micro-batch model* in which incoming data is grouped during short intervals, thus triggering a batch processing towards the end of a time window.

Apache Spark operates on the Micro-batch model and *Apache Flink* operates on the Dataflow model. The applications are generally structured as Directed Acyclic Graphs of operators. The streaming data passes through a series of operators which applies transformations producing another data stream or the result.

2.2 Timing Concepts

Processing time refers to the system time of the machine that is executing the respective operation.

Event time refers to the time that each individual event occurred on its producing device.

Ingestion time refers to the time that events enter the streaming system.

2.3 Windowing Concepts

Windows are at the heart of processing infinite streams. Windows split the stream into “buckets” of finite size, over which we can apply computations. In stream processing, windows sizes are defined by means of the time (any one of time explained in Section 2.2) or by means of count of number events. The two main types of windows are as follows.

Fixed windows assigns each element to a window of a specified window size. Fixed windows have a fixed size and do not overlap. For example, if you specify a fixed window with a size of 5 minutes, the current window will be evaluated and a new window will be started every five minutes.

Sliding windows assigns elements to windows of fixed length. Similar to a fixed windows, the size of the windows is configured by the window size parameter. An additional window slide parameter controls how frequently a sliding window is started. Hence, sliding windows can be overlapping if the slide is smaller than the window size. In this case elements are assigned to multiple windows. For example, you could have windows of size 10 minutes that slides by 5 minutes. With this you get every 5 minutes a window that contains the events that arrived during the last 10 minutes.

2.4 Sentiment of a tweet

Sentiment of a tweet indicates the mood or emotion of the text present in the tweet.

3. System Architecture and Methodology

This section explains the architecture of our system that is used to compare the stream processing systems.

3.1 Architecture

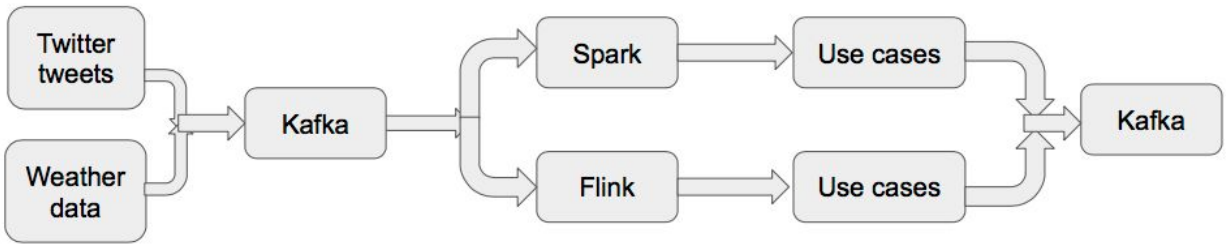

Figure 1: System Architecture

All of the use cases that are described in the following section make use of this architecture and follow the same set of common steps:

1. US location based Geo-tagged Tweets are streamed from the twitter and published to kafka topic named “tweets”. The weather data is generated for all the state capitals in US and publishes to kafka topic named “weather”.
2. The stream processing systems subscribes to the topics “tweets” and “weather” and pick up the data from kafka.
3. Use case dependent processing is then performed in the stream processing system.
4. The processed results are written back into a different kafka topic.
5. The total time taken is measured by finding the end to end time taken to process all the streamed tweets.

We analyze the time taken to perform our entire architecture in each use case for both Spark and Flink. We calculate this time by subtracting the *Kafka ingestion time of the last record in step 4* with *the Kafka ingestion time of the first record in step 2*.

3.2 System Setup

- ❏ The batch size of Apache spark is set to be 100ms to emulate a real time streaming behavior.
- ❏ Tweets are streamed at a rate of 10-12 tweets per second using Python API.
- ❏ Weather data is generated at a rate of 10 messages per second using Python Script.
- ❏ Spark and Flink are subscribed to both the data streams from Kafka.

- ❏ Both the Spark and Flink programs are written in Java and was allocated 1GB of JVM heap size.
- ❏ Macbook Pro 8GB RAM 2.9 GHz Intel Core i5 is used for running the performance analysis.
- ❏ The web application is developed using Plotly dash framework directly by consuming the data from the result topic stream in Kafka.

3.3 Sentiment Analysis

Sentiment analysis is the process of identifying and categorizing an emotion that is expressed in a piece of text such as positive, negative or neutral. In most of the use cases, sentiment analysis is performed on all the tweets in the stream. The stanford CoreNLP library is used for the predicting the sentiment value. Stanford CoreNLP is a pre-trained machine learning classifier which gives an integer output of -2 to +2 for the given input text. In our project, we have used a *sigmoid function* to rescale the sentiment value from 0 to 1. The output is interpreted as a range of values from 0 to 1 implying negative to positive.

