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 aims to implement an Apache Flink application in Java that streams data from Twitter, transports the data into Elasticsearch, and presents real-time analytical visualizations using Kibana.

KEYWORDS

Flink, Twitter, Streaming, Elasticsearch, Kibana.

LEARNING OBJECTIVES

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

BACKGROUND

Stream processing is a type of real-time data processing that is designed with infinite data sets in mind.

While Spark has certain streaming capabilities using micro-batches, it does not support native-streaming, i.e. it cannot process records as soon as they arrive.

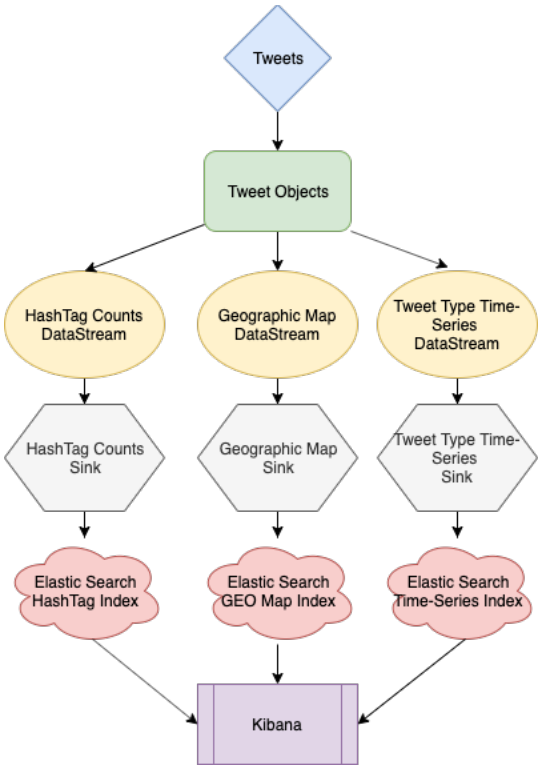
We wanted to learn Flink because it is the leading innovation in the data streaming community. It is the first streaming framework that is capable with native-streaming, with advanced features such as event time processing. It has very low latency and high throughout. It also ensures that each record is processed exactly once, an ideal trait of a streaming framework.

Since Twitter offers an abundant amount of streaming data, and Flink has a well-maintained Twitter connector API, we decided to use Twitter as our data source.

We wanted to present our data and analysis visually, so we chose to use Elasticsearch and Kibana as our data storage and visualization platforms since they have nice synergies with the Flink workflow.

APPLICATION OVERVIEW

The flow chart below illustrates how our application works:



First, tweets are streamed from the Twitter API to Flink through the Twitter connector. Flink would then process each tweet and store it in a Tweet object. Next, Flink creates three separate data streams that conducts three distinct data mining algorithms in parallel. Each data stream then sinks the results into an Elasticsearch index. At last, Kibana creates three visualizations based on these indices.

SET-UP

1 Apache Flink and Twitter Connector

Setting up Apache Flink 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 establish a framework for 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.

2 Window and EndPoint

2.1 Window

Since streams are continuous, we needed to split the streams into different "buckets" of finite size in order to do computation (e.g. aggregation).

Flink supports two types of windows – *keyed* and *non-keyed*. The only difference between the two is that, with keyed, one must first define a key field in the data using the *keyBy* function. For example, for one of our tasks in counting hashtags, the key would be hashtag, and the value would be the count.

The advantage of having a keyed stream is that, similar to partitioning in MapReduce, windowed computation can now be done in parallel by multiple task nodes. This is what makes it a distributed streaming system.

Another thing we had to decide on is the type of window assigners. Similar to Spark streaming, Flink supports both *sliding window* and *tumbling window*. This is entirely situational, and for our application and the specific tasks that we implemented, we decided to go with tumbling window for all data streams.

2.2 EndPoint

An *EndPoint* from the Twitter API is used to control which tweets are being streamed into our program. For example, if we are only interested in tweets that are sent within North America, it would be highly inefficient to have to receive every single tweet from all around the world and then rely on our mapping process to filter out all the undesired tweets. Instead, we could use EndPoint as a first line of filtering.

We needed to select an appropriate EndPoint for our application. After researching about a few different endpoints, we decided to use *StatusesFilterEndpoint* because it best suited our application.

