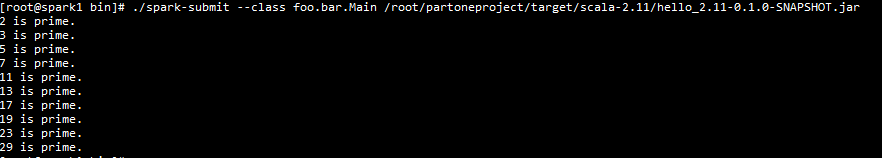
Nishant Velagapudi

W251 – Hw 9

**Part 1:**

The simple application prints out prime numbers between 0 and 30:



**Part 2:**

**Please connect to 169.53.143.170**

To run the twitter streaming application, please run:

**cd ~/twitter\_popularity**

**sbt package && $SPARK\_HOME/bin/spark-submit --packages org.apache.bahir:spark-streaming-twitter\_2.11:2.1.0 --master spark://spark1:7077 $(find target -iname "\*.jar") arg1 arg2 arg3**

Where:

arg1 = Number of popular topics to output in short and long windows (set for 10 in screenshot run)

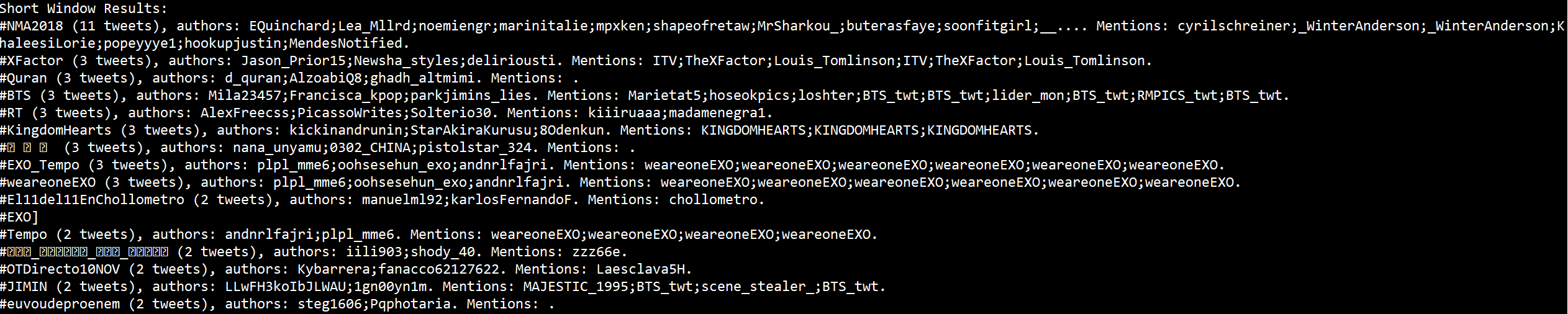
arg2 = short window duration (seconds) (set for 90 in screenshot run)

arg3 = long window duration seconds) (set for 1800 in screenshot run)

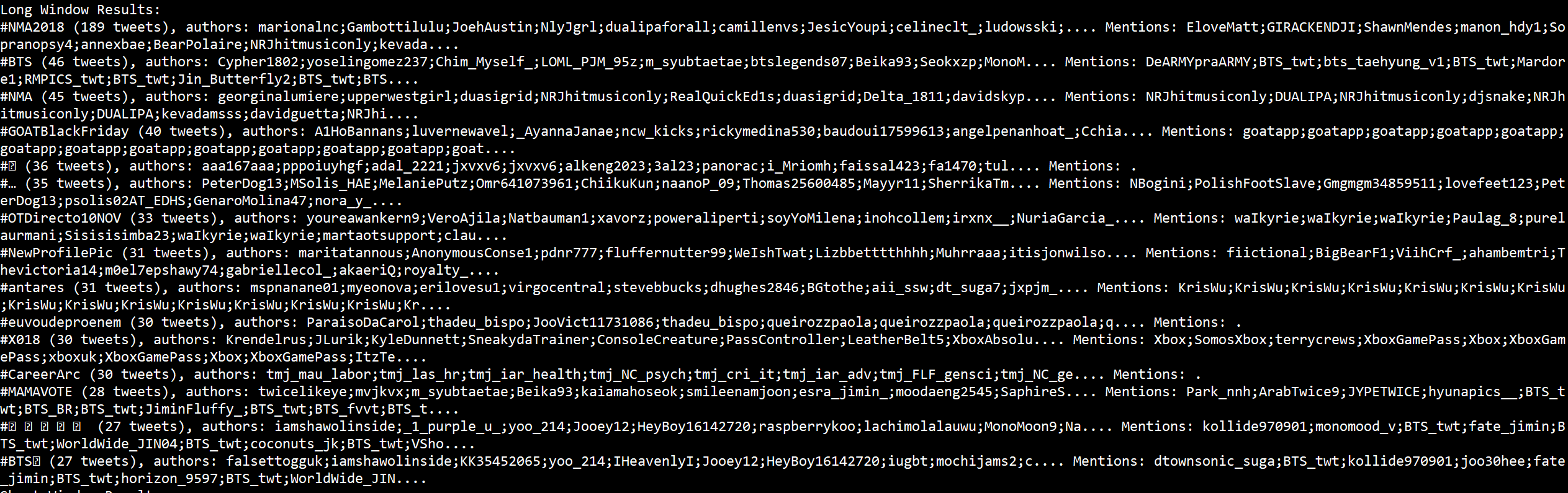
One key design decision is to cache the RDD dstream and then parse it twice to achieve the requirement of short and long windows over the same data. The first reduce by key and window will produce short window aggregates, while the second will have a running total up until the long window length thus saving compute. I also used a topic -> tuple(topic, mentions, authors) structure. I kept mentions and authors as arrays mapped to each topic and then just printed using mkString to concatenate individual elements as semicolon delimited.

