**Harvard University Extension School**

**"Principles of Big Data Processing"**

**CSCI E-88, Fall 2018**

**Final Project**

**by Andrew Caide**

## Project Goal and Problem Statement

The inspiration for my project stems from my current role in a reputation and marketing driven firm. In my project shall conduct NLP on twitter data to investigate brand perception across the world for a select group of brands within the same industry.

The technology stack used in this project is outlined as follows:

1. Flume
2. Kafka
3. Python, Aws Comprehend
4. Kafka, Kafka Connect
5. Elastic Search, Kibana

The two underlined technologies are the novel implementations for this project; they haven’t been used in class. For this project, we shall investigate the following companies in the technology space such as Apple, Microsoft, and Tesla.

## YouTube Video URL

<https://youtu.be/oDCv5LeM0ME>

## Big Data Source

The data source for this project is twitter. A lot of tweaking was conducted to the Flume source files to extract non-standard parameters from twitter, such as geo-spacial coordinates and user information. The text data will be analyzed with AWS Comprehend to obtain sentiment scores, but only if geo-spacial data exists for the tweet. This is to produce interesting data geographical plots and not overwhelm AWS comprehend.

Key fields pushed into ElasticSearch will include

* Company Name
* Tweet
* User
* Date-Time of Tweet
* Composite Sentiment Score
* Negative Score
* Positive Score

## Expected results

There are no analytical expectations for this pipeline, however I am expecting companies like Apple, Amazon, and Facebook to receive the lowest scores based on their company corporate-social reputation for workplace/ethics violations and companies like Google and Microsoft to stay good to neutral. These scores will be calculated from the AWS Comprehend result (composite = positive score – negative score). I am not sure how these companies will look on a global perspective.

## Processing Pipeline

There are a lot of edits to configuration files for this project. To keep the documentation in clean shape, I’ll provide them in the next section below. The final pipe-line is as follows:

1. **Source (Twitter):** Free data shall be collected from the 1% Twitter firehouse with Flume agents. This will be streaming, since content is added continuously.
2. **Collection (Flume)**: Flume shall be used as the collection tier with a Kafka endpoint. Flume source files will have to be tweaked to extract the information I need most. Because there native/predeveloped clients for this job, I’ll be using Flume for collection.
3. **Messaging (Kafka)**: Kafka shall be used to pass data to a Python app which will trigger a series of calculations. Kafka will be used again to collect the data post analysis for storage, indexing, and visualization.
4. **Analysis (Python, AWS Comprehend)**: We shall use AWS Comprehend to compute sentiment scores for our selected data and append the score each event. A pythonic application shall be developed to handle data access, analysis, and cleaning. Once AWS Comprehend has analyzed the event, the data shall be stored and indexed in ElasticSearch.
5. **Indexing/Presentation (ElasticSearch, Kibana)**: ElasticSearch is a document storage with strong indexing capabilities, so for this project I shall sidestep a master database storage for this project. Furthermore, Kafka temporarily stores data in its HDFS, so we’ll double down on the decision. However, in a production environment, all of the data collected would be stored before (or after depending on how analysis is conducted) in case any additional analysis would need to be repeated. The schema would look like **/technology/specific\_brand\_name/**. Key fields used has already been discussed above and shall be revisited below in the discussion of the ElasticSearch index.



## Implementation

1. **Flume**

The very first step is to log into the kafka broker and go to /app/kafka/… and investigate config/server.properties to find out where zookeeper is. Look for the IP address ending in :2181. Next find the advertised listeners ending with :9092. Make sure the listener is added to the flume sink later on.