Specifically, our application allows the end user to specify if they want to receive tweets that are:

1. Sent within North America
2. In specific languages
3. Contain certain terms/hashtags/user mentions

```
public StreamingEndpoint createEndpoint() {
    StatusesFilterEndpoint endpoint = new StatusesFilterEndpoint();

    if (this.onlyNA) {
        endpoint.locations(Collections.singletonList(
                new Location(
                        // North America
                        new Location.Coordinate(-168.48, 13.23),
                        new Location.Coordinate(-50.36, 72.76))
                )
        );
    }

    if (this.languages.size() > 0) {
        endpoint.languages(this.languages);
    }

    if (this.trackTerms.size() > 0) {
        endpoint.trackTerms(this.trackTerms);
    }

    endpoint.stallWarnings(false);
    endpoint.delimited(false);

    return endpoint;
}
```

3 What's in a Tweet?

We referred to the API documentation on Twitter Developer's website for the format of the tweet objects that are 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 didn't need majority of the attributes, and if we were to keep the original tweet JSONs and parse through them every time we wanted 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 needed for analysis in the future.

As such, the Tweet object stores *created_at* as a timestamp, *coordinates* as a tuple of floats, and *retweet* as a Boolean variable.

```java
public class Tweet {

    private String timestamp;
    private long id;
    private String text;
    private boolean retweet;
    private long user_id;
    private List<String> hashtags;
    private Tuple2<Float, Float> coordinates;
    private int favorite_count;

    public Tweet(JsonNode jsonNode) {
        this.timestamp = jsonNode.get("created_at").asText();
        this.id = jsonNode.get("id").asLong();
        this.text = jsonNode.get("text").asText();
        this.retweet = jsonNode.has("retweeted_status");
        this.user_id = jsonNode.get("user").get("id").asLong();
        if (jsonNode.has("entities") &&
                jsonNode.get("entities").has("hashtags") &&
                jsonNode.get("entities").get("hashtags").has(0)) {
            List<String> hashtags = new ArrayList<>();
            for (JsonNode i : jsonNode.get("entities").get("hashtags")) {
                hashtags.add(i.get("text").asText());
            }
            this.hashtags = hashtags
                    .stream()
                    .distinct()
                    .collect(Collectors.toList());
        }
```

We recognized a few pros and cons associated with this approach:

Pros:

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

Cons:

- If the JSON format changes (e.g. attributes are removed), the flatmap process would break, interrupting all downstream processes instead of just the ones that are directly impacted by the format changes
- If more attributes are needed 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 start our analysis.

DATA MINING

We decided to implement three representative data mining algorithms to illustrate how we may use the received Twitter data for analytical purposes.

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 sent by the Twitter API, hashtags are stored in a list. Originally when creating the Tweet object, we stored this list as-is in a *hashtag* attribute. Later on, we realized two 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 and flatmap logics 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.

```java
public class HashtagFlatMap implements FlatMapFunction<Tweet, Tuple2<String, Integer>> {

    /**
     * Tokenize hashtags and emit each hashtag as (hashtag, 1)
     */
    @Override
    public void flatMap(Tweet value, Collector<Tuple2<String, Integer>> out) throws Exception {
        // Hashtags
        if (value.getHashtags() != null) {

            for (String result : value.getHashtags()) {
                if (!result.equals("")) {
                    out.collect(new Tuple2<>(result.toLowerCase(), 1));
                }
            }
        }
    }
}
```

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.

```java
// Create Tuple2 <String, Integer> of <Hashtag, Count>
DataStream<Tuple2<String, Integer>> hashtagCount = tweets
        .flatMap(new HashtagFlatMap()) SingleOutputStreamOperator<Tuple2<String, Integer>>
        .keyBy(0) KeyedStream<Tuple2<String, Integer>, Tuple>
        .timeWindow(windowSize) WindowedStream<Tuple2<String, Integer>, Tuple, TimeWindow>
        .sum(1);
```

