TWITTER DATA ANALYSIS USING APACHE KAFKA & SPARK

CSCE 678 : FINAL PROJECT REPORT

MAY 2019

TEAM MEMBERS :
(1) Srividhya Balaji UIN : 827007169
(2) Abindu Dhar UIN : 928000823
(3) Harmeet Singh Sethi UIN : 727004166
(4) Raja Mohamed Aarif UIN : 928000635

INTRODUCTION

The past few years have seen tremendous interest in large-scale data analysis, as data volumes in both industry and research continue to outgrow the processing speed of individual machines.

Twitter is one of the most popular social media sites. Users post 500 million tweets daily in average on Twitter[1] and the number of active Twitter users exceeds 22% of the internet users in the world[1]. These statistics reflect Twitter's global outreach and potential impact. In addition to having a global coverage of issues, Twitter provides a media platform that enables sharing opinions easily using various content forms including text, images, links unlike many other social media platforms. Moreover, providing near real-time access to public posts through the API makes Twitter a suitable platform for large scale near real-time opinion mining.

Existing tools like Hadoop kicked off as an ecosystem for parallel data analysis for large clusters[2]. However, these tools have so far been optimized for one-pass batch processing of on-disk data, which makes them slow for interactive data exploration and for the more complex multi-pass analytics algorithms. We use Spark a new cluster computing framework that can run applications up to 40× faster than Hadoop by keeping data in memory, and can be used interactively to query large datasets with sub-second latency.

These motivated us to perform real time data analysis on Twitter feed using Spark computing Framework. We find this to be important as it provides a means to measure and boost our impact on Twitter.

RELATED WORK

For Twitter analysis, most of the existing work focuses on improving sentiment analysis, using deep learning and machine learning techniques to extract and mine the polarity of the feed data. Some of the early results on sentiment analysis of Twitter data were by Go et al. (2009)[11], (Bermingham and Smeaton, 2010) [12] and Pak and Paroubek (2010) [13]. Go et al. (2009) used distant learning to acquire sentiment data and used tweets ending in positive emoticons like ":)" ":-)" as positive and negative emoticons like ":(" ":-(" as negative. They built models using Naive Bayes, MaxEnt and Support Vector Machines (SVM). For feature space they used Unigram, Bigram model in conjunction with parts-of-speech (POS) features. Pak and Paroubek (2010) collected data following a similar distant learning paradigm. They performed a different classification task though: subjective versus objective. For subjective data they collected the tweets ending with emoticons in the same manner as Go et al. (2009). For objective data they crawled twitter accounts of popular newspapers like "New York Times", "Washington Posts" etc. They reported that POS and bigrams both help (contrary to results presented by Go et al. (2009)). Both these approaches, however, were primarily based on n-gram models.

Another significant effort for sentiment classification on Twitter data was by Barbosa and Feng (2010) [14]. They used polarity predictions from three websites as noisy labels to train a model and used 1000 manually labeled tweets for tuning and another 1000 manually labeled tweets for testing. They proposed the use of syntax features of tweets like retweet, hashtags, link, punctuation and exclamation marks in conjunction with features like prior polarity of words and POS of words. Further research by Apoorv and Xie (2011) extended their approach by using real valued prior polarity, and by combining prior polarity with POS that enhanced the overall classifier performance [15].

For this project, our primary focus is to analyse real time twitter feeds that showcase the application of distributed streaming services. We are using NLP processing library TextBlob, for analyzing sentiment attached to each tweet received in a batch. Novelty of our project lies in the fact that we are finally showcasing the sentiments attached with each trending hashtag for a given duration.

CLOUD COMPUTING PRELIMINARIES

In this section we describe the two major frameworks we employ into building our application for real time twitter data analysis.

A. APACHE SPARK FRAMEWORK

Distributed cluster computing frameworks like Hadoop/MapReduce have been in service for a long time, allowing parallel data intensive computation tasks to run in a fault tolerant and efficient manner. However, one of their major disadvantage is that they don't use “In Memory” computation, tasks are distributed among the nodes of a cluster, which in turn save data on disk, each node has to later load the data from the disk and save the data into disk, after performing an operation, incurring high I/O cost. Big Data poses another serious challenge, due to its large volume and high velocity, added to it is the complexity of working on a live stream of data to process events in real time, where data must be consumed at the rate at which it is being generated.

Keeping these requirements in mind Apache Spark [8] was created at University of California, Berkeley’s AMPLab in 2009. Apache Spark is open source, and runs on top of Hadoop, it can run tasks up to 20 times faster by utilizing in-memory computations. Spark provides API support for Scala, Java, R as well as Python.