The first step is to update a couple of twitter library files: twitter4j-core, media-support, and stream from 1.0.x to 4.0.x. These jars can be found at <https://mvnrepository.com/artifact/org.twitter4j>. Download them onto your flume library at apache-flume-1.6.0-bin/lib. Give the files execution permission so with **sudo chmod -x twitter4j-….jar**. Next give all the files read permission with **sudo chmod +rrr twitter4j-…** and it should be ready to go**.**

The next step is slightly more involved. In order to get the exact parameters we need out of Flume, including setting some options such as key-words and language filters, we’ll need to create a custom twitter source. This is an incredibly arduous process that requires solid Java coding. Lucky for me there are some pretty good sources floating around already. The one I chose for this project can be found at the following address: <https://github.com/mmartsen/flume-tools/tree/master/sources>

Download the three files to a temp folder. Before any more progression, make sure Java SDK is installed on the computer. Before we comepile the source code, we need to make sure the system knows where Flume is. Either add the Flume/lib directory and the temp folder to the CLASSPATH. A shortcut would be to execute the following command:

**export CLASSPATH="/home/ec2-user/apache-flume-1.6.0-bin/lib/\*:~/Final2/\*"**

Compile the source code with the command

**javac -d . TwitterSource.java TwitterSourceCounter.java TwitterSourceCounterMBean.java**

If everything worked successfully, a folder should have been created called “mmartsen” with a few more subdirectories and classes at the very end. Create a file called “Manifest.txt” with a text editor like vim. Here it should contain the following line (**note: hit enter at the end of the line!**)

**Main-Class: mmartsen.flume.sources.twitter.TwitterSource**

Now create a JAR with the following command:

**jar cfm NewTwitterSource.jar Manifest.txt mmartsen/flume/souces/twitter/\*.class**

Copy this file to Flume/lib!

Lastly, we have to configure the flume configuration. This has been done in class before; update the flume-conf.properties file to match the following:

#########################################################

# Andrew Caide

# CS88 Final Project

# Twitter Brand Analysis Project

# Three agents will run simultaniously

agent.sources = Twitter

agent.channels = Memory

agent.sinks = KafkaSink

# Configure the Sources.

# The agent below specifies the jar we just finished creating!

agent.sources.Twitter.type = mmartsen.flume.sources.twitter.TwitterSource

agent.sources.Twitter.consumerKey=<your consumer key>

agent.sources.Twitter.consumerSecret=<…>

agent.sources.Twitter.accessToken=<access token>

agent.sources.Twitter.accessTokenSecret=<…>

# Here are the new features: keywords and languages. Play around with these!

agent.sources.Twitter.keywords = apple watch, iphone, mac, macbook, imac, AAPL, tim cook

agent.sources.Twitter.language = en

agent.sources.Twitter.channels = Memory

# Configure the Sink

agent.sinks.KafkaSink.type = org.apache.flume.sink.kafka.KafkaSink

agent.sinks.KafkaSink.brokerList=<**ADVERTISED LISTENER FROM KAFKA**>:9092

agent.sinks.KafkaSink.topic = **TOPIC\_NAME**

agent.sinks.KafkaSink.batchSize = 40

agent.sinks.KafkaSink.channel = Memory

# Configure the Channel

agent.channels.Memory.type = memory

agent.channels.Memory.capacity = 1000

agent.channels.Memory.transactionCapacity = 100

#########################################################

To run the application:

**flume-ng agent -n agent -c appleconf/conf/ -f configFolder/conf/flume-conf.properties**

1. **Kafka**

Kafka should work just fine, thank god.

Create the topic, listen to it, and delete it with the following commands:

* app/kafka/kafka\_2.11-2.0.0/bin/kafka-topics.sh --zookeeper <**ZOOKEEPER\_ADDRESS**>:2181 --create --topic **TOPIC\_NAME** --partitions 2 --replication-factor 1
* /app/kafka/kafka\_2.11-2.0.0/bin/kafka-console-consumer.sh --bootstrap-server <**ZOOKEEPER\_ADDRESS**>9092 --topic **TOPIC\_NAME** --from-beginning
* /app/kafka/kafka\_2.11-2.0.0/bin/kafka-topics.sh --zookeeper <**ZOOKEEPER\_ADDRESS**>:2181 --delete --topic **TOPIC\_NAME**

1. **Python**

The full code is below.

import json, random, time, boto3, sys, glob, io, datetime

#import avro.schema, avro.io, avro.datafile,

from avro.io import DatumWriter

from kafka import KafkaConsumer

from confluent\_kafka import avro

from confluent\_kafka.avro import AvroProducer

############################################################################

# DONT TOUCH

input\_topic = sys.argv[1:][0]

if len(sys.argv[1:])==1:

output\_topic = input\_topic

else:

output\_topic = sys.argv[1:][1]

