The Sentiment Ranking Streaming Sample Application

The bestXStreamRating application was designed as a sample application to showcase the development of streaming applications with Apache Spark and Apache Flink. The application performs a live sentiment-analysis based ranking of tweets on terms of interest on the twitter stream in real time. Two implementations have been created that are as identical as the differences in the APIs of both of these frameworks allow and use common components whenever possible. This section describes the architecture and implementation of the application.

0.1 Application Architecture

An architectural overview of bestXStreamRating is given in figure 1. The terms that are subject of analysis are configurable using a simple external configuration file to keep the application open for ranking on any terms of interest. For each term a textual identifier, a display name, several synonyms and a picture can be specified in a csv-style configuration file. Each streaming application connects to the Twitter streaming API [9] using a framework-specific source function that handles the connection to twitter. Since the Twitter API expects a single connection per application the source functions were build as non-parallelized. It will register with Twitter to receive tweets that match the terms configured in the term configuration file. Each received tweet will match one of the configured terms and will be handed over to the computational workflow in Spark or Flink. Then the streaming application implemented in the respective framework-specific API will process each tweet in a parallelized workflow and among other things analyze the sentiment of each. It will then write out the results to a Redis message broker. This workflow is explained in detail in the next section. All of this processing will potentially run in parallel using the parallel processing capabilities of the respective framework. There will be a dedicated queue for each of the two streaming applications to be able to separate the results.

Decoupled from the previously explained components runs a web application which is itself decoupled in two components. A back-end web application cares for interacting with the Redis queues and makes the results available in a publishable format for incoming client requests from the browser client application.

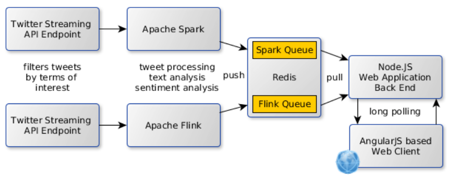

Twitter Streaming API Endpoint
Apache Spark
filters tweets by terms of interest
tweet processing text analysis sentiment analysis
push
Spark Queue
Redis
Flink Queue
pull
Node.JS Web Application Back End
long polling
Twitter Streaming API Endpoint
Apache Flink
AngularJS based Web Client

Figure 1: Architectural Overview

0.2 Streaming Job Workflow

The two streaming applications on Spark and Flink follow the same workflow of transformations to the extend possible by their APIs. A full workflow graph is shown in the appendix in figure 3 for Spark and figure 4 for Flink. In order to reuse as much of the processing logic as possible both implementations had been done in Scala, since Scala was declared to be a fully supported language for both frameworks.
For both versions the configuration has to be parsed and both Spark and Flink provide a method to read text files from local file system or HDFS. The actual parsing is done in a mapper and the result is collected to the driver program and then converted to a map. While Spark offers the convenience method `collectAsMap` for this rather frequent use case, the map conversion has to be done using the standard scala `toMap` in the Flink application.
So far the actual stream has not been initiated which is why the Twitter sources have to be invoked at this place as a basis for creating the data stream of tuples to act upon in the workflow. Both Spark and Flink provide a built-in Twitter source, the Spark one seems to work fine but only sampled the full Twitter stream and did not support filtering by certain terms which is essential in this project. However, the provided Source could be used with only slight modifications to the original program code. Additionally, a language filter has been added to filter for English tweets only. For Flink the provided Source did not work well and showed unpredictable behaviour several times during the development of this project why a new custom source had to be developed. It uses Twitter4J[1] to connect to the Twitter stream and applies the term and language filter in the same way as the modified Spark streaming source. These sources are not able to provide an exactly-once guarantee why the whole system cannot adhere to this semantics if seen as a whole.
On the data stream a series of transformations are executed identically on Spark and Flink. First two simple `map` functions extract the tweet post followed by a filtering of non-word and non-processable characters.
A `flatMap` transformation is then applied to do the actual sentiment analysis for each tweet. First the tweet has to be assigned to the term it actually matches. The matching is done by a simple presence check for the word in the tweet. In order to make the filter terms and their synonyms available from the driver in the `flatMap` operations a broadcast is used in Spark. For Flink broadcast cannot be used on arbitrary Scala types which is why it is added to the closure of the `flatMap` operation. Since in bestXStreamRating the list of terms is presumably very small anyway this is still tractable. If the list was larger Flink supports two ways of making side data available: either broadcasting of a static data stream or reading side data in a *rich* `flatMap` function's open hook. Since