I ran the script for a short window of 100 seconds and a long window of 30 minutes (1800 seconds). For each popular hashtag, authors and mentions are printed next. The format is hashtag – authors – mentioned users. The authors and users arrays from the container object are truncated to 100 characters for readability.

Short window capture (100 seconds):



30 minute results capture:



**Twitter\_popularity.scala:**

package streaming.twitter

import org.apache.spark.streaming.{Seconds, StreamingContext}

import org.apache.spark.SparkContext.\_

import org.apache.spark.streaming.twitter.\_

import org.apache.spark.SparkConf

import scala.collection.mutable.ListBuffer

import org.apache.log4j.{Level, Logger}

object Main extends App {

//args

//0 - how many hashtags to use

//1 - Short sampling duration (seconds)

//2 - long sampling duration (seconds)

//define class for holding relevant data to the task

//we want the topic of interest ("hashtag"), the related users, and related authors

//thus, each hashtag linked with collection of users and collection of authors

case class HashTagSummary(hashtag:String, users: Array[String], authors:Array[String]) {

def +(add:HashTagSummary) : HashTagSummary = {

new HashTagSummary(hashtag, users++add.users, authors++add.authors)

}

}

//attempt to get rid of overwhelming number of INFO level logs

val rootLogger = Logger.getRootLogger()

rootLogger.setLevel(Level.ERROR)

val filters = Array[String]()

// get a Twitter stream

// note: Twitter credentials are in twitter4j.properties

val sparkConf = new SparkConf().setAppName("twitter\_popularity")

val context = new StreamingContext(sparkConf, Seconds(args(1).toInt))

val stream = TwitterUtils.createStream(context, None, filters)

//parse out hashtags, map into the defined container class

//This will window by all hashtags and by the short sampling duration

val hashtags = stream.flatMap(raw => {

val topic = raw.getText.split(" ").filter(\_.startsWith("#"))

topic.map(single => (single, new HashTagSummary(single,

raw.getUserMentionEntities.map(\_.getScreenName),

Array(raw.getUser.getScreenName))))

})

//in the compute graph, hashtags should be used twice. once for calculating tags over a short window, once for long window

//thus, trying to save it to memory for the sake of performance

hashtags.cache()

val shortTags = hashtags.reduceByKeyAndWindow(\_ + \_, Seconds(args(1).toInt))

val longTags = hashtags.reduceByKeyAndWindow(\_ + \_, Seconds(args(2).toInt))

//since we can only sort by key, re-arrange the tuple to be keyed on occurrences

//use the length of authors who wrote a tweet including topic as proxy to count number of tweets

val HashTagCounts\_short = shortTags.map{case (topic, tupleSummary) =>

(tupleSummary.authors.length, tupleSummary)

}

val HashTagCounts\_long = longTags.map{case (topic, tupleSummary) =>

(tupleSummary.authors.length, tupleSummary)

}

//sort the RDD so that most often occurring hashtags are at the top

val sortedHashTagCounts\_short = HashTagCounts\_short.transform(\_.sortByKey(false))

val sortedHashTagCounts\_long = HashTagCounts\_long.transform(\_.sortByKey(false))

//output the requested number of hashtags + details to console

sortedHashTagCounts\_short.foreachRDD(rdd => {

val topList = rdd.take(args(0).toInt)

System.out.println("Short Window Results: ")

topList.foreach{case (count, summary) => {

var printAuthors = summary.authors.mkString(";")

var printUsers = summary.users.mkString(";")

if(printAuthors.length > 100) {

printAuthors = printAuthors.substring(0, 100).concat("...")

}

if(printUsers.length > 100) {

printUsers = printUsers.substring(0,100).concat("...")

}

System.out.println("%s (%s tweets), authors: %s. Mentions: %s.".format(summary.hashtag, count, printAuthors, printUsers))

}}

})

sortedHashTagCounts\_long.foreachRDD(rdd => {

val topList = rdd.take(args(0).toInt)

System.out.println("Long Window Results: ")

topList.foreach{case (count, summary) => {

var printAuthors = summary.authors.mkString(";")

var printUsers = summary.users.mkString(";")

if(printAuthors.length > 100) {

printAuthors = printAuthors.substring(0, 100).concat("...")

}

if(printUsers.length > 100) {

printUsers = printUsers.substring(0,100).concat("...")

}

System.out.println("%s (%s tweets), authors: %s. Mentions: %s.".format(summary.hashtag, count, printAuthors, printUsers))

}}

})

context.start()

context.awaitTermination()

context.stop()

}