SCHEMA\_PATH =glob.glob("./\*avsc")[0]

print("Reading data from {}, exporting calculations to {}.".format(input\_topic, output\_topic))

# CONSUMER (import)

consumer = KafkaConsumer(input\_topic,

group\_id=None,

bootstrap\_servers=['ec2-00-000-0-000.compute-1.amazonaws.com:9092'],

auto\_offset\_reset='earliest') #This ensures you read everything from the very begining

# CALCULATION

comprehend = boto3.client(service\_name='comprehend', region\_name='us-east-1')

# PRODUCER (export)

AvroProducerConf = {

'bootstrap.servers': 'localhost:9092',

'schema.registry.url': 'http://localhost:8081',

'broker.address.family': 'v4'

}

value\_schema = avro.load(SCHEMA\_PATH)

avroProducer = AvroProducer(

AvroProducerConf,

default\_value\_schema=value\_schema

)

# Misc (keeping track of events)

count = 0

i=0

n=0

###########################################################################################

try:

for msg in consumer:

val = json.loads(msg.value.decode("utf-8"))

# We'll only look at people who have geospacial coordinates for presentation purposes.

if val['geo'] == None:

if count%10 == 0:

print("Counted {} without geographical coordinates.".format(count))

count += 1

else:

print("Found somebody with Geo Cords:")

res = json.dumps(comprehend.detect\_sentiment(Text=val['text'], LanguageCode=val['lang']))

res = json.loads(res)

print('----')

i+=1

#######################

# Clean up!

# Dealing with missing data for location; let's dump it somewhere exotic

# This shouldn't even matter; we're filtering out non-localizable cases.

place = val['place']

if place == None:

geo = "MD"

else:

geo = place['country\_code']

if val['coordinates'] == None:

cords = "18.7669,46.8691"

else:

print(val['coordinates'])

cords=str(round(val['coordinates']['coordinates'][1],4)) +","+str(round(val['coordinates']['coordinates'][0],4))

# Not going to report neutral score - it's boring. Not sure what to do with it.

# Let's do our own elaborate calculation for demostration purposes.

unrounded=(res['SentimentScore']['Positive']-res['SentimentScore']['Negative'])\*100

comp\_score = round(unrounded,2)

print("Composite score: {}".format(comp\_score))

# Convert milliseconds to datetime

us = int(val['timestamp\_ms'])

ms = us/1000

dt = datetime.datetime.fromtimestamp(ms).strftime("%Y-%m-%dT%H:%M:%S.%fZ")#('%c')

print(dt)

########################

# Pass through avro/Kafka

value = {

"company":input\_topic,

"user": val['user']['screen\_name'],

"tweet":val['text'],

"timestamp":str(dt),

"sentiment":res['Sentiment'],

"compositeScore":comp\_score,

"negScore":round(res['SentimentScore']['Negative'],4),

"posScore":round(res['SentimentScore']['Positive'],4),

"geo":geo,

"coordinates":cords

}

print(value)

avroProducer.produce(topic=output\_topic, value=value)

avroProducer.flush()

# Keeping track of neutral rate:

if res['Sentiment']=="NEUTRAL":

n+=1

print("Neutral rate: {}%".format(n/(i+1)\*100))

print("-------\n")

except KeyboardInterrupt:

sys.stderr.write('%% Aborted by user\n')

Additionally, twitter.avsc is an Avro schema we need to create to pass data into KafkaConnect from Python. Make sure python can find it with glob.glob().