[1]Twitter4J is an open source Java library for connecting to the Twitter API. Details can be found at http://twitter4j.org.

one tweet could presumable talk about more than one filter term a `flatMap` operation is used to emit one tuple for each matching filter term.
The actual sentiment analysis of the tweet is done at this point as well by using the trained sentiment analysis neural network from StanfordNLP[2] [2, 7]. The tweet is parsed into the StanfordNLP standard format and assigned a numerical sentiment value between -2 and 2 whereas higher represents a better sentiment. Since StanfordNLP's sentiment recognizer can only work for sentences the sentiment is only extracted from the longest sentence in the tweet in case there are multiple sentences. This is a considerable simplification suggested in many tutorials on tweet sentiment analysis [5, 6] which can be argued about, but at least some justification for it comes with Twitter's limitation to 140 characters per tweet. Furthermore, it should be emphasized again that sentiment analysis was not the actual focus of this work but just served as a use case for evaluating two different streaming systems.
The next step in the workflow is applying the windowing semantics which differs considerably in the API for Spark and Flink. While Spark does an implicit conversion from an RDD to a key/value RDD this has to be explicityly done in Flink with the `keyBy` transformation. The logical separation of processing by key is crucial here since processing should be done on tuples of the form

$$(term, (sentimentValue, tweetCount))$$

in logical separation by the term. Since flink supports many different semantics of windowing the desired has to be explicitly defined by either calling a predefined window function or providing a custom implementation. Since bestXStreamrating relies on time-based sliding windows the built in `timeWindow` function can be used. In Spark the former two operations can be omitted and instead a windowing reduce operations using `reduceByKeyAndWindow` can be used which applies a reduce function on a key/value stream using a sliding window. In Flink the reduce operation is called directly on the windowed stream created in the previous step. The reduce operation specified will simply sum up all the sentiment values and tweet counts by term and arrive at one tuple of the form

$$(term, (\sum sentimentValue, \sum tweetCount))$$

for every term of interest that occurred in a window. In a final map step the previously summed sentiment values are normalized by the respective tweet counts to compute the average sentiment for tweets of that term in the current window.
Lastly, the completely processed tuples can be handed over to the respective output operation. In the Spark implementation `foreachRDD` is used to perform that operation on each Mini-batch and then a map partitions function is applied to push the tuples within each partition to the Redis message queue. In contrast

[2]Optionally, the sentiment analysis can be performed in a simple way based on a sentiment dictionary instead of using the neural network. In this mode the sentiment of each term in a tweet is looked up in a sentiment dictionary then all values are added up to retrieve the sentiment rating for a sequence of words. A sentiment dictionary from [3] was used.

to that a rich map function can be directly applied as the output function in Flink. The open hook can be used to build up a Redis connection that can then be used throughout the lifetime of the mapper for pushing tuple-by-tuple to Redis in the pace they arrive at the mapper. The data is pushed to Redis in a defined schema using self-defined model classes that get serialized to JSON. This ensures easy deserializability from the JavaScript based aplication.