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

2 Geographic Heat Map

When someone sends a tweet using a phone, if he/she had his/her location service enabled, the tweet would contain the exact GPS coordinate of the location he/she sent the tweet. Or, if he/she chooses to tag a specific place (e.g. Dana Porter Library), a bounding box be included in the tweet. To capture this information, when we parse through the tweet JSON, we either store the location in which the tweet is 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)*.

```
public class GeoMapFlatMap implements FlatMapFunction<Tweet, Tuple2<Tuple2<Float, Float>, Integer>> {

    /**
     * Returns (\<Latitude, Longitude\>, 1)
     */
    @Override
    public void flatMap(Tweet value, Collector<Tuple2<Tuple2<Float, Float>, Integer>> out) {
        Tuple2<Float, Float> location = value.getCoordinates();

        if (location != null) {
            out.collect(new Tuple2<>(location, 1));
        }
    }
}
```

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

3 Tweet Type Time Series

Each tweet is either an original tweet, or a retweet. Sometimes it is 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 is a retweet, otherwise it is an original tweet.

Once again, we created a flatmap function that given a Tweet, emits a tuple of *("Tweet", 1)* or *("Retweet", 1)*.

```
public class TweetTypeCountFlatMap implements FlatMapFunction<Tweet, Tuple2<String, Integer>> {
    /**
     * Retweet/Tweet
     */
    @Override
    public void flatMap(Tweet value, Collector<Tuple2<String, Integer>> out) {
        if (value.getRetweet()) {
            out.collect(new Tuple2<>("Retweet", 1));
        } else {
            out.collect(new Tuple2<>("Tweet", 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.

```
// Create Tuple3<String, Long, Integer> of <Tweet/Retweet, Time, Count>
DataStream<Tuple3<String, Long, Integer>> tweetTypeCount = tweets
        .flatMap(new TweetTypeCountFlatMap()) SingleOutputStreamOperator<Tuple2<String, Integer>>
        .keyBy(0) KeyedStream<Tuple2<String, Integer>, Tuple>
        .timeWindow(windowSize) WindowedStream<Tuple2<String, Integer>, Tuple, TimeWindow>
        .process((tuple, context, iterable, collector) → {
                int count = 0;
                String type = "";
                for (Tuple2<String, Integer> item : iterable) {
                    type = item.f0;
                    count += item.f1;
                }
                collector.collect(new Tuple3<>(type, context.window().getStart(), count));
        });
```

The end result is a stream of tuple of *(Tweet/Retweet, window_start_time, count)*.

DATA STORAGE AND VISUALIZATION

Elasticsearch, Logstash and Kibana (commonly 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 count entry could be a single document in an index called *hashtag_count_index*. Kibana then allows you to visualize your data in these indices.

Flink's DataStream API offers a connector to Elasticsearch. 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.

1 Index

An index is a collection of documents. The easiest way to initialize an index is to do it in Elasticsearch's built-in terminal (or curl into it using bash):

```
curl -X PUT "localhost:9200/geomap-count-index?pretty" -H 'Content-Type: application/json' -d'
{}
'
curl -X PUT "localhost:9200/geomap-count-index/_mapping/_doc?pretty" -H 'Content-Type: application/json' -d'
{
  "properties" : {
    "location" : {
      "type" : "geo_point"
    },
    "count" : {
      "type" : "long"
    }
  }
}
```

We can also easily delete an index using the terminal:

```
curl -X DELETE -allow_no_indices "localhost:9200/geomap-count-index/"
```

2 Transport Client and Sink

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

Each application only needs one transport client. To initialize it:

```
private static Client instance = null;

private Client() {
    RestHighLevelClient client = new RestHighLevelClient(RestClient.builder(
            new HttpHost("127.0.0.1", 9200, "http")));
}

public static Client getInstance() {
    if (instance == null) {
        instance = new Client();
    }
    return instance;
}
```

Every time we want to perform an index change, the transport client would submit an index request to the Elasticsearch cluster.

```
public IndexRequest requestIndex(String indexName, String typeName,
                                 String docId, XContentBuilder json) {

    return Requests.indexRequest()
            .index(indexName)
            .type(typeName)
            .id(docId)
            .source(json);
}
```

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. A request could be creating a new index, updating and deleting an existing index.

