Twitter Streaming with Apache Flink

Siqi Liu  
 Computer Science  
 University of Waterloo  
 Waterloo Ontario Canada  
 sq2liu@uwaterloo.ca

Zibai Wang  
 Computer Science  
 University of Waterloo  
Waterloo Ontario Canada  
 zibai.wang@uwaterloo.ca

ABSTRACT

Our project is to implement an Apache Flink application in Java that streams data from Twitter, transports the data into a data storage, and presents real-time analytical visualizations such as hashtag count, geographic heatmap and retweet percentage.

**KEYWORDS**

Flink, Twitter, Streaming, Elasticsearch, Kibana.

**LEARNING OBJECTIVES**

1. To understand the fundamental concepts and challenges associated with stream processing, and how it’s different from traditional batch processing
2. Explore the Apache Flink framework, including its architecture, environment, and the development process of a Flink DataStream application
3. Learn how to set up a real-time analytics dashboard using Elasticsearch and Kibana

**BACKGROUND**

1. Stream processing is a type of real-time data processing that is designed with infinite data sets in mind (ref medium article)
2. While Spark has certain streaming capabilities using micro-batches, Flink is capable of native-streaming – processing every record as soon as it arrives without waiting for others
3. Why we chose Flink for stream processing: it is the leading innovation in the data streaming community, the first truly native-streaming framework with advanced features like event time processing. It also ensures that each record is processed exactly once, low latency with high throughput, all of which are ideal traits of a streaming framework

**METHODOLOGY**

Set-up Apache Flink and Twitter Connector

Setting up Apache Flink locally was straightforward – in MacOS X, just needed to install the latest version using HomeBrew. Using the Apache Flink archetype in Maven, we were able to quickly set up our project. We then added several necessary dependencies in the pom.xml file – Twitter connector, Elasticsearch connector, Elasticsearch client, and log4j

To access data from Twitter, we had to apply for a developer account. After the application was approved, we obtained our keys and tokens that would be passed into our Flink application to allow for Twitter authentication.

Window, Parallelism and EndPoint

2.1 Window

To be added

2.2 Parallelism

To be added

2.3 EndPoint

An *EndPoint* is how we control which tweets are being streamed into our program. For example, if we are only interested in tracking tweets sent from North America, or in English, it would be highly inefficient to have to stream every tweet from the world and then rely on our application to filter out all the undesired tweets. Instead, we could rely on EndPoint to do a first line of filtering.

We needed to select an appropriate endpoint provided by Twitter for our program. After researching about a few endpoints, including *StatusesSampleEndpoint*, *StatusesFilterEndpoint*, and *SitestreamEndpoint*, we decided to use *StatusesFilterEndpoint* because it best suited in our application.

What’s in a Tweet?

We referred to the API documentation on Twitter Developer’s website (ref) for the format of the tweet objects that were sent through the Twitter connector. In essence, each tweet is given as a JSON object, with fundamental attributes such as *id*, *created*\_at. Some attributes are JSON objects themselves – for example, *user* is a JSON object consists of *name*, *location*, etc. In total, there are over 150 different attributes in a single tweet. We realized that for the purpose of our analysis, we don’t need the mass majority of the attributes, and if we were to keep the original tweet JSONs and parse through them every time we want to extract relevant attributes, it would take up too much memory space and runtime. Therefore, we decided to create a single flatmap process that parses through each tweet JSON exactly once when it is streamed, and construct a Tweet object, keeping only the attributes that we need for analysis in the future. For example, the Tweet object stores *created\_at* as a timestamp, *coordinates* as a tuple of floats, and *retweet* as a Boolean variable.  
A screenshot of a cell phone

Description automatically generated

We recognize a few pros and cons associated with this approach. **Pros:**

1. Each tweet JSON is parsed through only once, eliminating repetitive work, saving memory space and runtime
2. Tweet object are much cleaner to work with

**Cons:**

1. If the JSON format changes (e.g. attributes are removed), creating Tweet object could encounter errors, interrupting all of downstream tasks instead of just the task that’s affected by the attribute removal
2. If our analysis requires more attributes in the future, we would need to store more attributes, diminishing the memory and runtime benefit

Once we have set up the DataStream for the Tweets, we are ready to finally start our analysis.

Analysis

4.1 Hashtag Count

A hashtag is a type of metadata tag used in Twitter that allows users to apply user-generated tagging which makes it possible for others to easily find messages with a specific theme or content. Hashtags are distinguishable by a single leading number sign #. When a person tweets, he/she can choose to include one, or multiple hashtags in his/her tweet.

In the tweet JSON, hashtags are stored in the list format. Originally when creating the Tweet object, we stored this list as-is in the *hashtag* attribute. Later on, we realized two essential facts about hashtags that we did not consider:

1. Some users like to include the same hashtag multiple times in a given tweet
2. Hashtags are case-sensitive

From an end-user analytics perspective, we decided that each hashtag should really be considered only once in a single tweet, and that they should be case-insensitive. Therefore, we modified our parsing logic and flatmap logic to make hashtags distinct and all lower-cased.