{

"namespace": "example.avro",

"type": "record",

"name": "tweet",

"fields": [

{"name":"company", "type":"string"},

{"name": "user", "type": "string"},

{"name": "tweet", "type": "string"},

{"name": "timestamp", "type":"string"},

{"name": "sentiment", "type": "string"},

{"name": "compositeScore", "type": "double"},

{"name": "negScore", "type": "double"},

{"name": "posScore", "type": "double"},

{"name": "geo", "type": ["null","string"]},

{"name": "coordinates", "type": "string"}

]

}

**Note how simple everything is: there are no fancy logical types nor lists. Keep it simple.**

The code should be able to read from your AWS Kafka Broker if you provided the correct address. Use the following command to execute the program:

**python3 AndrewsCode.py InputTopicFromKafka OutputTopicToKafkaConnect**

1. **KafkaConnect**

There are a couple of configuration files that needed to be updated. Before any of that, let’s make sure Confluent is running with <confluent\_location>/bin/confluent start. Make sure the avro localhost directory can be updated when playing around with updates. Run the following command:

**curl -X PUT http://localhost:8081/config -d '{"compatibility": "NONE"}' -H "Content-Type:application/json"**

Next we need to update the KafkaConnect configurations. Navigate/Vim to etc/kafka-connect-elasticsearch/<config file, make a new one, doesn’t matter> and update it with the following lines:

name=elasticsearch-sink

connector.class=io.confluent.connect.elasticsearch.ElasticsearchSinkConnector

tasks.max=1

topics=tesla,apple,microsoft

topic.index.map=tesla:company\_index,microsoft:company\_index,apple:company\_index

#tesla\_index,apple:apple\_index,microsoft:microsoft\_index # Previously I mapped everything to different indexes; this is bad because it makes it difficult to do visualizatgions across different indexes.

# CRITICAL: Map your topics to the index with **TOPIC\_NAME:ELASTICSEARCH\_INDEX\_NAME otherwise you’ll never get your data!**

key.ignore=true

connection.url=http://localhost:9200

type.name=kafka-connect

key.converter=io.confluent.connect.avro.AvroConverter

key.converter.schema.registry.url=http://localhost:8081

value.converter=io.confluent.connect.avro.AvroConverter

value.converter.schema.registry.url=http://localhost:8081

That should be about it for KafkaConnect. To begin the connector sink, execute with

**bin/confluent load elasticsearch-sink -d etc/kafka-connect-elasticsearch/tesla.properties**

1. **ElasticSearch/Kibana**

There are some complex data types (time and coordinates) that should be explicitly specified by the construction of the index, otherwise they’ll default to string types. Use the schema below to match the data.

PUT /company\_index/

{

"mappings":{

"kafka-connect":{

"properties":{

"company":{

"type":"keyword"

},

"user":{

"type":"keyword"

},

"tweet":{

"type":"keyword"

},

"timestamp":{

"type":"date",

"format":"date\_time"

},

"sentiment":{

"type":"keyword"

},

"compositeScore":{

"type":"double"

},

"negScore":{

"type":"double"

},

"posScore":{

"type":"double"

},

"geo":{

"type":"keyword"

},

"coordinates":{

"type":"geo\_point"