4. Comparison of the stream processing systems using various use cases

This section presents various use cases and its implementation details of each use in both the systems. It also provides the performance analysis graphs and result of running for each use case.

4.1 Use case 1 : Pure Streaming

In the pure streaming use case, the sentiment of each tweet is analyzed individually and sent back to kafka. We make use of the Stanford CoreNLP library to calculate a sentiment score for each of the tweets. In spark, we used `MapToPair` function which maps each tweet in the incoming stream to a string containing tweet_id, tweet_text and sentiment value. In flink, we use `Map` function to do the same process.

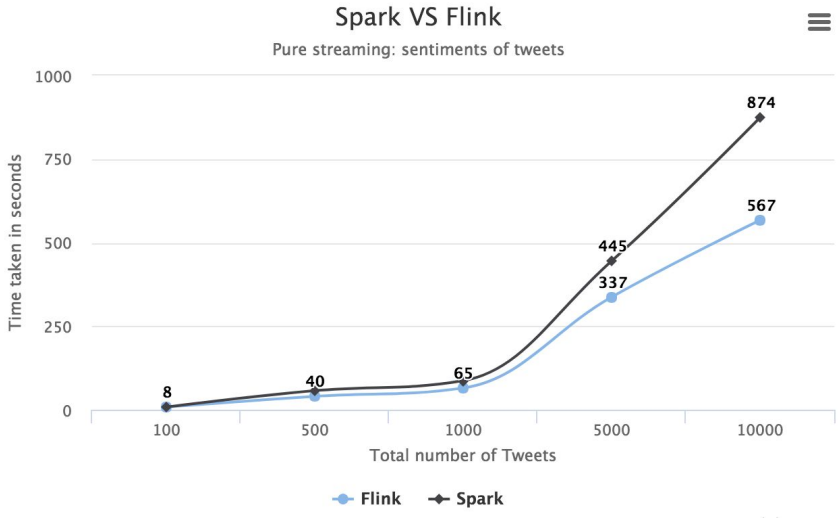

Figure 2: Performance of pure streaming in spark and flink

As we can see in Figure 2, Spark performed at a slightly lower rate than Flink for less than 1000 tweets but the time taken drastically increased as the number of tweets increased. In both the systems the time taken is very large as CoreNLP library taking a large amount of time in order to analyse every tweet.

4.2 Use case 2 : Windowed Streaming

In this use case, average sentiment of each state in USA is calculated from the tweets present in a given time window. A fixed windows of 5 seconds is used for this analysis. We first find the sentiment of the tweet and location from which it originated using the tweets location parameter. We set the location (i.e) US state name in our case - to be the key and then in each window, we group by the key and perform a window analysis where we find the average sentiment for each key in the window.

In Spark, we found the sentiment and state in `MapToPair` function. We then grouped by key and formed a window using `reduceByKeyAndWindow` and then find the average sentiment in the `map` function.

In Flink, we found the sentiment and state in the `Map` function. We then grouped by key using the `keyBy` function and then formed a window using the `window` function. Finally, we found the average sentiment in the `apply` function for the window.

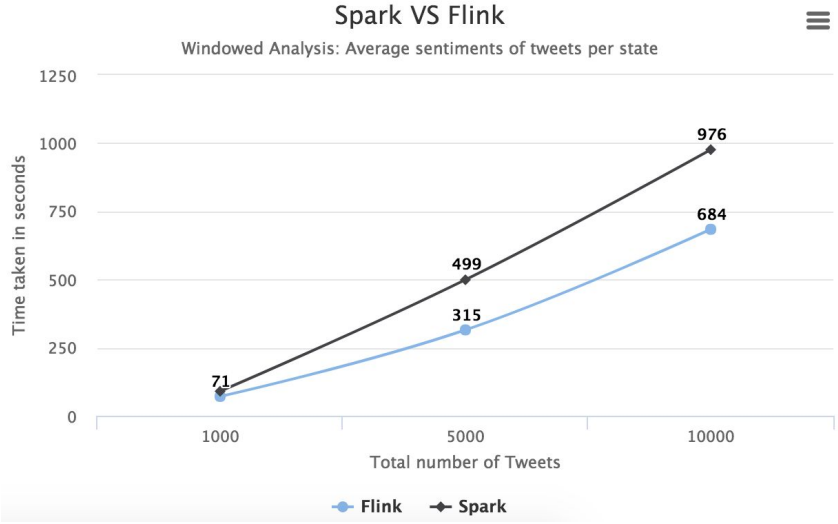

Figure 3: Performance of windowed streaming in spark and flink

As we can see from the figure 3, Flink is outperforming Spark from 1000 till 10,000 tweets by a reasonable margin. In both the cases, the transformations were pretty straightforward and easy to use and implement the use case.

