**Popular movies:**

object PopularMovies {

/\*\* Our main function where the action happens \*/

def main(args: Array[String]) {

// Set the log level to only print errors

Logger.getLogger("org").setLevel(Level.ERROR)

// Create a SparkContext using every core of the local machine

val sc = new SparkContext("local[\*]", "PopularMovies")

// Read in each rating line

val lines = sc.textFile("../ml-100k/u.data")

// Map to (movieID, 1) tuples

val movies = lines.map(x => (x.split("\t")(1).toInt, 1))

// Count up all the 1's for each movie

val movieCounts = movies.reduceByKey( (x, y) => x + y )

// Flip (movieID, count) to (count, movieID)

val flipped = movieCounts.map( x => (x.\_2, x.\_1) )

// Sort

val sortedMovies = flipped.sortByKey()

// Collect and print results

val results = sortedMovies.collect()

results.foreach(println)

}

}

**Popular superhero from marvels:**

/\*\* Find the superhero with the most co-appearances. \*/

object MostPopularSuperhero {

// Function to extract the hero ID and number of connections from each line

def countCoOccurences(line: String) = {

var elements = line.split("\\s+")

( elements(0).toInt, elements.length - 1 )

}

// Function to extract hero ID -> hero name tuples (or None in case of failure)

def parseNames(line: String) : Option[(Int, String)] = {

var fields = line.split('\"')

if (fields.length > 1) {

return Some(fields(0).trim().toInt, fields(1))

} else {

return None // flatmap will just discard None results, and extract data from Some results.

}

}

/\*\* Our main function where the action happens \*/

def main(args: Array[String]) {

// Set the log level to only print errors

Logger.getLogger("org").setLevel(Level.ERROR)

// Create a SparkContext using every core of the local machine

val sc = new SparkContext("local[\*]", "MostPopularSuperhero")

// Build up a hero ID -> name RDD

val names = sc.textFile("../marvel-names.txt")

val namesRdd = names.flatMap(parseNames)

// Load up the superhero co-apperarance data

val lines = sc.textFile("../marvel-graph.txt")

// Convert to (heroID, number of connections) RDD

val pairings = lines.map(countCoOccurences)

// Combine entries that span more than one line

val totalFriendsByCharacter = pairings.reduceByKey( (x,y) => x + y )

// Flip it to # of connections, hero ID

val flipped = totalFriendsByCharacter.map( x => (x.\_2, x.\_1) )

// Find the max # of connections

val mostPopular = flipped.max()

// Look up the name (lookup returns an array of results, so we need to access the first result with (0)).

val mostPopularName = namesRdd.lookup(mostPopular.\_2)(0)

// Print out our answer!

println(s"$mostPopularName is the most popular superhero with ${mostPopular.\_1} co-appearances.")

}

}

**Spark Streaming:**

**object** PopularHashtags {

/\*\* Makes sure only ERROR messages get logged to avoid log spam. \*/

**def** setupLogging() = {

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

**val** rootLogger = Logger.getRootLogger()

rootLogger.setLevel(Level.ERROR)

}

/\*\* Configures Twitter service credentials using twiter.txt in the main workspace directory \*/

**def** setupTwitter() = {

**import** scala.io.Source

**for** (line <- Source.fromFile("../twitter.txt").getLines) {

**val** fields = line.split(" ")

**if** (fields.length == 2) {

System.setProperty("twitter4j.oauth." + fields(0), fields(1))

}

}

}

/\*\* main function where the action happens \*/

**def** main(args: Array[String]) {

// Configure Twitter credentials using twitter.txt

setupTwitter()

// Set up a Spark streaming context named "PopularHashtags" that runs locally using

// all CPU cores and one-second batches of data

**val** ssc = **new** StreamingContext("local[\*]", "PopularHashtags", Seconds(1))//processing info 1 sec at a time

// Get rid of log spam (should be called after the context is set up)

setupLogging()

// Creating a receiver that listens to the stream of tweets d-stream:continuous sequence of RDDs representing a continuous stream of data

**val** tweets = TwitterUtils.createStream(ssc, None)

// Now extract the text of each status update into DStreams using map()

**val** statuses = tweets.map(status => status.getText())

//each word into a new DStream

**val** tweetwords = statuses.flatMap(tweetText => tweetText.split(" "))

// Now eliminate anything that's not a hashtag

**val** hashtags = tweetwords.filter(word => word.startsWith("#"))

// Map each hashtag to a key/value pair of (hashtag, 1) so we can count them up by adding up the values

**val** hashtagKeyValues = hashtags.map(hashtag => (hashtag, 1))

// Now count them up over a 5 minute window sliding every one second

**val** hashtagCounts = hashtagKeyValues.reduceByKeyAndWindow( (x,y) => x + y, (x,y) => x - y, Seconds(300), Seconds(1)

// Sort the results by the count values

**val** sortedResults = hashtagCounts.transform(rdd => rdd.sortBy(x => x.\_2, **false**))

sortedResults.print

// Set a checkpoint directory, and kick it all off

ssc.checkpoint("C:/checkpoint/")

ssc.start()

ssc.awaitTermination()

}

}