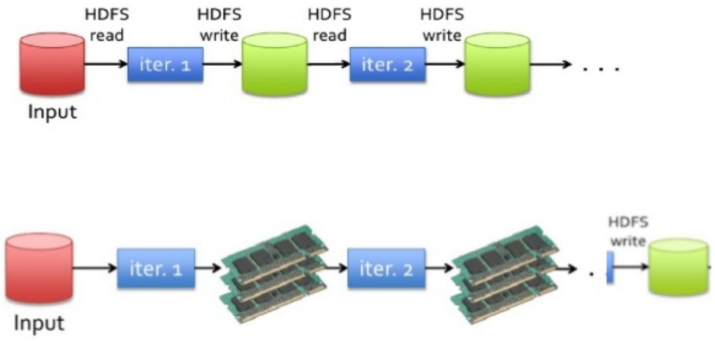

FIGURE 1 : Spark vs Hadoop

One of the major abstractions that Spark provides is the RDD [7], the resilient distributed datasets, that enables efficient data reuse. RDDs are efficient, fault tolerant, parallel data structures, that let users explicitly choose data to persist in

memory, control partitioning patterns for optimisation, and provide highly optimised set of transformations (map, filter, flatmap etc) and actions (collect, take etc) on these datasets. RDDs are a very efficient abstraction for applications involving data reuse, especially in real time data analytics problems.

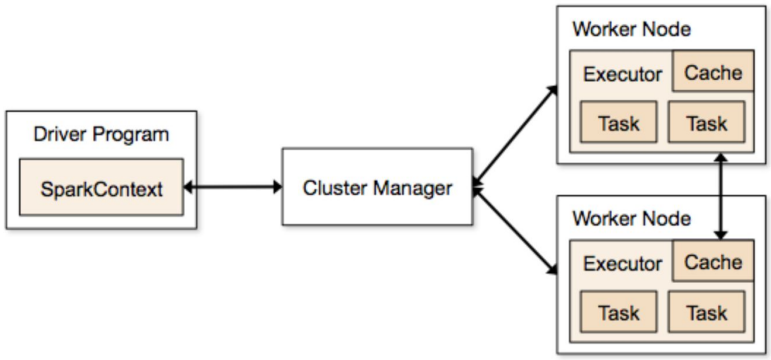

FIGURE 2 : Spark Application Architecture

Figure 2 shows the architecture of a spark application. The spark context holds a connection with Spark cluster manager and all Spark applications run as independent set of processes, coordinated by the spark context. The driver runs the main function of an application and creates the spark context. A worker, on the other hand, is any node that can run program in the cluster. The cluster manager allocates resources to each application in driver program. There are three types of cluster managers supported by Apache Spark – Standalone, Mesos and YARN. Apache Spark is agnostic to the underlying cluster manager, so we can install any cluster manager, each has its own unique advantages depending upon the goal. The proposed model architecture section describes our particular setup.

We used PySpark, the Python Spark API to code our spark application.

B. APACHE KAFKA FRAMEWORK

Apache Kafka[9] provides an abstraction for building a real time data pipeline for streaming application, similar to message queues. The data pipelines reliably get data between systems or applications, and real-time streaming applications transform or react to the streams of data.

Figure 3 below describes the architecture of Kafka [9].

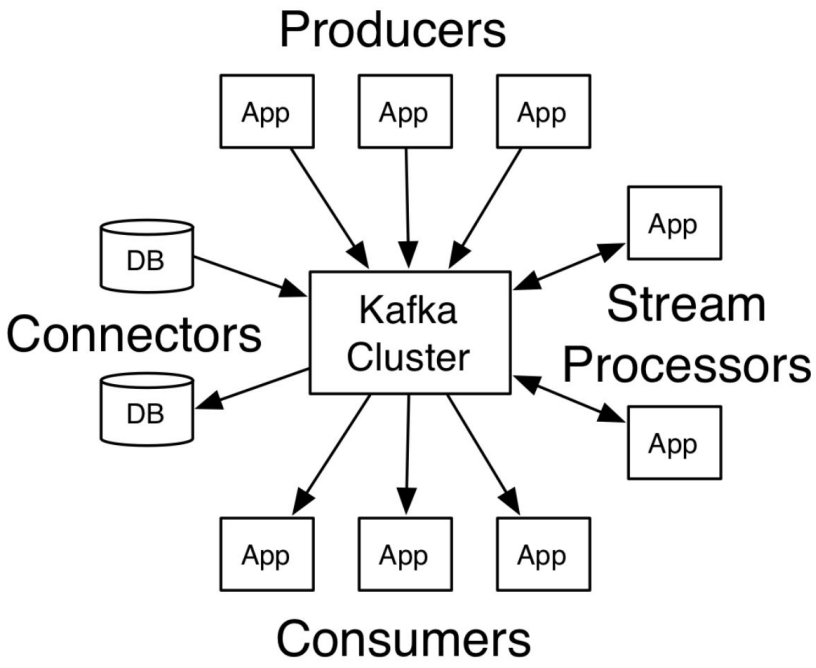

FIGURE 3 : Kafka Architecture

Kafka is run as a cluster on one or more servers, which stores streams of records in categories called topics. There are four key APIs of Kafka:

- The Producer API allows an application to publish a stream of records to one or more Kafka topics.
- The Consumer API allows an application to subscribe to one or more topics and process the stream of records.
- The Streams API allows an application to act as a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.
- The Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems.

Kafka has been used to build user activity tracking pipelines, as a set of real-time publish-subscribe feeds for website activity tracking. [10]

In our implementation we use Kafka as a stream processor, we publish live twitter data to a Kafka topic, which is then processed by the consumer api by subscribing to it. We chose Kafka because compared to most messaging systems Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large scale message processing.

The particular architecture we used in our setup is described in the next section.