Recognizing the fact that not all tweets have hashtags, we created a custom flatmap function that, given a Tweet object, iterates through its *hashtag* list and emits tuples of *(hashtag, 1)* to the collector.



For aggregation, we applied a simple *keyBy 🡪 tumbling window 🡪 sum* logic that sums up all occurrences of a given hashtag, within the specific tumbling window period.



The end result is a stream of tuples of *(hashtag, count)*.

4.2 Geographic Map

When you send a tweet on your phone, if you had your location service enabled, your tweet would contain the exact GPS coordinate of the location you sent the tweet. Or, if you choose to tag a specific place (e.g. Dana Porter Library), a bounding box be included in your tweet. To capture this information, when we parse through the tweet JSON, we either store the location in which the tweet was sent (if available), or simplify the bounding box information by taking the average of the corner values, i.e. finding the midpoint, and store it in a *coordinate* attribute as a tuple of floats representing its latitude and longitude.

Similar to hashtags, not all tweets have locations associated with them, so we created another flatmap that emits tuples of *(<latitude, longitude>, 1)*. Later on, we realized that Elasticsearch actually reverses the order in which geographic data points are stored (i.e. longitude before latitude), so we had to our sinking function to reverse the coordinates. A screenshot of a cell phone

Description automatically generated

We applied the same aggregation logic as in Trending Hashtags. The end result is a stream of tuples of *(<latitude, longitude>, count)*.

4.3 Tweet Type Time-Series

Each tweet is either an original tweet, or a retweet. Sometimes it’s useful to see how the tweet/retweet composition evolves overtime, given a particular hashtag.

We can identify if a tweet is a retweet or not by looking at its *retweet\_status* attribute – if it is not null, then it’s a retweet, otherwise it’s an original tweet.

Once again, we created a flatmap function that emits tuples of *(“Tweet”, 1)* or *(“Retweet”, 1)*.



In order to create a time-series, we also need to store the time. This was not entirely trivially, and we had to create a custom *process* function that, not only sums up all instances of Tweet/Retweet in a given window period, also appends the start time of the window as an additional field.



The end result is a stream of tuple of *(Tweet/Retweet, window\_start\_time, count).*

Elasticsearch, Logstash and Kibana

5.1 Overview

Elasticsearch, Logstash and Kibana (also known as ELK) is a suite of real-time data storage, analytics and visualization tools. In essence, data (in JSON formats) are stored in Elasticsearch as *documents*, which then get grouped into different *indices*. For example, a hashtag could be a single document in an index called *hashtag\_index*. Kibana then allows you to visualize your data in these indices.

Flink’s DataStream API offers a connector to Elasticsearch. This connector provides *sinks* that can request different actions on documents/indices, such as creating, updating and deleting.

At first, we installed the latest version (v7.4) of the ELK stack onto our local computers. Then, as we were developing the sink functions from Flink to Elasticsearch, we encountered several compatibility issues between the version we had (v7.4), and the version that Flink uses for its connector (v6.8), so we had to roll-back the ELK stack to v6.8 as well.

5.2 Index

As mentioned previously, an index is a collection of documents. The easiest way to define the metadata for an index is to do it in Elasticsearch’s built-in terminal (or curl into it using bash):



We can also delete an index in the same fashion:



* 1. Transport & Sink

For Flink to communicate with an Elasticsearch cluster, it uses a type of transport client called the *RestHighClient*.

Initializing the transport client is straightforward:

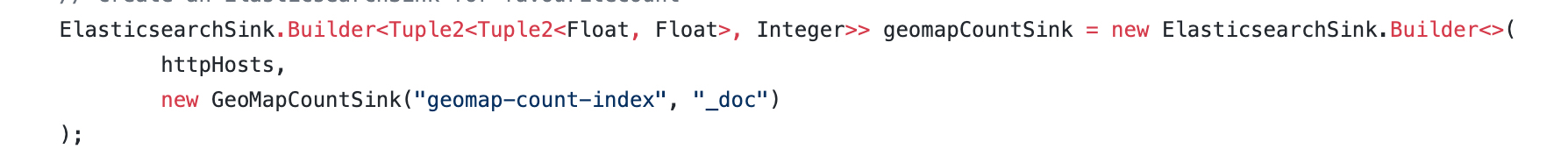


Once initialized, every time we want to perform an index change (e.g. add a document), the transport client would submit an index request to the Elasticsearch cluster.



