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 a Flink application written in java that analyzes Twitter Streaming data including HashTag count, Tweets Geographic data and Retweets percentages based on the given search terms, hashtags and geodata as input, transfer the data into ElasticSearch and further visualize using Kibana.

KEYWORDS

Flink, Twitter, Streaming, ElasticSearch, Kibana

LEARNING OBJECTIVES

1. Understand the fundamental concept of Stream processing, challenges of Stream Processing, the difference between batch processing and stream processing.

2. Explore Flink framework, including learning its architecture, setting up a Flink environment, developing and debugging Flink applications, learn Flink's DataStream API

3. Learn how to set-up a real-time analytics dashboard using ElasticSearch and Kibana

BACKGROUND

1. Stream processing is a type of in real-time data processing engine that is designed with infinite data sets in mind (ref medium article)

2. Difference between Spark Stream Processing and Flink Stream Processing: While Spark is essentially a batch with Spark streaming as micro-batching and a special case of Spark Batch, Flink is essentially a true streaming engine treating batch as a special case of streaming with bounded data. (ref [2])

3. Why we chose Flink for Stream processing: It is the first True streaming framework with all advanced features like event time processing, watermarks, etc.

METHODOLOGY

Set-up Apache Flink

1. Create a Maven project using Archetype
2. Add appropriate dependencies in pom.xml, including twitter-connector, elasticsearch-connector, log4j, etc..
3. Understand parallelism
4. Understand window (tumbling vs rolling)

Twitter connector

1. Understand credential pass-in through parameter

After we applied a Twitter developer account, we read the documentation of Flink ParameterTool and learned how to pass in twitter consumerKey and tokens to authenticate our program to get data from Twitter.eter

2. Understand EndPoint

We needed to select an appropriate endpoint provided by Twitter for our program, thus we did our research about a few endpoints, including StatusesSampleEndpoint, StatusesFilterEndpoint, SitestreamEndpoint, etc.. We decided to use StatusesFilterEndpoint which is best suited in our application.

3. Understand Tweet JSON format

In order to do further data analysis on the received tweet data, we familiarized ourselves with Tweet JSON objects, created a corresponding java class and use the different attribute data for the different analyses.

```
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 referred to the API documentation on Twitter Developer's website (ref) for the format of the Tweet objects. 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 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.

We recognize 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. fields are removed), Tweet object creation would break, interrupting all of the tasks instead of just the task that's affected by the field removal.
If we require more number of attributes in the future, the memory and runtime benefit could be eliminated

Tasks
Task #1: Trending Hashtags
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 store this list as-is in a *hashtag attribute*.

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

Later on, we realized two essential facts regarding 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 perspective, we decided that each hashtag should really be considered only once, and that they should be case-insensitive. Therefore, we modified our parsing logic and flatmap logic to make hashtags distinct and lower-case.

```
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) {
            Iterator<String> iter = value.getHashtags().iterator();

            while (iter.hasNext()) {
                String result = iter.next();

                if (!result.equals("")) {
                    out.collect(new Tuple2<>(result, 1));
                }
            }
        }
    }
}
```

For aggregate we applied a simple *keyBy -> Tumbling Window -> sum* logic that aggregates up all occurrences of a given hashtag, within the specific tumbling window period.

```
DataStream<Tuple2<String, Integer>> hashtagCount = tweets
        .flatMap(new HashtagFlatMap())
        .keyBy(0)
        .timeWindow(windowSize)
        .sum(1);
```

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

Task #2: Geographic Map
When you tweet something, if you are tweeting from your phone and you had your location service enabled, your tweet would contain your location information. Or, you can choose to tag a specific place (e.g. Dana Porter Library). This shows up as a bounding box of coordinates in the tweet JSON. When we parse the JSON, we either store the location in which the tweet was sent (if location service was enabled), or simplify the bounding box by taking the average of the corner values, i.e. finding the midpoint, and store it in the *coordinate* attribute as a tuple of floats representing its latitude and longitude.

Similar to hashtags, not all tweets have locations associated with them (in cases where you disabled location service, and you didn't tag a location), so we created a flatmap that emits tuples of (<latitude, longitude>, 1), if available. Later on, we realized that ElasticSearch actually reverses the order in which geo_points are stored (i.e. longitude before latitude), so we modified our sinking function to reverse the coordinates.

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

```
@Override
public void flatMap(Tweet value, Collector<Tuple2<Tuple2<Float, Float>, Integer>> out) throws Exception {
    Tuple2<Float, Float> location = value.getCoordinates();

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

Task #3: Tweet Type Time-Series

Each tweet is either an original tweet, or a retweet. Sometimes it's interesting to see to see this tweet typo evolve 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's not null, then it's a retweet, otherwise it's an original tweet.