4.3 Use case 3 - Windowed Streaming + Sorting

In twitter, the users can add a hashtag to their tweets to allow their Tweet to be categorized and easily searched in Twitter. For example #Spark to allow their Tweet to be categorized in a #Spark topic. The top most used hashtags at a given point of time are chosen to be trending topics on twitter. In this use case, we wish to find the top N trending topics that are found from the tweets present in a given time window. Sliding windows are used with a window length of 4 seconds with 2 seconds sliding frequency.

First we break the incoming tweets into set of words and then filter the words beginning with a hash (#) symbol. We then form a sliding window and count the frequency of each hashtag and sort it in the decreasing order. Then finally we take the top N words which will give us the current trending topics in our data stream.

In Spark the Tweets are split into words in a `flatMap` operation and then filter the words beginning with hash using `filter`. We then find the count of each hashtag and form a window using

`reduceByKeyAndWindow`. Here the word is the key that we use in the windows. Then we sort using `transformToPair`. The top N words are then finally extracted using `transform`.
In Flink, the Tweets are split into words in a `flatMap` operation. We then filter the words beginning with hashtag using `filter`. We then create a sliding window using `windowAll` operation. Finally, the `apply` function is used to find the count of each word and then the top N words are extracted using a priority queue as sort is not support on the streams in flink.

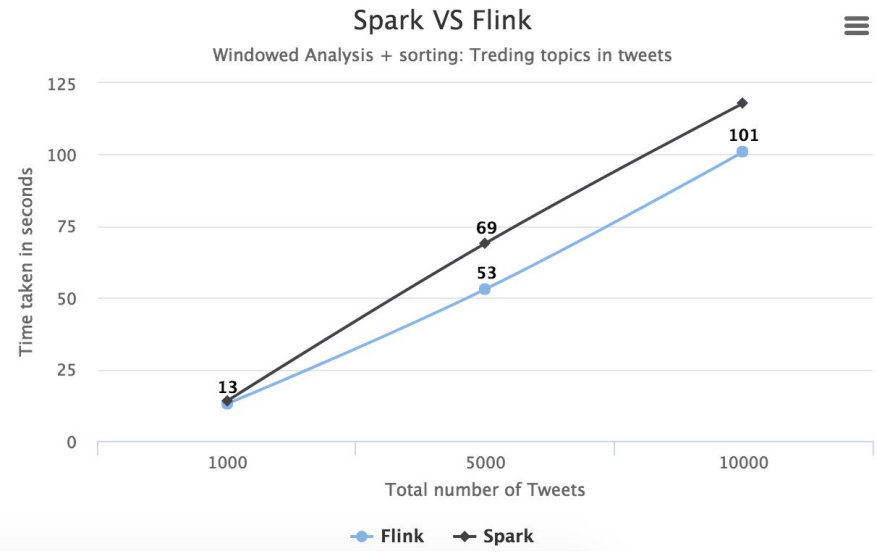

Figure 4: Performance of windowed streaming + sorting in spark and flink

As we can see from the figure 4, similar to the previous use case, we found Flink outperforming Spark from 1000 till 10,000 tweets by a reasonable margin. This use was very easy to implement in Spark as spark supports sorting operation because Spark operates on batches of data. But in flink, we had to write own sort operation using priority queues as Flink doesn't support sorting on streams. We can notice the time taken to process 10,000 tweets is very less compared to other use cases as the sentiment analysis operation is not involved in this use case.

4. Joining Two Windowed Streams

In this case, we made use of two different data streams - a twitter data stream and a weather data stream. The average sentiment and average temperature of each state in USA is calculated by joining the tweets and weather stream over a time window. We made use of a fixed window of 2 seconds.
We performed the two seperate windowed analysis on two data streams using the same set of operations as mentioned in the use case 2. On the resultant windowed data streams, we performed an `Equi-Join` operation in both Spark and Flink.
As it can be seen from figure 5, Flink out performs Spark similar to other use cases. And also we can see that the time taken by the Flink system to process 10,000 tweets along with join operation is similar to other use cases. But the time taken by Spark to process 10,000 tweets is increased due to the join operation in comparison with other use cases.