}

}

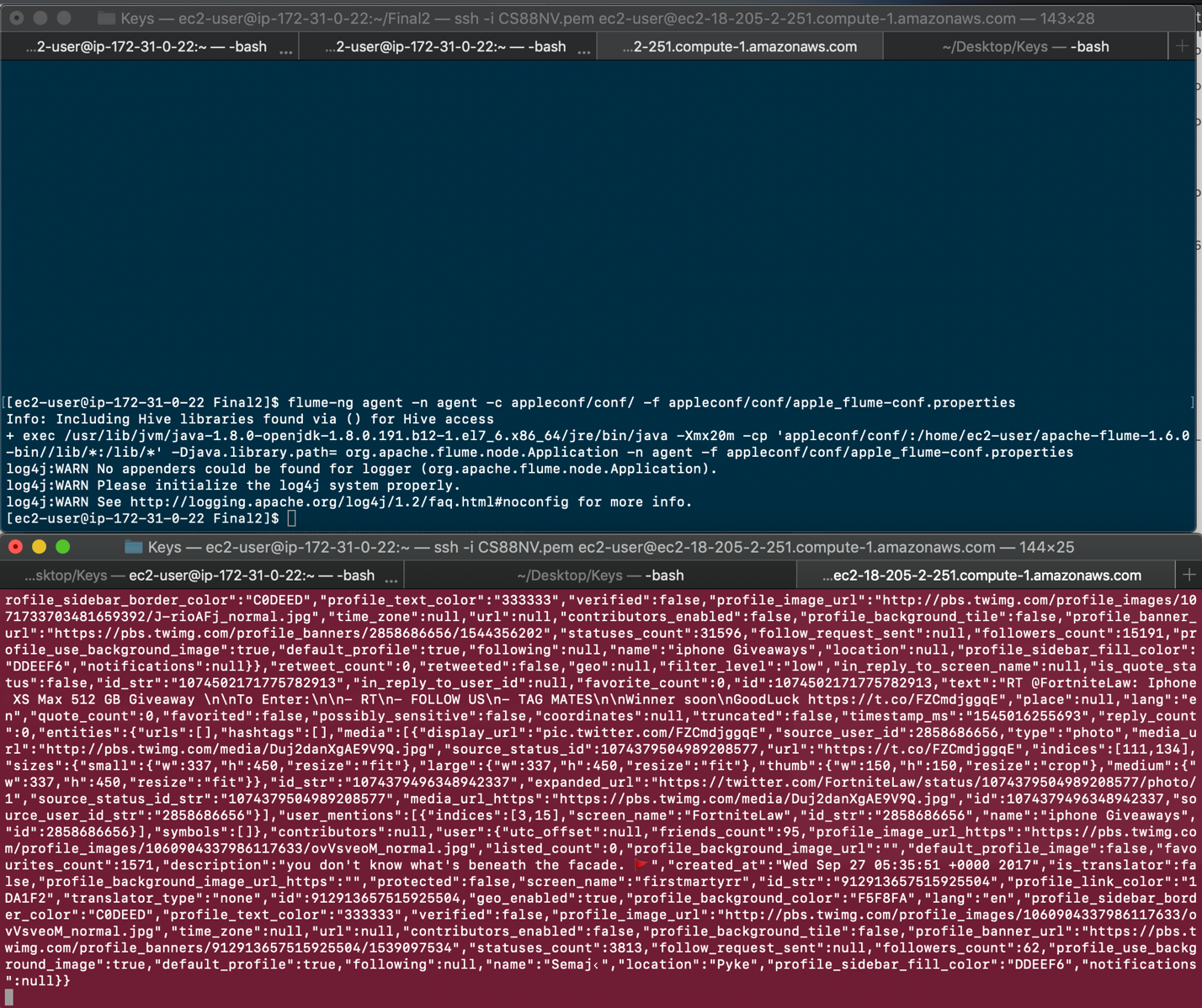
}

}

}

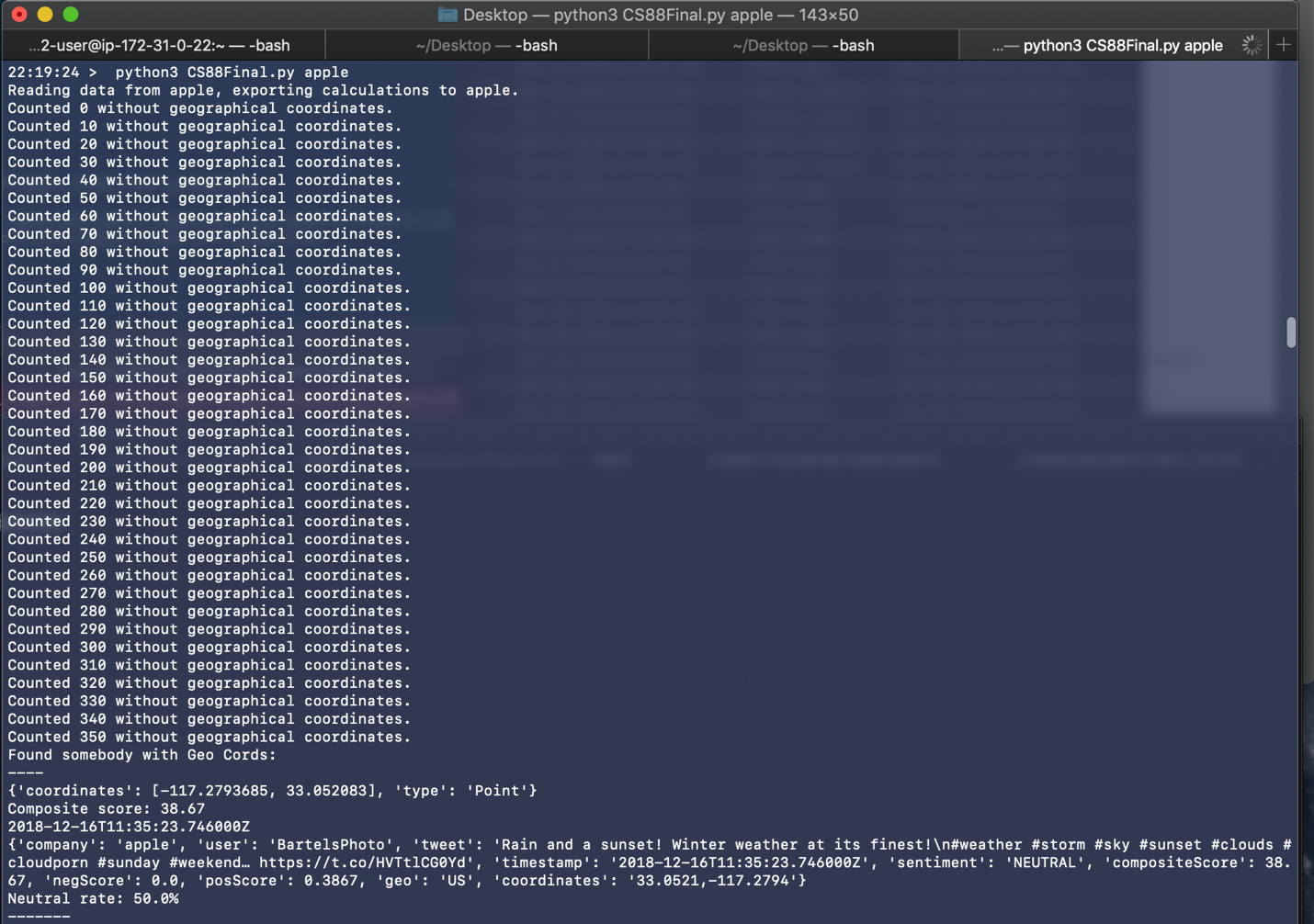