PROPOSED ARCHITECTURE

In this subsection, we describe the proposed end-to-end data pipeline architecture for Identification and Sentiment Classification of Trending Twitter Hashtags with scalable models capable of making quick predictions on the live data. Figure 4 below depicts the model architecture.

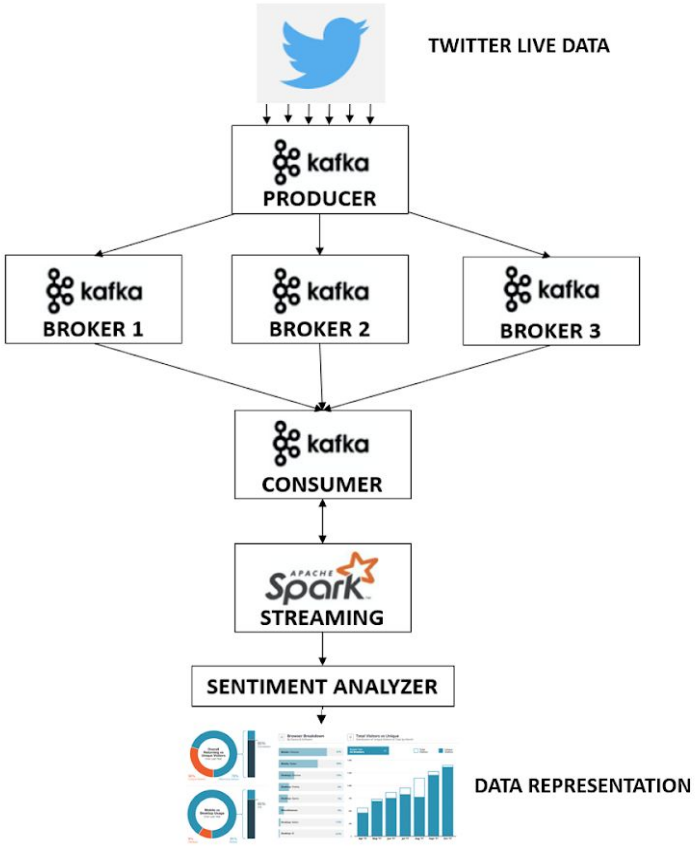

FIGURE 4 : MODEL ARCHITECTURE

Various stages and their explanations are as below,

A. FETCHING LIVE TWITTER DATA :

To obtain live data , Twitter provides the Twitter Streaming API [3]. Major rules involved in streaming the data are ,

- Create a persistent connection to the Twitter API, and read each connection incrementally.

- Process tweets quickly and handle errors and other issues properly.

There are a variety of clients for the Twitter Streaming API across all major programming languages which provide error-free streaming APIs to access the real Time Twitter Data through appropriate Authentication Protocols.

B. KAFKA PRODUCER :

Kafka producer is a client which is majorly responsible in publishing the stream messages to the corresponding Kafka topic. In our model, a single topic is created and live Twitter stream is fetched by the producer client and published to the kafka cluster topic. Our first decision is to make a partition within the topic the smallest unit of parallelism. This means that at any given time, all messages from one partition are consumed only by a single consumer.

C. KAFKA BROKERS :

Since Kafka is distributed in nature, Kafka cluster typically consists of multiple brokers. Once a topic is created for the Twitter Data , it will be partitioned among the broker nodes for load balancing, where each broker would store one or more of these partitions. As shown in the model, we define 3 brokers in our Kafka cluster. To facilitate the coordination among the nodes, we employ a highly available consensus service Zookeeper[4]. A specific Zookeeper node is defined which maintains the metadata of the Kafka cluster nodes.

D. KAFKA CONSUMER :

Kafka consumer subscribes to the topic it wants to listen to and is responsible for retrieving messages published on this topic from various broker partitions . Our model uses a single consumer node which is subscribed to the topic created for Twitter stream data.

E. SPARK APPLICATION AND SENTIMENT ANALYZER :

At the consumer side, we create a Spark streaming application. We create a Spark streaming context to continuously retrieve the streamed messages over batches. These messages are processed using Spark's Map reduce and Transform frameworks. Once we obtain the required data, we process the tweets to obtain

the Trending Hashtags. We also analyze the sentiment of these tweets based on their tweet text using Text Blob.

IMPLEMENTATION:

This section provides details about how the various stages proposed in the model architecture were implemented.

A. ENVIRONMENT DESCRIPTION :

The project development setup consist of following requirements and corresponding packages versions have been enlisted in Table 1.

S.No.	Requirement	Description	Version Details
1.	Twitter Developer account	Twitter account for Tweet feed API usage	NA
2.	Apache Zookeeper	Zookeeper required for configuration, membership, message queuing	Zookeeper version: 3.4.14-4c25d480e66aadd371de8bd2fd8da255ac140bcf
3.	Apache Kafka	Kafka used for processing real time twitter feed.	Kafka version: 2.2.0
4.	Apache Spark & Pyspark	Spark framework used for twitter hashtag sentiment analysis.	Spark version 2.3.3 Pyspark version 2.4.2
5.	Java	JVM dependency for Kafka and Spark installation	Openjdk 11.0.2 2019-01-15

Table 1: Release Version Control

B. ENVIRONMENT SETUP :

We used Google Cloud Platform to simulate our complete model architecture. Four Virtual Machines were created and configured as a Kafka Cluster. As discussed in model, we used 3 Virtual Machines as broker nodes and 1 Virtual

Machine as a Zookeeper node. The producer and consumer scripts were executed in two different broker nodes. The machine configuration of the nodes is described in Table 2.