We created a Flatmap that emits tuples of ("Tweet", 1) or ("Retweet", 1).

```
12      @Override
13      public void flatMap(Tweet value, Collector<Tuple2<String, Integer>> out) throws Exception {
14          if (value.getRetweet()) {
15              out.collect(new Tuple2<>("Retweet", 1));
16          } else {
17              out.collect(new Tuple2<>("Tweet", 1));
18          }
19      }
20  }
```

In order to create a time-series, we need time as a data field. To do this, we had to create a custom *process* function that, not long aggregates all instances of "Tweet"/"Retweet" in a given window period, also append the start time of the window as an additional field.

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

@Override
public void process(Tuple3<String, Long, Integer> t, RuntimeContext ctx, RequestIndexer indexer) {
    try {
        XContentBuilder json = XContentFactory.jsonBuilder()
                .startObject()
                .field("tweet_type", t.f0)
                .field("timestamp", t.f1)
                .field("count", t.f2)
                .endObject();
        IndexRequest indexRequest = Client.getInstance().requestIndex(this.indexName, this.docName, this.docId.toString(),
        indexer.add(indexRequest);
        this.docId = this.docId + 1;
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

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

Task #4: Elasticsearch, Logstash and Kibana

Task #4.1 Set-up

Elasticsearch, Logstash and Kibana (also known as ELK) is a suite of open-source 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 would be a single document in an index called *hashtag_index*. Kibana then lets you visualize the data in these indices.

Flink's DataSteam API offers a connector to Elasticsearch. It is fairly easy to set-up - just need to specify the dependency in the Maven repository. This connector provides *sinks* that can request different actions on documents/indices, such as creating, updating and deleting.

First, we installed the latest version (v7.4) of the ELK stack to our local computers. Then, as we were developing the sink functions from Flink to Elasticsearch, we realized several compatibility issues between the version we had (v7.8), 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.

Task 4.2 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):

```
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 delete an index in the same fashion:

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

Task 4.3 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 straight-forward: just need to specify a port:

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

Once initialized, everytime we want to perform an index change (e.g. add a document), the REST 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.

```
@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(),
        indexer.add(indexRequest);
        this.docId = this.docId + 1;
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
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. 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:

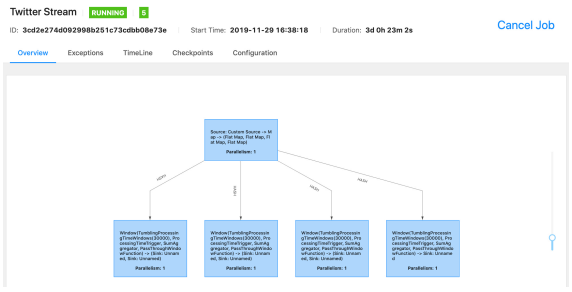
```
wordCountSink.setBulkFlushMaxActions(1);
hashtagCountSink.setBulkFlushMaxActions(1);
tweetTypeCountSink.setBulkFlushMaxActions(1);
geomapCountSink.setBulkFlushMaxActions(1);
```

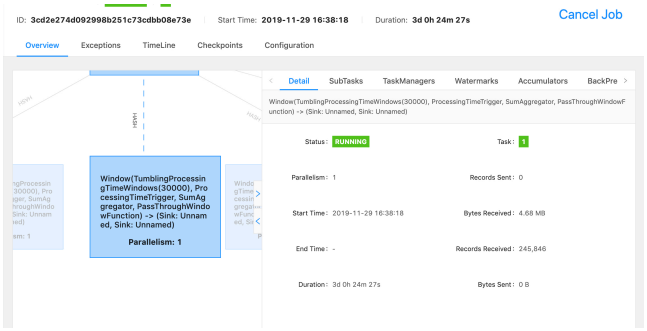
Lastly, we just need to attach them to the approrpirate DataStream.