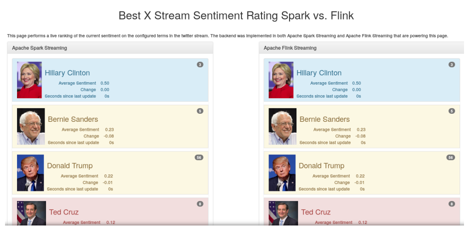
0.3 Web Application

While the previously presented components were actually sufficient for evaluating the stream processing capabilities of Spark and Flink some kind of easy accessible user interface was developed for easier visual comparison of the results. A natural way of doing that in such a project is through a web application which has also been developed for bestXStreamRating. This web application features a back-end component based on a Node.JS[3] [4] application and a browser client component implemented as AngularJS [1] application for creating a modern web user interface.

0.3.1 Web Application Back-End Component

The web application back-end runs two independent control flows: one cares for interaction with Redis and updates the internal state. And one serves the incoming client requests from the browser app. When a new result is available in the Redis queue the back-end web application component pulls it and stores it internally in memory. It keeps an in-memory priority queue holding at most one entry for each term of interest with the last two discovered sentiment values. One queue is kept for each of the two frameworks. Each of the entries is assigned a timestamp representing the time of receiving it from the Redis queue. This timestamp is used in exchange of tuples with the browser client for checking if the server's version is newer than that of the client.
When a request from the browser application is received the last version received by this client will be sent along with the GET request as URL parameter. If that version is older than the current version stored in the internal state, the internal state will be sent to the client as is. Otherwise, the request will be kept open for a certain period to wait for a new state to become available. Either a state becomes available within the waiting time frame, in which case it is returned to the client, or after a certain deadline the request is returned with the not modified HTTP status without any payload. Given the assumption that there will always be enough tweets available for analysis, the latter should never happen as long as the streaming applications in Spark or Flink don't fall behind in processing of tuples. Data is exchanged between the back-end and browser component in the JSON format and a separate HTTP endpoint is provided for Spark and Flink, respectively.

[3]Node.JS is a standalone runtime for JavaScript and can be used for server-sided use of JavaScript [4].



Best X Stream Sentiment Rating Spark vs. Flink

This page performs a live ranking of the current sentiment on the configured terms in the twitter stream. The backend was implemented in both Apache Spark Streaming and Apache Flink Streaming that are powering this page.

Apache Spark Streaming

Hillary Clinton 3
Average Sentiment 0.50
Change 0.00
Seconds since last update 0s

Bernie Sanders 5
Average Sentiment 0.23
Change -0.08
Seconds since last update 0s

Donald Trump 56
Average Sentiment 0.22
Change -0.01
Seconds since last update 0s

Ted Cruz 8
Average Sentiment 0.12

Apache Flink Streaming

Hillary Clinton 3
Average Sentiment 0.50
Change 0.00
Seconds since last update 0s

Bernie Sanders 5
Average Sentiment 0.23
Change -0.08
Seconds since last update 0s

Donald Trump 56
Average Sentiment 0.22
Change -0.01
Seconds since last update 0s

Ted Cruz 8
Average Sentiment 0.12

Figure 2: The bestXStreamRating Web Application

0.3.2 Web Application Client Component

The client has been implemented as a one page web application based on AngularJS and follows the model view controller pattern. The controller periodically opens a polling request to the back-end application to ask for new data. Two independent pollers exist to poll for the result of either Flink or Spark. Once new data is returned the model is updated and the view refreshed. The view uses the Bootstrap [8] UI framework to provide a state of the art user interface. The web application can be seen in figure 2 rating tweets on the candidates for the US presidential elections 2016.

For each term the number of analyzed tweets is displayed in the upper right corner in the grey circle. The term that had the best average sentiment in the currently analyzed window is shown first followed by the term with the second best average sentiment and so on. A background color visualizes the sentiment value from negative to positive in either red, yellow, blue or green. If a link to an image representing the term has been provided in the configuration file it is presented next to the display name.

Appendix

The following shows the complete workflow graphs for both the Spark and Flink streaming application.