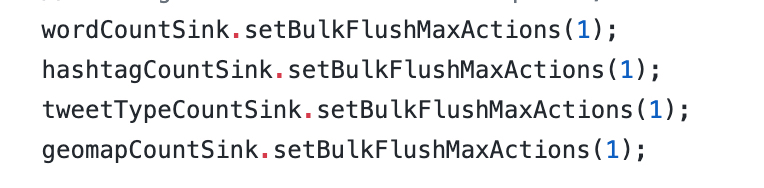
*Sink functions* are what’s between the Flink DataStream and the transport client. In essence, they are used to transform the data from the DataStream into the JSON format that Elasticsearch requires and tell the client to which index it should submit the update request to.



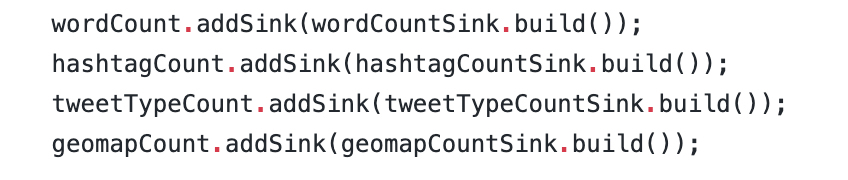


Each task would have its own sink function. They are all fairly similar, except for Tweet Type Time-Series because it also needs to include a timestamp field.

We want the data to be streamed to Elasticsearch as soon as it’s ready (i.e. at the end of the tumbling time window), so we set *bulk.flush.max.action* to 1:

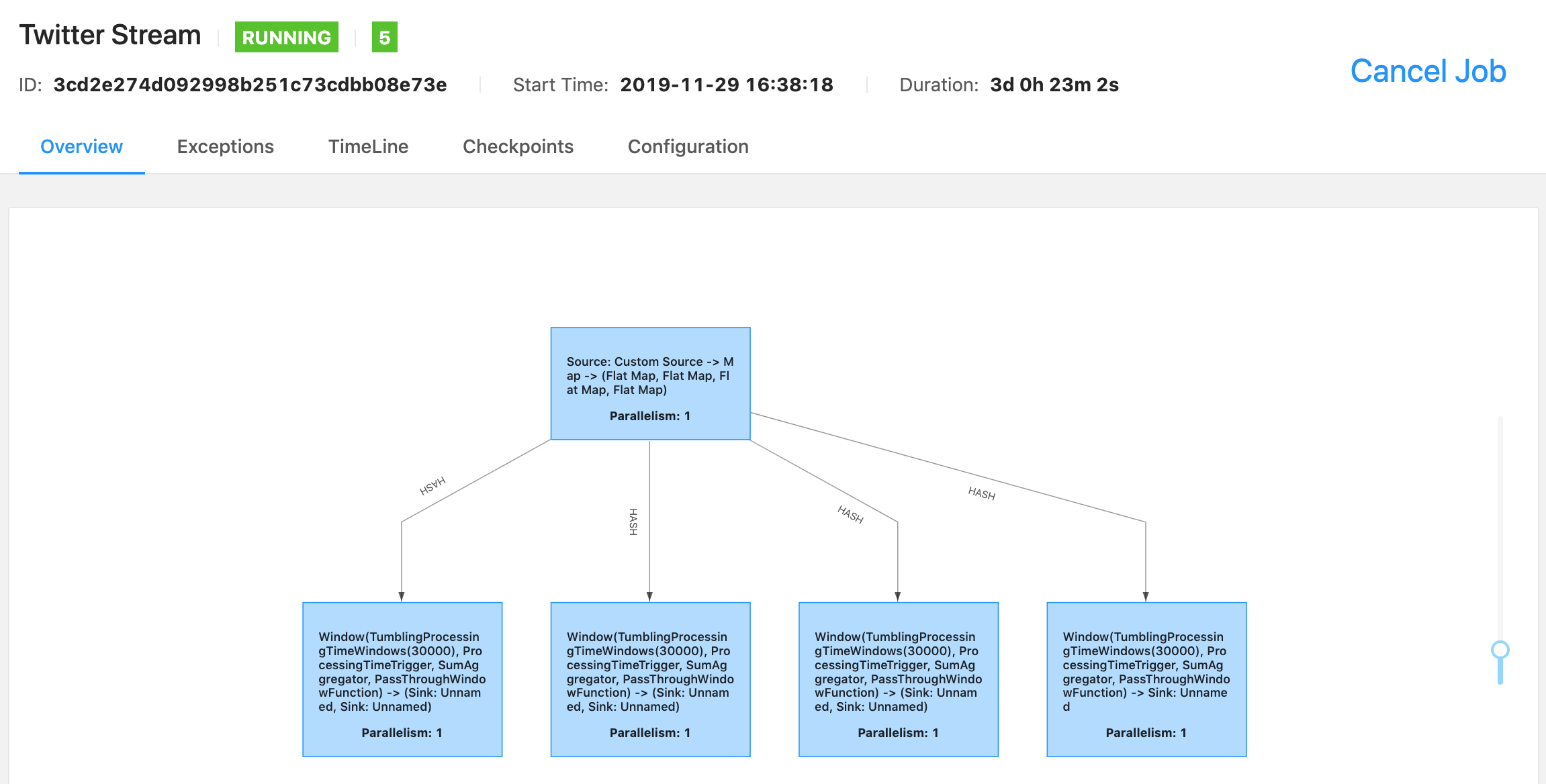


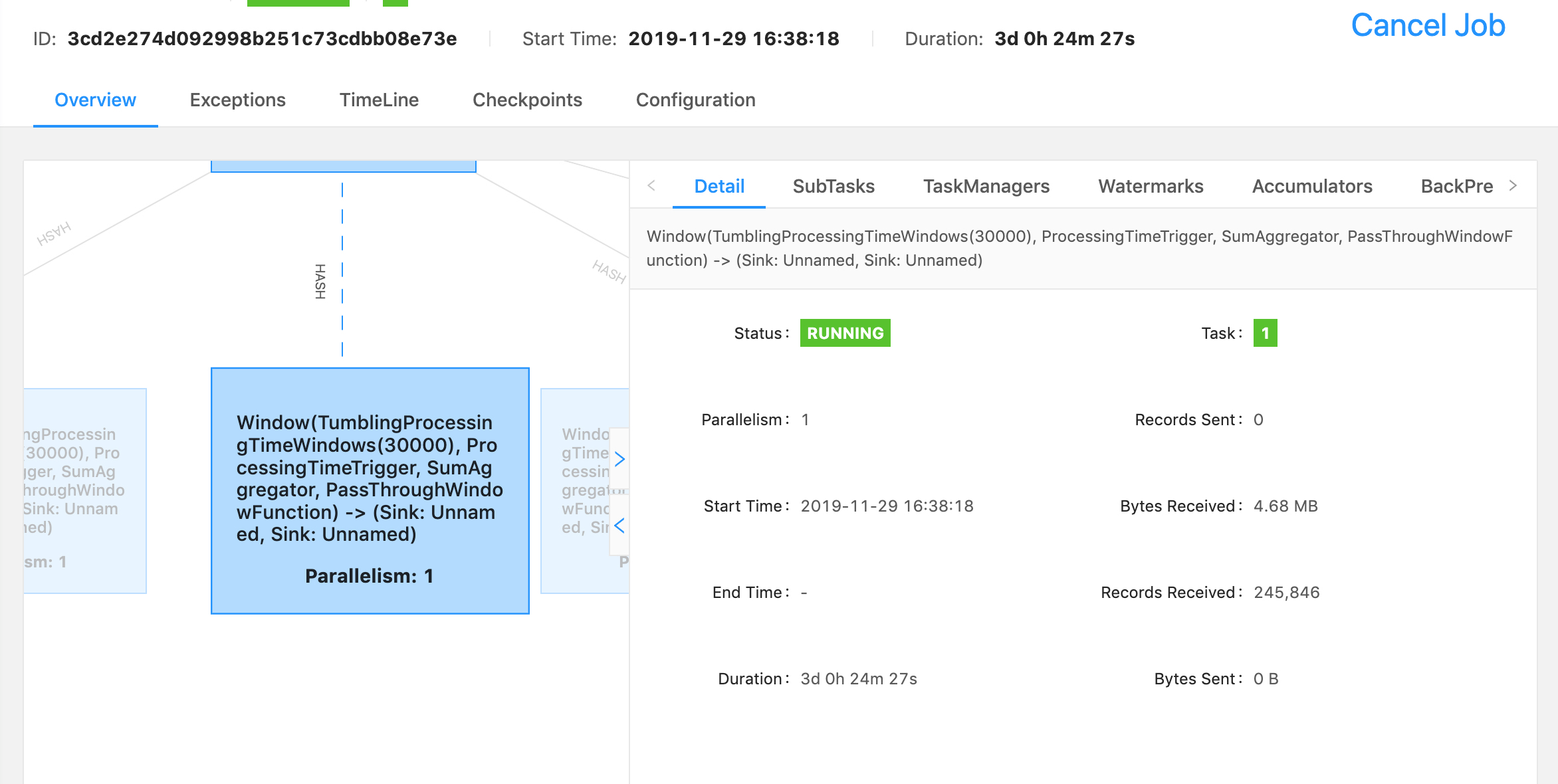
Lastly, we attached the sinks to the appropriate DataStream.



5.4 Monitor Stream

Flink offers a great dashboard to visualize the data flow from one DataStream to another.





Here, you can see how many tweets are processed by the first DataStream job (which parses through each tweet JSON and creates a Tweet object), and the subsequent DataStream jobs.