Node	VM Configuration
Zookeeper	Ubuntu 18.04.2 LTS, 1 vCPU, 4 GB RAM, 10 Standard persistent disk Storage
Broker Node 1 / Producer	Ubuntu 18.04.2 LTS, 1 vCPU, 4 GB RAM, 10 Standard persistent disk Storage
Broker Node 2	Ubuntu 18.04.2 LTS, 1 vCPU, 4 GB RAM, 10 Standard persistent disk Storage
Broker Node 3 / Consumer	Ubuntu 18.04.2 LTS, 1 vCPU, 4 GB RAM, 10 Standard persistent disk Storage

Table 2: System Configuration of Cluster

C. MODEL IMPLEMENTATION :

- **PRODUCER SCRIPT :**

  This includes implementation for getting Live Twitter Data, creating a topic, broker partitions, and feeding to the Kafka Cluster.

  To obtain Twitter data, first using a Valid Twitter account, Twitter API Keys were generated for authenticated developer access for Live Twitter Data.

  OAuthHandler package was used to provide corresponding authentication to live stream of data in the producer script. A python based Twitter Streaming API called **tweepy [5]** was used to fetch the data using the authenticated OAuthHandle. Once we open a connection we'll listen for tweets coming over the connection using a Listener. The Listener class **TweeterStreamListener** defined is a derived class of tweepy.StreamListener class. The on_status(), on_error(), on_timeout() of the base class are overridden and re-defined. Twitter sends Tweets as json objects. The Listener listens over the connection established and obtains these json object data sent by Twitter in its on_status() function. For this project, due to overhead on the Twitter stream data, we specify topic filters

which ensure only those tweets corresponding to those topics are received in the stream.

Once we obtain the data, we send these messages to the Kafka cluster using a producer instance of SimpleProducer. We create a Kafka topic named **"twitterstream"** with a default of single partition configuration and direct the live twitter messages to the Kafka Cluster. Further, the code also includes error handling for cases when connection timeout occurs, when an exception occurs or an error occurs. The code is attached in Appendix.

- **CONSUMER SCRIPT**

  The consumer script implements the spark application for processing received twitter feed and give top trending hashtags along with their sentiments. We have used pyspark API on spark for implementing consumer code.

  First a sparkcontext was created for implementing any spark functionality followed by streaming context creation for connecting to spark cluster for a particular batch duration [6]. Once this is done, kafka direct stream is setup to directly pull messages from Kafka in each batch duration and process without storing. A timer is set up for evaluating batch processing time cumulatively for the course of application runtime. The first phase of code creates a new Dstream that extracts twitter raw data feed. Next using TextBlob, sentiment analysis is performed on each tweet using a pre trained model, creating a new stream of RDD objects, which are tuple of hashtag, count and sentiment here for each tweet [7]. Final step includes filtering out top 10 trending hashtags using sortBy count operation and returning each trending hashtag along with its count and sentiment corresponding to its tweets. Once the last step of spark job is completed, cumulative execution time is evaluated for the batch Dstream and using lazy evaluation, given as output feed to the terminal. The code is attached in Appendix.

D. COMMANDS TO EXECUTE :

The following commands are executed sequentially to bring the entire system up and obtain timely updated on sentiments of Trending Hashtags,

STEP 1 : Start Zookeeper service : bin/zkServer.sh start
STEP 2 : Connect all brokers to Zookeeper : bin/kafka-server-start.sh config/server.properties
STEP 3 : Run producer script : python3 producer.py
STEP 4 : Execute the Spark application in Consumer side : spark-submit --jars spark-streaming-kafka-0-8-assembly_2.11-2.4.2.jar,spark-core_2.11-1.5.2.logging.jar consumer_sentiment_analysis.py

E. SOURCE CODE & DEMO :

The source code is available in the below Github link :
https://github.com/sri-1007/CSCE-678-PROJECT

The demo of complete setup is available in the link :
https://drive.google.com/open?id=17epWAJ_lpYV8rgE6l-JEbMr3S3e0QCML

RESULTS AND ANALYSIS

For evaluation of spark application, twitter data was collected for a duration of 12 hours between April 30th 8:00 PM - May 1st 8:00 AM. Overall hashtag trends along with their count have been shown in Figure 5 and Table 3:

Figure 5: Top trends

Sentiment	Hashtag	
positive	gameofthrones	65,980
	got8	4,644
	uncc	4,068
	avengersendgame	4,001
	aryastark	3,883
	cerseilannister	3,036
	happinessbegins	1,928
	avengers	822
	ucl	484
neutral	ave	240
negative	nottoday	3,147
	nightking	2,339

Table 3: Hashtag trends and Counts

For analyzing pattern in trending hashtags for collected data, a graph is plotted for Log(Hashtag Tweet Count) vs Hashtags as shown in Figure 6. It can be observed that majority of these trending hashtags have a positive sentiment attached to them followed by some negative sentiment trends and finally one neutral hashtag. This reconciles with the fact that most of the tweets are prejudiced towards either a positive or a negative feeling and it is rare to find any neutral statements on social media as seen in Figure 7. Using these statistics, a collaborative attempt can be made along with Twitter to try and suppress these negative trending hashtags and reduce negative feelings among common social media users.