```
public class GeoMapCountSink implements ElasticsearchSinkFunction
        <Tuple2<Tuple2<Float, Float>, Integer>> {

    private String indexName;
    private String docName;
    private Integer docId;

    public GeoMapCountSink(String indexName, String docName) {
        this.indexName = indexName;
        this.docName = docName;
        this.docId = 1;
    }

    @Override
    public void process(Tuple2<Tuple2<Float, Float>, Integer> t, RuntimeContext ctx,
                        RequestIndexer indexer) {
        try {
            XContentBuilder json = XContentFactory.jsonBuilder()
                    .startObject()
                    .startObject("location")
                    .field("lat", String.format("%.2f", t.f0.f0))
                    .field("lon", String.format("%.2f", t.f0.f1))
                    .endObject()
                    .field("count", t.f1)
                    .endObject();
            IndexRequest indexRequest = Client.getInstance().requestIndex(this.indexName,
                    this.docName, this.docId.toString(), json);
            indexer.add(indexRequest);
            this.docId = this.docId + 1;
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
// Create an ElasticsearchSink for Geographic Map
ElasticsearchSink.Builder<Tuple2<Tuple2<Float, Float>, Integer>> geomapCountSink =
        new ElasticsearchSink.Builder<>(
                httpHosts,
                new GeoMapCountSink("geomap-count-index", "_doc")
        );
```

Each task would have its own sink function.

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

```
hashtagCountSink.setBulkFlushMaxActions(1);
tweetTypeCountSink.setBulkFlushMaxActions(1);
geomapCountSink.setBulkFlushMaxActions(1);
```

Lastly, we attached the sinks to the appropriate DataStream.