## Results

Flume/Kafka:



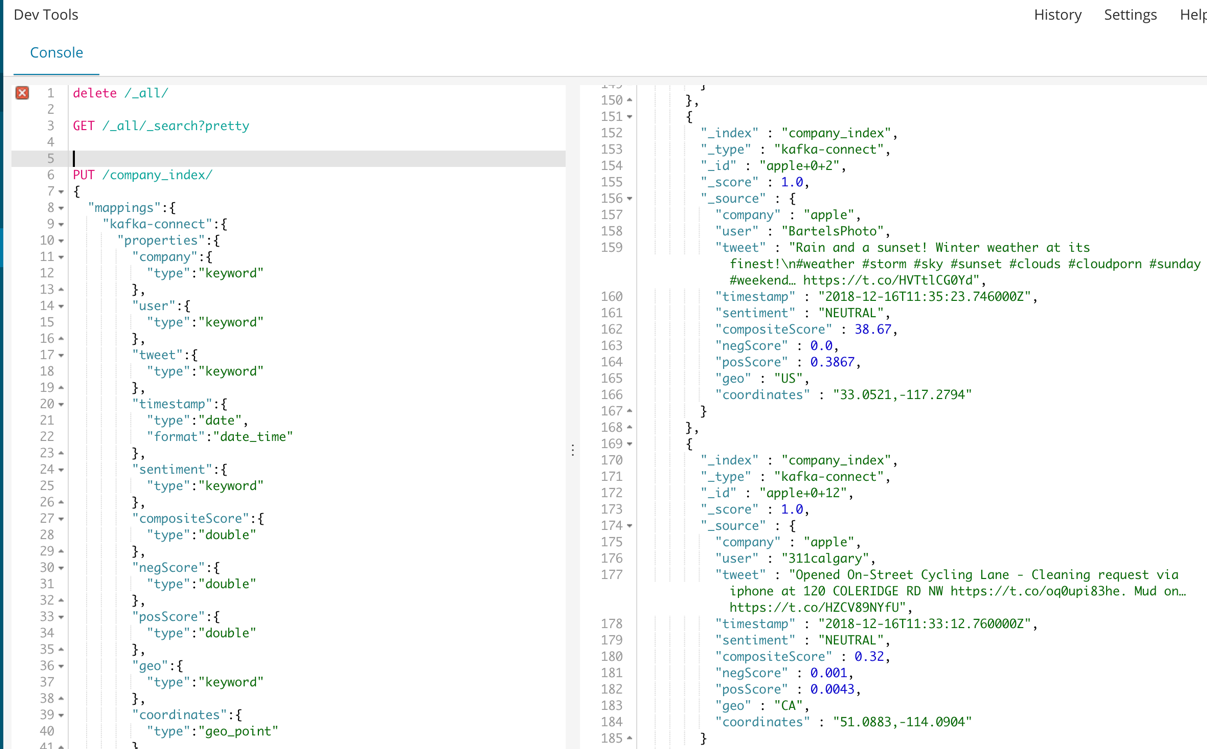
Python Script:

Note: I’m filtering to only accept/analyze data with coordinates.

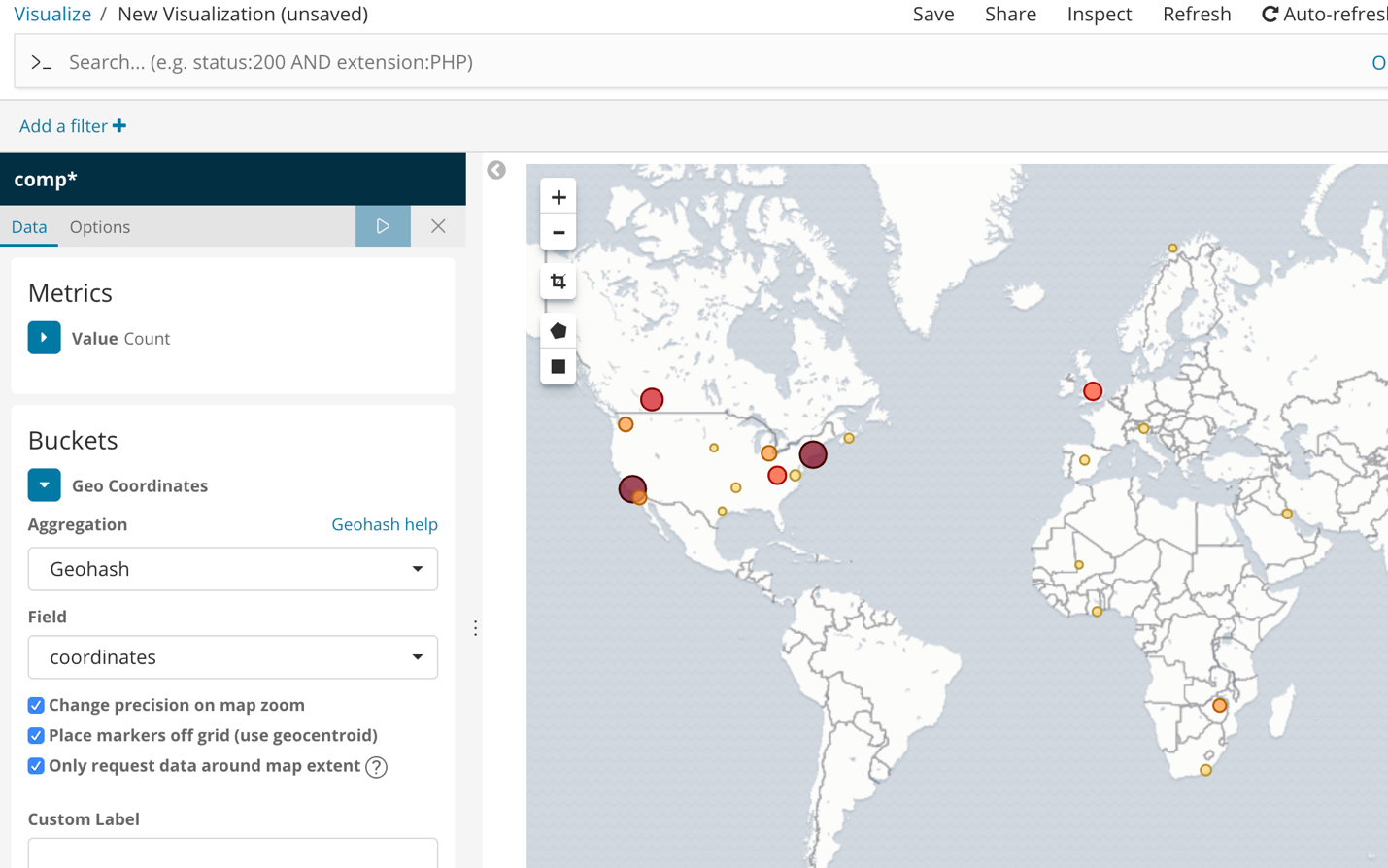


Visualization Tier:

Note that the first text matches the result above!



Visualization Continued: Note that geo-spacial coordinates work!



## Conclusions and Lesson Learned

Describe what you have learned during this project:

* What issues did you have?

Countless; each step of the pipeline yielded tremendous work, from configuration connection. Establishing a successful pipe to Kafka Connect from Python was the longest endeavor (~3 days of configuration troubleshooting). In summary:

1. Flume
   1. Accessing Twitter was easy
   2. Obtaining useful twitter data was brutal. Several flume source files required update.
   3. Java scripts for information pulling had to be compiled and Maven’ed to be used as additional kwargs for the flume agent.
   4. Two days of troubleshooting.
2. Kafka
   1. This was the least of my problems. No real troubleshooting required.
3. Python
   1. Scripting was two relatively easy days of work. Data was originally being registered as Avro formats, but this was corrected. The guides to using AWS Comprehend were comprehensible.
   2. Passing data into Kafka Connect, however, was an entirely different story.
4. Kafka Connect
   1. A component of the Confluent stack in conjunction with Kafka (regular).
   2. Data had to be passed in as Avro files.
   3. An Avro file repository (localhost:8081) had to be configured and updated.
   4. Configuration files had to be drafted for the ElasticSearch sink.
   5. The Source had to be Avro, which required a very specific Python module to be used.
   6. About three days were lost while I was struggling to pipe data in from Python.
5. ElasticSearch