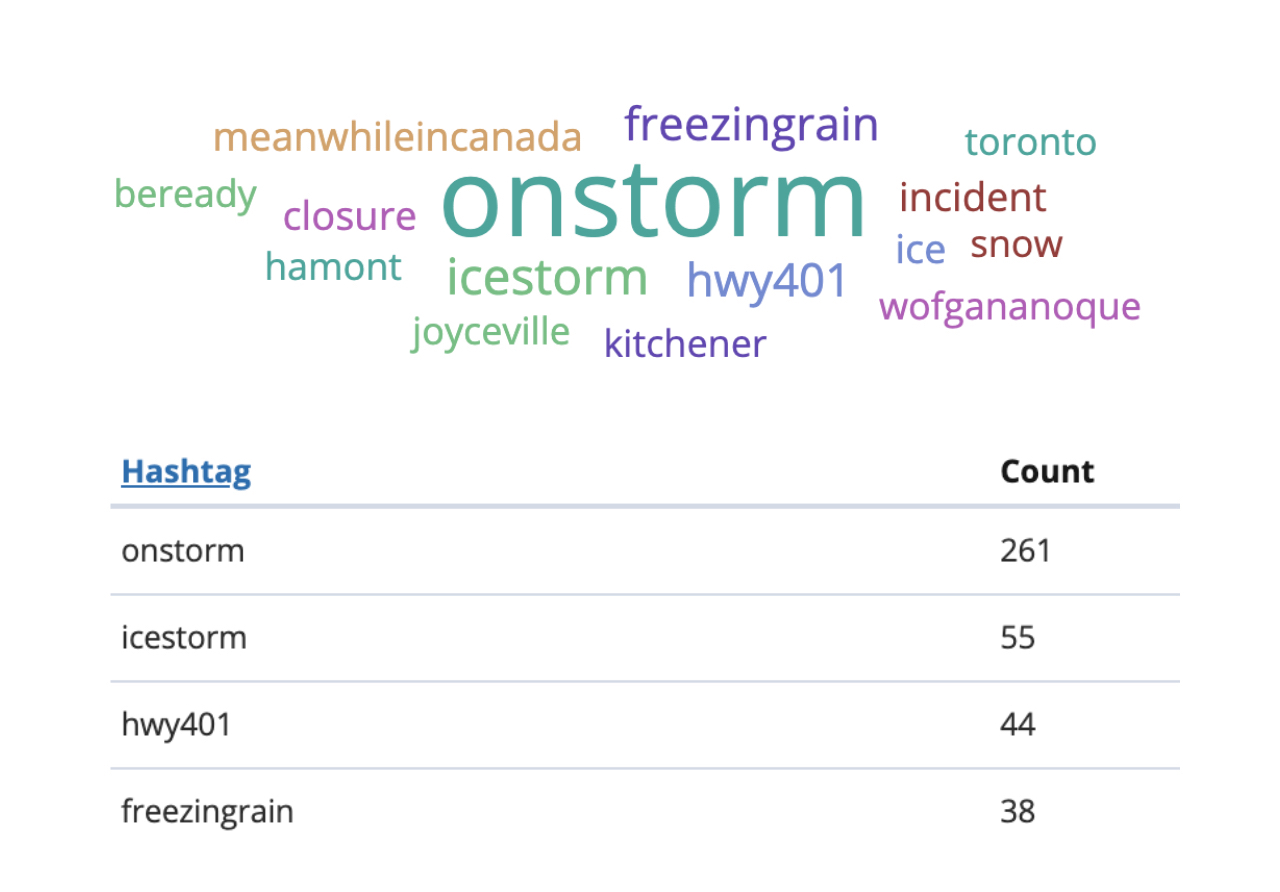
5.5 Visualization

The following section contains several screenshots of the visualizations in Kibana that were updated in real-time using the indices in Elasticsearch.