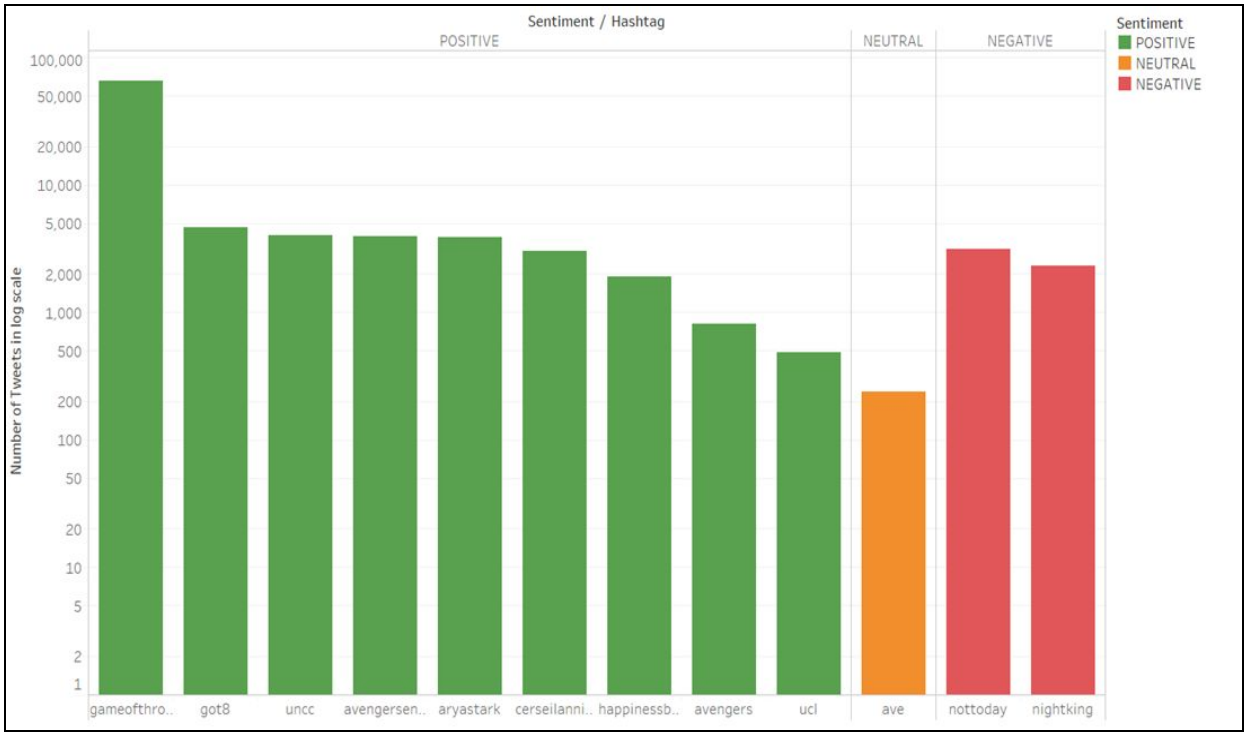

Figure 6: Hashtag Sentiment Analysis

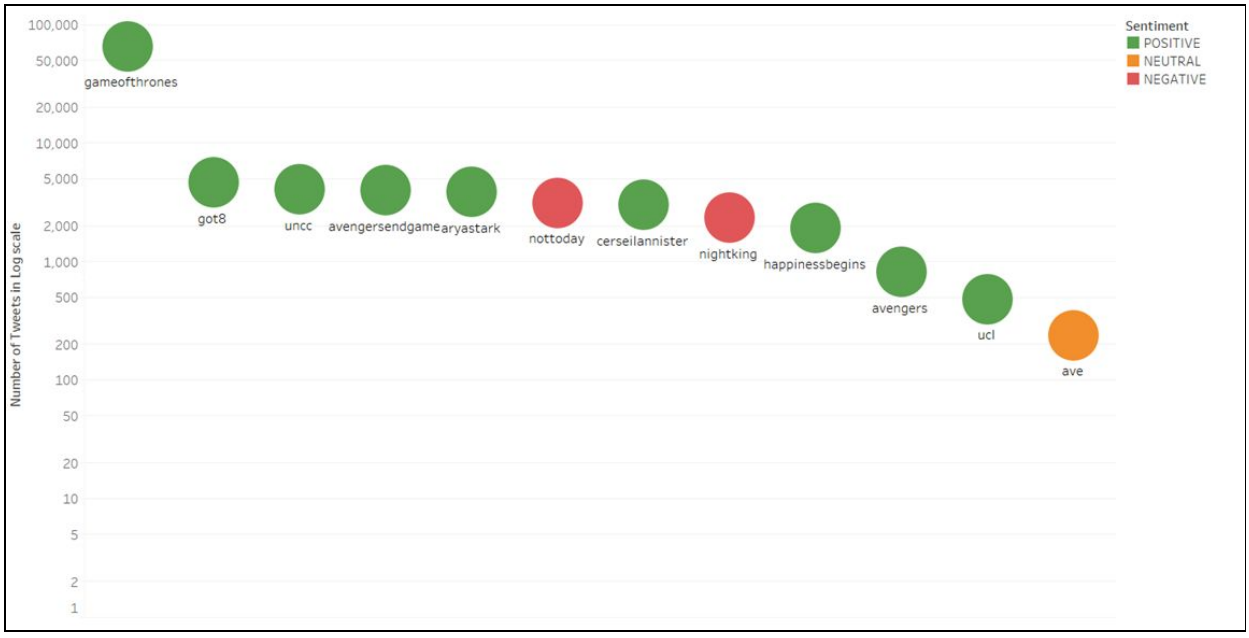

Figure 7: Trend comparison for Sentiments

EVALUATION & DISCUSSION

We evaluated the spark application based on processing time for each batch/interval, which was processing tweets collected in a 30 mins interval.
The Table 4 below presents raw measurements of cumulative processing time.

Batch Number	Cumulative Processing Time (seconds)
Batch 1	8.605
Batch 2	13.865
Batch 3	19.194
Batch 4	24.118
Batch 5	29.253
Batch 6	34.016
Batch 7	38.978
Batch 8	43.752
Batch 9	49.004
Batch 10	**53.641**

Table 4: Cumulative Processing Time till each Batch

The total processing time (filtering, top hashtag extraction, sentiment analysis) for the 10 batches was 53 seconds. The figure below shows the variability of processing time for each batch based on **difference of cumulative time between each batch and last batch.**

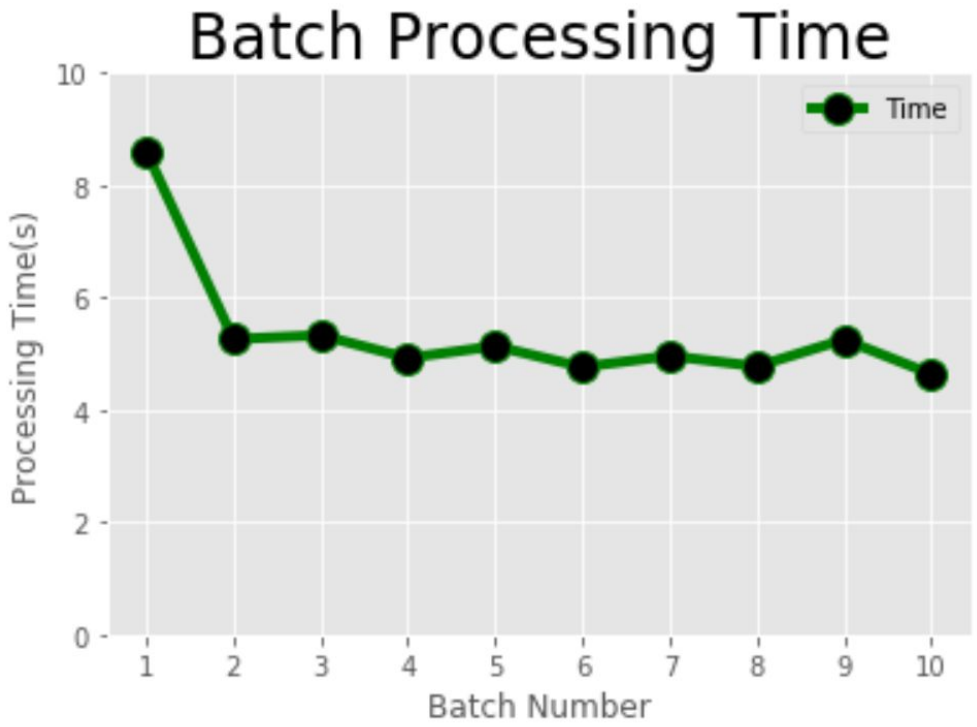

Figure 8: Batch Processing Time

Except for the first batch which takes 8 seconds to process, rest of the batches take around 5 seconds each. The mean is **5.3641s** and standard deviation is **1.103s**. Using this metric, we would characterize the **performance** of the system as **very good**, since we were able to process upwards of **10,000 tweets** in close to **5 seconds**.

The results of the analysis in the last section, square well with our intuitive understanding of what the trends and sentiments should look like. A new 'Game of Thrones' episode had released just around the time we deployed the application. The episode caused a lot of furore online, generating massive traffic. 'Nightking', one of the main antagonists in "Game of Thrones", clearly has an overall negative sentiment, whereas the protagonist 'AryaStark' and the show itself, predictably has an overall positive sentiment.

'Avengers Endgame' was also released around the time, and the movie was well received online, with a clearly positive sentiment. There was also a shooting on UNCC campus around the same time, and though there was short term negative sentiment we observed

in several batches, there were possibly more tweets overall showing support and expressing solidarity, resulting in overall positive sentiment.

Using above observations as indications of **reliability** we would characterize the system as **reasonably reliable in sentiment analysis** and **very reliable in terms of message queuing** service provided by Kafka (strong consistent). Additional improvement in sentiment reliability could be achieved using more advanced machine learning. One bottleneck in our design is the standalone Zookeeper node, in future we would use a cluster for protection in case of its failure.