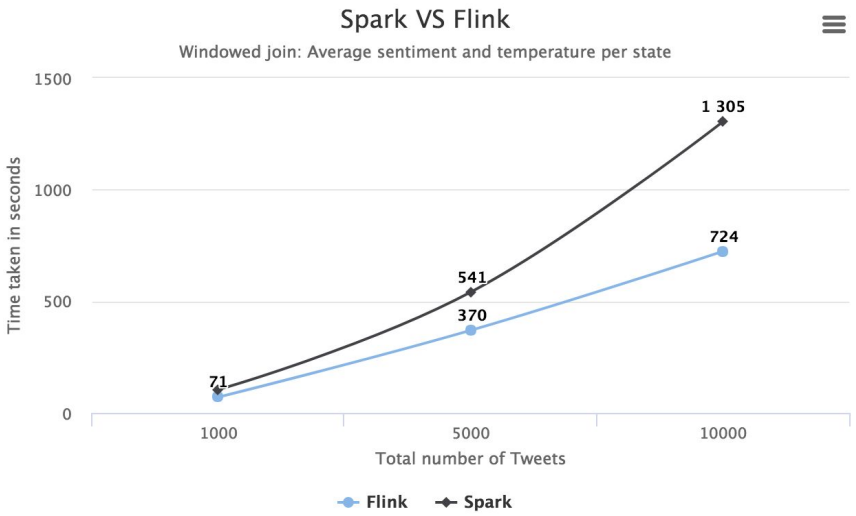

Figure 5: Performance of joining two windowed streaming spark and flink

5. Comparison results

This sections presents the results of the comparison of Apache Spark and Apache Flink based on the above use cases.

Category	Flink	Spark
Language support	Java, Scala	Java, Scala, Python, R
Community support	Flink has less user base but community support is good	Spark have large user base and community support is very good
Documentation	Flink documentation is very good. Easy to find features even without community support	Spark documentation is not good. Unless the community support is available, it is very difficult to find the features.
Performance	High due to data flow model	Less than flink due to batching overhead
Ease of use	Not many convenient features are developed. Users have to write their own code.	Many convenient wrappers are available for common functions like sort, outer joins.

Table 1: Comparison of Apache Spark and Apache Flink

6. Real time Geo-spatial data analysis

As a result of our comparison analysis, we chose flink to implement the real time geo-spatial analysis use case. (Similar to the use case 4). We developed an web application using Plotly dash. The user can input any search term. The application will stream the tweets based on the search term. The tweets stream and the weather stream is published into kafka. The flink performs a windowed join operation on the two joins as we mentioned in use case 4. The result of the window join operation on twitter stream and weather data streams will be in the form of *<State name, Average Sentiment, Average temperature>*. We developed a web application which listens to the kafka topic in which the flink writes the result of the windowed join operations. The UI updates for every 2 seconds which is same as the fixed window size used in the Flink's windowed join operation.

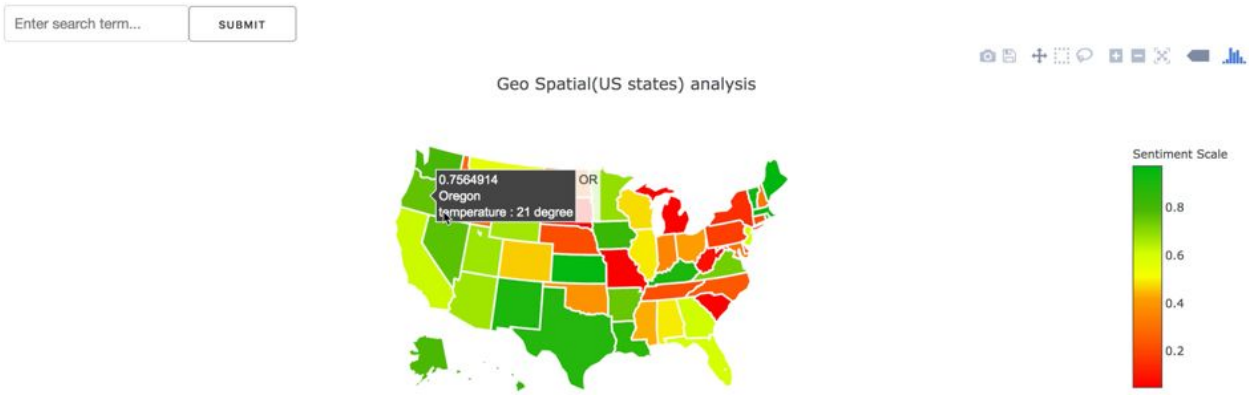

Figure 6: Screenshot of real time geo-spatial data analysis web application

7. Conclusion

This report presented a comparison of Apache Spark and Apache flink across various use cases. As a result, we inferred that both Spark and Flink are very good streaming systems with intuitive programming models and easy programming constructs. The comparison results will help the users to select a stream processing system based on their needs.

References

1. Spark: Cluster Computing with Working Sets
2. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing
3. Discretized Streams: Fault-Tolerant Streaming Computation at Scale
4. Apache Flink™: Stream and Batch Processing in a Single Engine
5. https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/datastream_api.html
6. https://spark.apache.org/docs/2.3.0/streaming-programming-guide.html
7. https://databricks.com/blog/2017/10/11/benchmarking-structured-streaming-on-databricks-runtime-against-state-of-the-art-streaming-systems.html
8. https://stanfordnlp.github.io/CoreNLP/api.html
9. https://dash.plot.ly/getting-started