```
hashtagCount.addSink(hashtagCountSink.build());
tweetTypeCount.addSink(tweetTypeCountSink.build());
geomapCount.addSink(geomapCountSink.build());
```

3 Visualization

Once the indices are stored in Elasticsearch, we set up several visualizations in Kibana that get updated in real-time as the indices get updated.

The Evaluation section below contains some screenshots of these visualizations.

EVALUATION

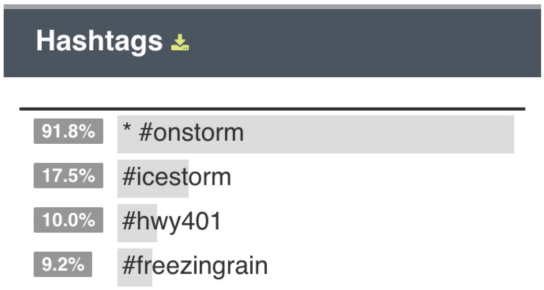
We wanted to make sure that the data we received and the indices we produced were correct. To do so, we streamed for one hour, from 3:30 pm to 4:30, on December 1st, tracking the hashtag #ONStorm, and compared our results against the same query from a third-party Twitter analytics website.

1 Hashtag Count

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

Hashtag	Count
onstorm	261
icestorm	55
hwy401	44
freezingrain	38

From the third-party analytics website:

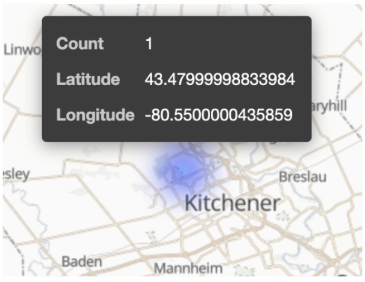


The results matched up quite well. The other hashtags also made a lot of sense (#meanwhileincanada).

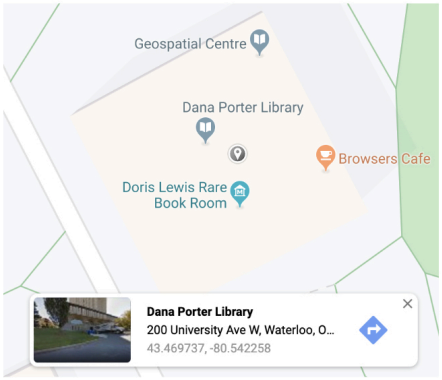
2 Geographic Heat Map

We sent a tweet from Dana Porter Library, with location service enabled.

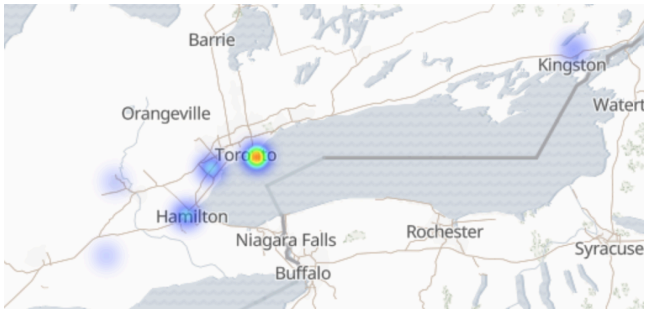
Here it is on our Kibana dashboard:



Based on Google Maps, the GPS coordinates were spot-on:



This is what it looks like for the Kitchener – GTA – Kingston area, produced by our application:

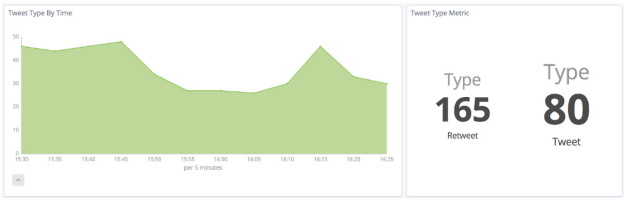


And the equivalent from the third-party website:

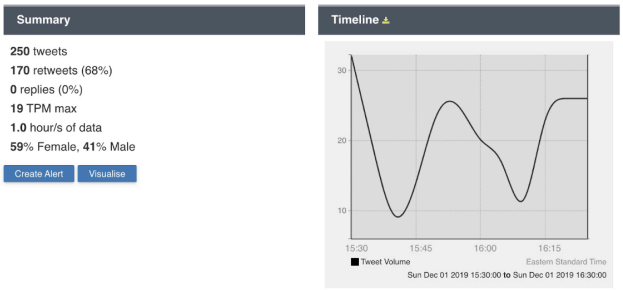
There were small levels of discrepancies, which we believe was due to differences in the level of details between our coordinate definitions.

3 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. The numbers on the right shows the breakdown over the one-hour period:



From the third-party analytics website:



Our counts, for both tweet and retweet, matched up against the third party's quite well. The trends also looked very similar.

LESSONS LEARNED

Throughout this project, we have learned a great deal about stream processing using Apache Flink. However, besides all the technical knowledge, there are also a few other valuable lessons:

1. Never assume that your data is perfect. Even though the Twitter Developer page has detailed documentations on each of the attributes in the tweet JSON, we have encountered issues with null values in fields that shouldn't be nullable. It is therefore always a good practice to implement safeguards to ensure that your data is within expectation
2. Always ensure consistent versions across the project. Initially, we installed the latest Elasticsearch 7, although we saw that the Flink Elasticsearch connector was only tested on Elasticsearch 6. This mistake caused us a lot of hindrance during the development phase. Everything seemed to be working but no data was being flown into Elasticsearch. We then realized this is due to the deprecation of some features in Elasticsearch 7. After we rolled back our Elasticsearch to v6, things started to run smoothly
3. Before diving into the application development process, always think from the ender user's perspective. In addition to the three tasks we discussed about, we had actually implemented another task that just counts word occurrences. Immediately after we've implemented the task, we realized that all of the top words are just filler words such as "the" and "is". Unless we could filter out these filler words, it couldn't have provided any meaningful insights from an end-user's perspective
4. There exists a constant trade-off between how often we want the data to be refreshed, and how large the data storage would be. Since databases do not perform any kind of concatenation

of data, the more frequent you write to the database, the large it will become. Therefore, depending on the use case, sometimes it is necessary for the data to be written to the database every five minutes, while other times, less frequency could suffice

FUTURE IMPROVEMENTS

In the end, we want to highlight some areas of improvements that could be useful for similar projects in the future:

1. Dockerize – since all the frameworks (Flink, Elasticsearch, Kibana) that we used in our project can be self-contained, we can actually build our project into a docker app and publish it on the docker hub. As dockers can be used in a wide variety of operating systems, and 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. In fact, there's a large open-source project on GitHub that has the ELK stack built in Docker using Docker Compose
2. Our application only runs on a single task node. However, to fully utilize the power of distributed computing systems and Flink, it would be interesting to set up parallel task nodes to handle each task in parallel
3. In light of the trade-off between data writing frequency and the size of the index, we could explore options to see if we can somehow flush and condense indices down to a more permanent storage space periodically to save space in Elasticsearch

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

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

[6] https://www.trendsmap.com/

[7] https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/tweet-object