In terms of **scalability**, adding more brokers to handle an even higher velocity of incoming data is very feasible, and only if we choose to change partition parameters would we need to disrupt the application.

For **security**, we completely depend on the cloud service provider, in our case GCP and this would come with the pitfalls of public cloud security. Our system would be vulnerable to hardware level attacks, side channel attacks etc. In this regard, commercial solutions like Amazon MSK ie Managed Streaming for Kafka and EMR ie Elastic Mapreduce are available as paid enterprise grade services which offer more security. Yet they too are limited in terms of security and a final option could be a private cloud/data centre deployment for critical use cases.
In summary, our evaluation is that this system is reliable, fault tolerant and strongly consistent with moderate security.

CHALLENGES FACED :

Latest version of Apache Spark version **2.4.2** does not support PySpark streaming APIs that are compatible with Kafka **2.2.0**. As a result, we had to downgrade Spark to version **2.3.3**.
Although there are a lot of tutorial available on each Kafka and Spark deployment on Google Cloud Platform, we could not find any reference that exhaustively covers all necessary dependencies for integrating both environments which made it a challenging task.

CONCLUSION

To conclude, we demonstrated a pertinent use case for a commercial application built on production distributed systems. We used live twitter streaming data to monitor trends by filtering top trending hashtags and measured sentiment corresponding to these hashtags. By building a data pipeline in Kafka, and using Spark's stream processing we were able to analyze streaming data, as well as create visualizations based on observed logs, thereby building a production-ready end to end system.

The practical use case for such applications is enormous, for instance in website monitoring and user retention analysis, product/feature launch and sentiment analysis, political campaigns/speech reception among others.

For future work, we would like to focus on using advanced machine learning techniques for the task of sentiment analysis, using the Mllib APIs in PySpark to get even better results.

REFERENCES

[1] Öztürk, Nazan, and Serkan Ayvaz. "Sentiment analysis on Twitter: A text mining approach to the Syrian refugee crisis." Telematics and Informatics 35.1 (2018): 136-147.
[2] Zaharia, Matei, et al. "Fast and interactive analytics over Hadoop data with Spark." Usenix Login 37.4 (2012): 45-51.
[3] Twitter Live Streaming API : https://developer.twitter.com/en/docs
[4] Apache Zookeeper : https://zookeeper.apache.org/
[5] Tweepy : https://tweepy.readthedocs.io/en/latest/
[6] Spark Streaming : https://spark.apache.org/docs/0.7.2/api/streaming/spark/streaming/DStream.html
[7] Spark RDD : https://spark.apache.org/docs/latest/rdd-programming-guide.html
[8] Spark: https://spark.apache.org/docs/2.4.2/
[9] Kafka: https://kafka.apache.org/intro
[10] Kakfa Use Cases: https://kafka.apache.org/uses
[11] Alec Go, Richa Bhayani, and Lei Huang. 2009. Twitter sentiment classification using distant supervision. Technical report, Stanford.
[12] Adam Bermingham and Alan Smeaton. 2010. Classifying sentiment in microblogs: is brevity an advantage is brevity an advantage? ACM, pages 1833–1836
[13] Alexander Pak and Patrick Paroubek. 2010. Twitter as a corpus for sentiment analysis and opinion mining. Proceedings of LREC.
[14] Luciano Barbosa and Junlan Feng. 2010. Robust sentiment detection on twitter from biased and noisy data. Proceedings of the 23rd International Conference on Computational Linguistics: Posters, pages 36–44
[15] Apoorv and Xie, Boyi and Vovsha, Ilia and Rambow, Owen and Passonneau, Rebecca 2011. Sentiment Analysis of Twitter Data. Proceedings of the Workshop on Languages in Social Media.

APPENDIX FOR CODE

A. Kafka producer code -

```
import json
from kafka import SimpleProducer, KafkaClient
import tweepy
import configparser

# Note: Some of the imports are external python libraries. They are installed on the
current machine.
# If you are running multinode cluster, you have to make sure that these libraries
# and currect version of Python is installed on all the worker nodes.

class TweeterStreamListener(tweepy.StreamListener):
    """ A class to read the twitter stream and push it to Kafka"""
    def __init__(self, api):
        self.api = api
        super(tweepy.StreamListener, self).__init__()
        client = KafkaClient("localhost:9092")
        self.producer = SimpleProducer(client, async = True,
                          batch_send_every_n = 1000,
                          batch_send_every_t = 10)

    def on_status(self, status):
        """ This method is called whenever new data arrives from live stream.
        We asynchronously push this data to kafka queue"""
        print('ok')
        msg =  status.text.encode('utf-8')
        try:
            self.producer.send_messages('twitterstream', msg)
        except Exception as e:
            print(e)
            return False
        return True

    def on_error(self, status_code):
        print("Error received in kafka producer",status_code)
        return True # Don't kill the stream

    def on_timeout(self):
        print("Timeout occurred")
        return True # Don't kill the stream
```

```
if __name__ == '__main__':

    # Read the credententials from 'twitter-app-credentials.txt' file
    config = configparser.ConfigParser()
    consumer_key = 'UopwaRDXmyG56bzjgxZwC6o9y'
    consumer_secret = 'j3eFs3PQwPLJGR263viSKutHYhpxP4EF1Sv9MAyevakWDluSVU'
    access_key = '90159436-mhrgUtdfmfeP8yrd1MYgcMtSB27tPPB5u4nPFWqpp'
    access_secret = 'wGjnX9zdk5zl0zfpqhOdxD15mANy5c3tBmz5iAEf8lTSt'

    # Create Auth object
    auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
    auth.set_access_token(access_key, access_secret)
    api = tweepy.API(auth)


    # Create stream and bind the listener to it
           stream   =   tweepy.Stream(auth,   listener   =   TweeterStreamListener(api))
#Note: use verify = False (in case of OpenSSL error)

    #Custom Filter rules pull all traffic for those filters in real time.
                                                        stream.filter(track             =
['Ajax','Avengers','GameofThrones','AryaStark','F8','JonasBrothers','UNCC','Ziyech'],
languages = ['en'])
    #stream.filter(locations = [-122.75,36.8,-121.75,37.8], languages = ['en'])
```