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

Task 4.4 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 subsequence DataStream jobs.

Task 4.5 Visualization

After import index from ElasticsSearch into Kibana, you can visualize the result shown in the next section.

EVALUATION

1. Hashtag

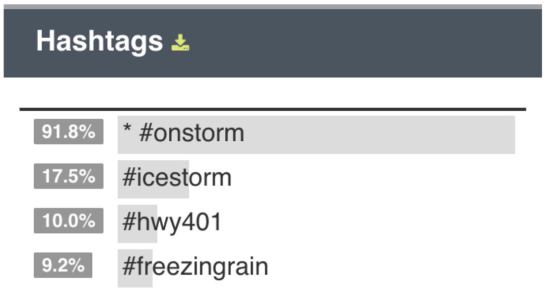
We ran our program on December 1st, with input search term ONStorm, from 3:30 pm to 4:30 pm

Here is the result of ours:

Hashtag	Count
onstorm	261
icestorm	55
hwy401	44
freezingrain	38

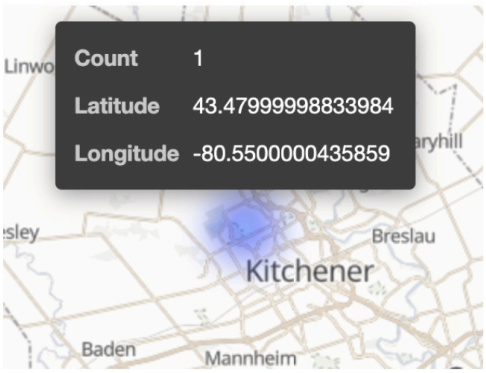
It matches the result on 3rd party website.

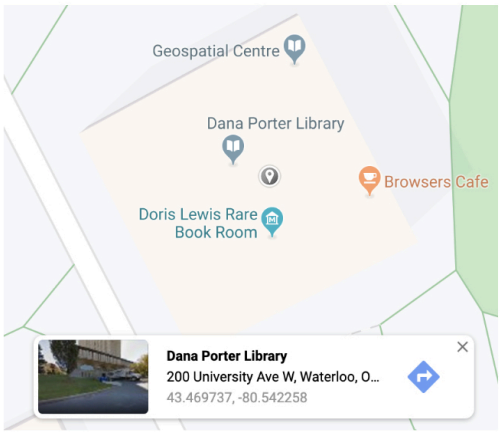
https://www.trendsmap.com/



2. GeoMap

After enabled location service, we sent a tweet from Dana Porter Library with hashtag #ONStorm, we are able to see it on our dashboard.

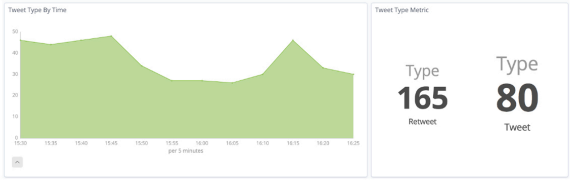




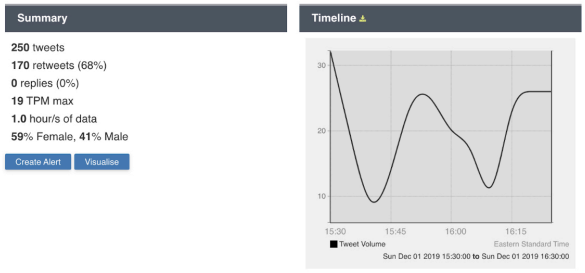
3. Tweet vs Retweet rate

We can see our program has about the same percentage as the 3rd party website.

Our result:



3rd party result:



LESSON LEARNT

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

1.1 WordCount - doesn't really mean much since a lot are filler words. Unless we can filter out, the result doesn't matter

1.2 FavouriteCount - doesn't make much sense either since normally the tweet will not be liked by someone else in such a short window.

6. Irregularities in data (emojis, pictures, user mentions)

FUTURE IMPROVEMENTS

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

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