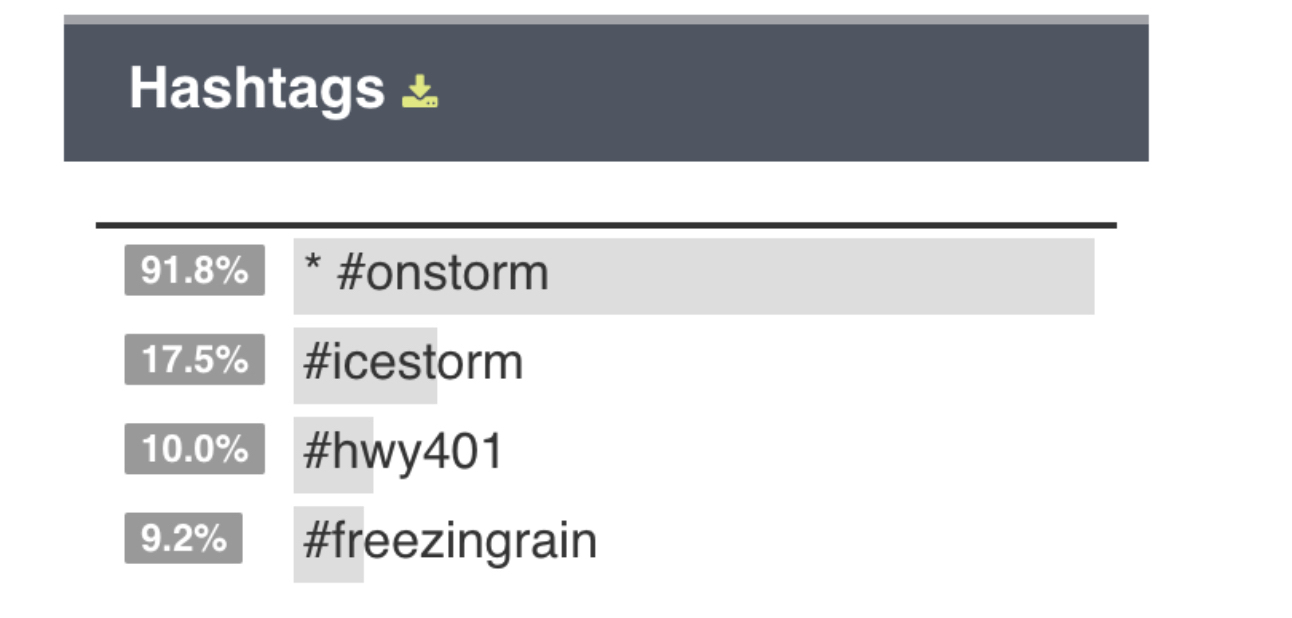
EVALUATION

We wanted to make sure that the data we received and the indices we produced were correct. To do so, we streamed for one hour, from 3:30 pm to 4:30, on December 1st, tracking the hashtag #ONStorm, and compared our results against the same query from a third-party Twitter analytics website [ref].

6.1 Hashtag Count

Below are some of the top hashtags produced by our application:  


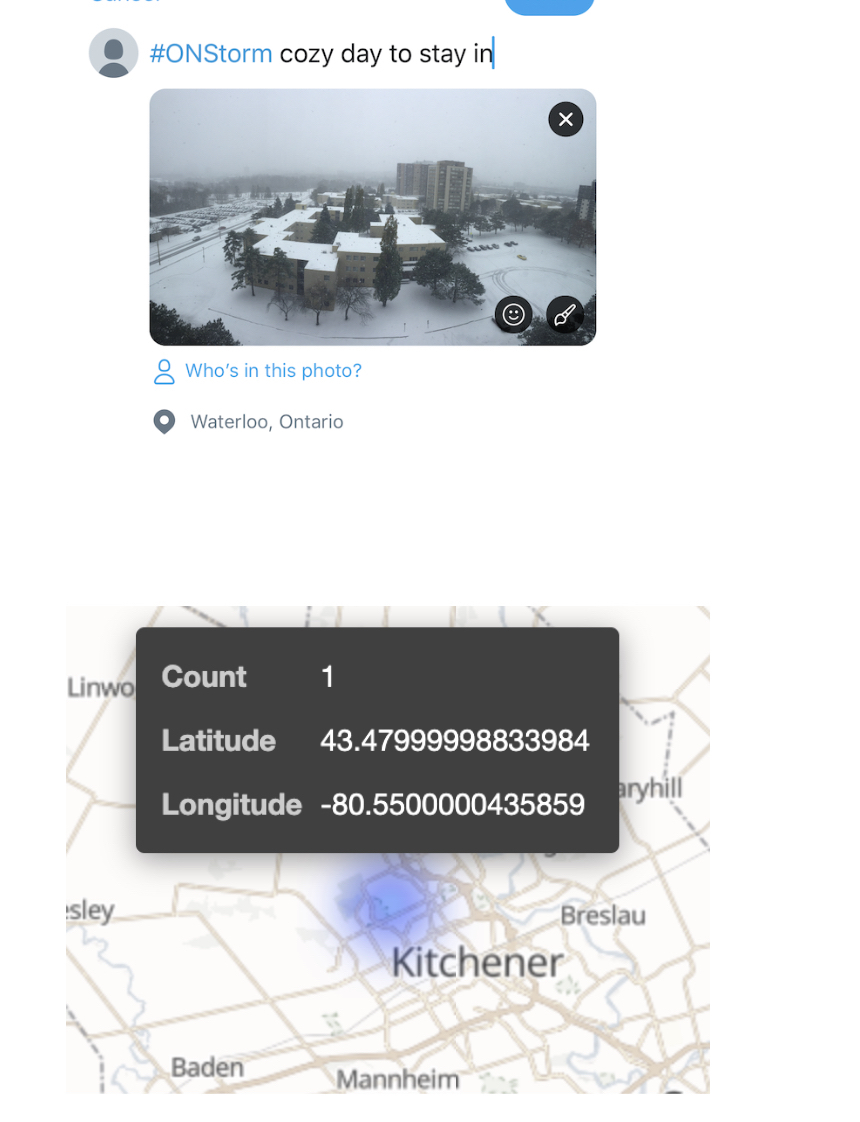
From the third-party analytics website:



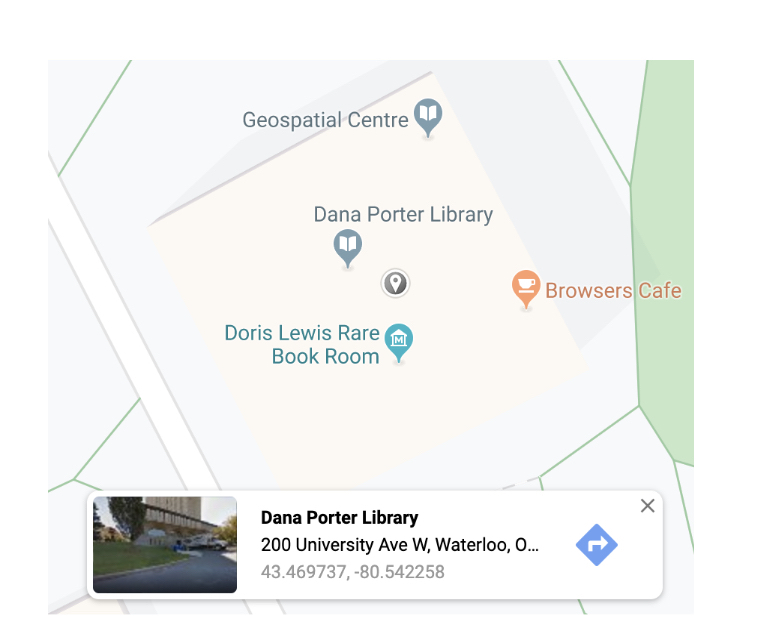
The results did match up quite well. The other hashtags also made sense (#meanwhileincanada).

6.2 Geographic Heat Map

We sent a tweet from Dana Porter Library, with location service enabled. Here it is on our dashboard:

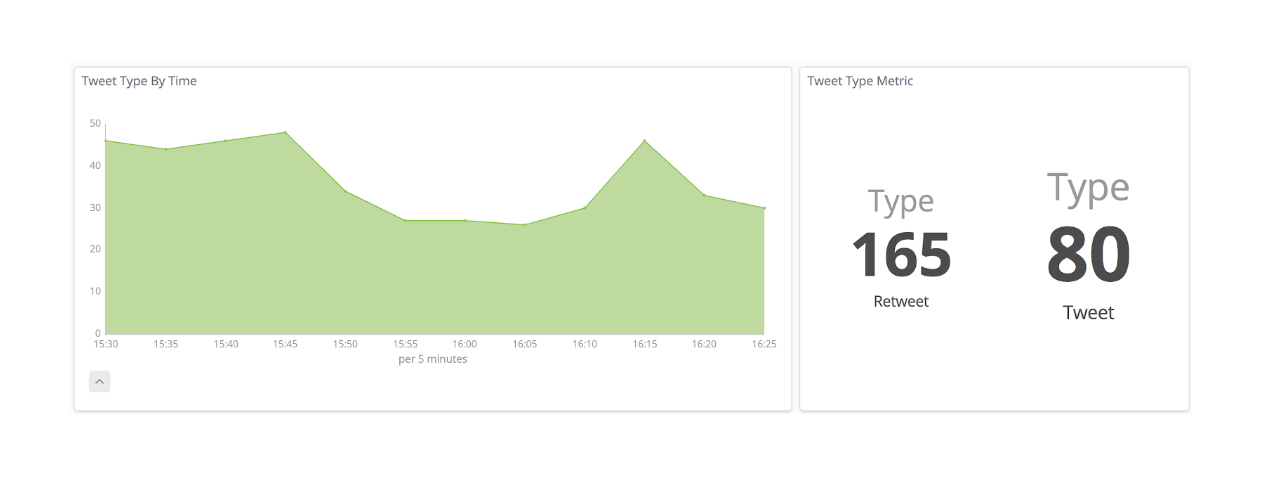


Based on Google Maps, the GPS coordinate was spot-on:

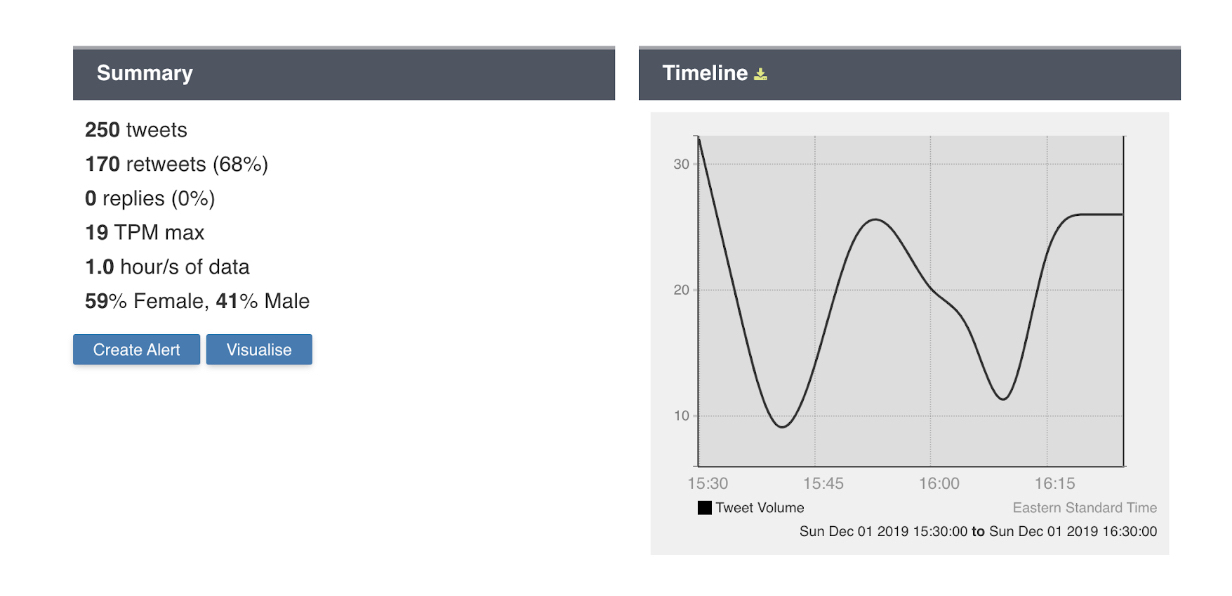


Tweet Type Time-Series

The graph on the left shows the tweet count with both tweet and retweets combined, since the third-party website does not provide such breakdown. However, the number on the right shows the total breakdown over the one-hour period:



From the third-party analytics website:



Our results, for both tweet and retweet match up quite nicely. There are some small levels of discrepancies, which we suspect is due to connection issues with the Twitter API.

Lessons Learned

Through this project, we have learned a great deal about stream processing, and certainly more about Apache Flink. However, besides all the technical knowledge, there are a few other valuable lessons that we would like to highlight:

1. Always assume there will be missing data.  
   Some attributes in the Tweet JSON are Nullable and our code needed to handle it. Lack of handling Nullable data has caused some trouble for us
2. Always use the correct version, and read the correct versions’ documentation. Initially, we installed the latest Elasticsearch version (7.4.2) although we did see the Flink Elasticsearch Connector was only tested on ES version 6. This mistake resulted in a few issues, for example, the Mapping process changed dramatically from v6 to v7 which caused Flink Elasticsearch Connector no longer working with ES 7. After we changed our ES version to v6, things started to run smoothly.
3. End-user - do we care about #Trump vs #trump? Need to use lower cases. User inputs should be preprocessed in order to produce a more accurate result for End-user.
4. Trade-off between WindowSize and size of the index in ES
5. Before diving into the task, understand if the result will be meaningful
6. WordCount - doesn’t really mean much since a lot are filler words. Unless we can filter out, the result doesn’t matter
7. FavouriteCount - doesn’t make much sense either since normally the tweet will not be liked by someone else in   
   such a short window.
8. Irregularities in data (emojis, pictures, user mentions)

Future Improvements

In the end, we want to also highlight some of the areas of improvements for similar projects in the future

1. Dockerize: since all the frameworks (Flink, Elasticsearch, Kibana) that we used in our project can be dockerized, we can build our project into a docker app and publish it on the docker hub. As Docker can be used in a wide variety of Operating Systems, any user who is interested in using our project can pull it from the docker hub and get it up and running in minutes, without the need to install each component separated.
2. Set up parallel tasks to handle each task separately
3. See if we can flush indices periodically to save space in ES

REFERENCES

[1] <https://learning.oreilly.com/library/view/stream-processing-with/9781491974285/>

[2] <https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b>

[3] <https://ci.apache.org/projects/flink/flink-docs-stable/dev/connectors/twitter.html>

[4] [https://ci.apache.org/projects/flink/flink-docs-stable/dev/connectors/Elasticsearch.html](https://ci.apache.org/projects/flink/flink-docs-stable/dev/connectors/elasticsearch.html)

[5] <https://www.tutorialspoint.com/apache_flink/apache_flink_batch_realtime_processing.htm>

Conference Name:ACM Woodstock conference

Conference Short Name:WOODSTOCK’18

Conference Location:El Paso, Texas USA

ISBN:978-1-4503-0000-0/18/06

Year:2018

Date:June

Copyright Year:2018

Copyright Statement:rightsretained

DOI:10.1145/1234567890

RRH: F. Surname et al.

Price:$15.00