   1. The key take-away from this struggle was learning to pass everything in as strings and to avoid complex avro data-types. If a date can be registered as a string, make sure that’s how python is passing it out.
   2. This is critical for coordinates and time.
   3. Configure the index beforehand.
   4. Two days of space-time troubleshooting. Always look at the plotting requests and troubleshoot from there!

* What limitations, if any, did you run into with the technologies used?

Flume wasn’t up-to-date with the native packages. I had to do digging around maven/java repositories to upgrade source files.

Everything else was up-to-date enough for me to work with.

* What would you do differently next time?

Incorporate a master storage after processing through AWS Comprehend – using non-native, cost-driven analysis is expensive and should be reduced to a once-only/ever operation. Because I was tight on time towards the end of this project and data size wasn’t colossal, I omitted this step. Omission on production-scale would be massively ill-advised.

* What would you like to improve if you had more time?

See above! I would also move everything to AWS (I couldn’t do 4 and 5 in time; neither would be terribly intensive, just a hassle to troubleshoot).

* What alternatives to technologies you used you might consider?

AWS red-shift as a storage from the python app and an aws-based visualization tool that could be easily coupled with red-shift.

* Where would you take your project next?

There are three obvious areas of improvement that would push this project into something a lot stronger:

1. Completely cloud-based: move my analysis and visualization to AWS.
2. Expand my searches/flume sources to include data from other companies.
3. Fix a storage solution post-analysis.
4. Move analysis out of AWS comprehend to cut costs.