B. Kafka consumer code -

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
import json
import sys
import os
from textblob import TextBlob
import timeit
import time
if __name__ == "__main__":

    sc = SparkContext(appName="TweetCount")
    ssc = StreamingContext(sc, 1800)
    topics = ["twitterstream"]
    params = {"bootstrap.servers":"localhost:9092"}
    kafkaStream = KafkaUtils.createDirectStream(ssc,topics,params)
```

```
acc=sc.accumulator(0)
parsed = kafkaStream.map(lambda v: v[1])

def get_sentiment(tweet):

    start=time.time()
    analysis=TextBlob(tweet)
    end=time.time()
    acc.add(end-start)

    if analysis.sentiment.polarity>0:
        return 1
    elif analysis.sentiment.polarity==0:
        return 0
    else:
        return -1

def sentiment(tup):
    start=time.time()
    hashs,sentiment = tup
    count,sen = sentiment
    end=time.time()
    acc.add(end-start)
    if sen >= 1:
        return (hashs,count,'positive')
    elif sen == 0:
        return (hashs,count,'neutral')
    else:
        return (hashs,count,'negative')

def fun(tweet):
    start=time.time()
    hashes = tweet.split(' ')
    trend = [x for x in hashes if x.startswith('#')]
    end=time.time()
    acc.add(end-start)
    if trend:
        return (str(trend[0]).lower(),(1,get_sentiment(tweet)))
    return (str(trend),(0,0))

def trending(rdd):
    start=time.time()
    m = rdd.sortBy(lambda x: x[1], ascending=False).take(10)
    end=time.time()
    acc.add(end-start)
    return rdd.filter(lambda x: x in m)

hashtags = parsed.map(lambda tweet: fun(tweet))
```

```
    hashtagsCounts = hashtags.reduceByKey(lambda x, y: (x[0] + y[0], x[1]+y[1]))
      temp_trending = hashtagsCounts.transform(lambda rdd: rdd.sortBy(lambda x: x[1],
ascending=False))
    trendinghashtags = temp_trending.transform(lambda rdd: trending(rdd))
    hashtagsSentiments = trendinghashtags.map(lambda rdd: sentiment(rdd))
      time_time=hashtagsSentiments.transform(lambda rdd: print("Cumulative Processing
time is ({0}s)".format(acc.value)) or rdd)
    time_time.pprint()

    ssc.start()
    ssc.awaitTermination()
```

C. Zookeeper Configuration

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=/tmp/zookeeper
# the port at which the clients will connect
clientPort=2181
```

D. Kafka Configuration (Broker 1)

```
# see kafka.server.KafkaConfig for additional details and defaults

############################# Server Basics #############################

# The id of the broker. This must be set to a unique integer for each broker.
broker.id=0

############################# Socket Server Settings #############################
```

```
############################# Zookeeper #############################

# Zookeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify the
# root directory for all kafka znodes.
zookeeper.connect=10.128.0.16:2181

# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=6000
```

E. Kafka Configuration (Broker 2)

```
############################# Server Basics #############################

# The id of the broker. This must be set to a unique integer for each broker.
broker.id=1
```

```
############################# Zookeeper #############################

# Zookeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify the
# root directory for all kafka znodes.
zookeeper.connect=10.128.0.16:2181

# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=6000
```

F. Kafka Configuration (Broker 3)

```
# see kafka.server.KafkaConfig for additional details and defaults

############################# Server Basics #############################

# The id of the broker. This must be set to a unique integer for each broker.
broker.id=2
```

```
############################# Zookeeper #############################

# Zookeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify the
# root directory for all kafka znodes.
zookeeper.connect=10.128.0.16:2181

# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=6000
```

G. Spark and Java environment setup

```
export JAVA_HOME="/usr/lib/jvm/java-11-openjdk-amd64"
export SPARK_HOME="/home/harmeet_nith/spark"
export PATH="$SPARK_HOME/bin:$PATH"


export SCALA_HOME="/home/harmeet_nith/scala"
export PATH="$PATH:$SCALA_HOME/bin"
```

```
export PYSPARK_PYTHON=python3
export PYSPARK_DRIVER_PYTHON=python:3
```