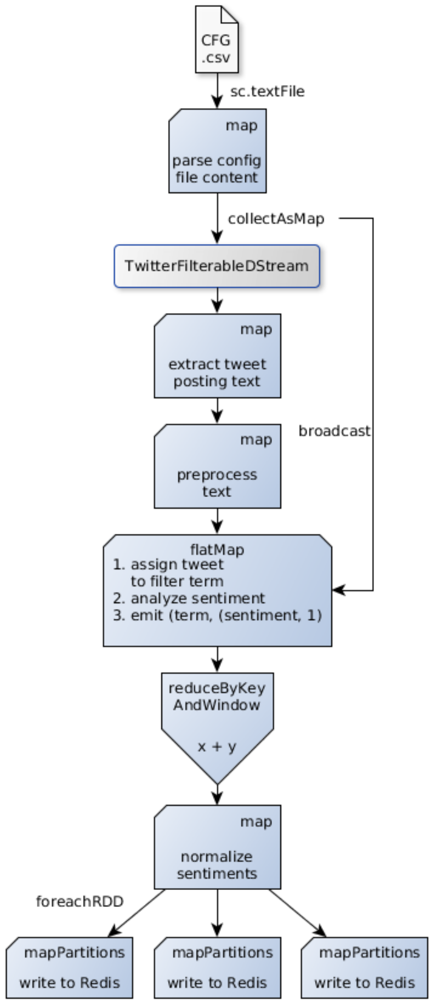


Figure 3: bestXStreamRatingSpark - Workflow

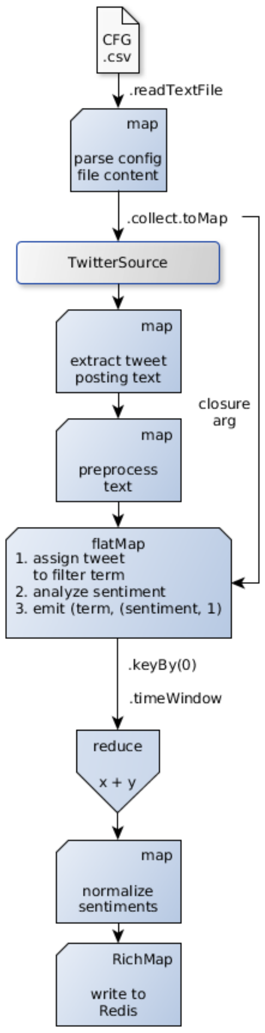


Figure 4: bestXStreamRatingFlink - Workflow

References

[1] Google. AngularJS - Superheroic JavaScript MVW Framework, 2016. [Computer Software, Online; accessed April 16, 2016; Available at https://angularjs.org/].

[2] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[3] F. Å. Nielsen. AFINN, 2011. [Online; accessed April 17, 2016; Available at http://www2.imm.dtu.dk/pubdb/p.php?601].

[4] Node.js Foundation. Node.js, 2016. [Computer Software, Online; accessed April 16, 2016; Available at https://nodejs.org].

[5] Rahul AR. Twitter sentiment analysis in less than 100 lines of code!, 2014. [Online; accessed April 16, 2016; Available at http://rahular.com/twitter-sentiment-analysis].

[6] Shekhar Gulati. Day 20: Stanford corenlp performing sentiment analysis of twitter using java, 2013. [Online; accessed April 16, 2016; Available at https://blog.openshift.com/day-20-stanford- corenlp–sentiment-analysis-of-twitter-using-java].

[7] R. Socher, A. Perelygin, J. Y. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the conference on empirical methods in natural language processing (EMNLP)*, volume 1631, page 1642. Citeseer, 2013.

[8] Twitter Inc. Bootstrap - The world's most popular mobile-first and responsive front-end framework, 2016. [Computer Software, Online; accessed April 16, 2016; Available at http://getbootstrap.com/].

[9] Twitter Inc. The Streaming APIs - Twitter Developers, 2016. [Online; accessed April 17, 2016; Available at https://dev.twitter.com/streaming/overview].