Hands-on 3:
Stream processing

1. Objective

The objective of this hands on is to let you "touch" the challenges implied in processing streams.

- In class we will use Spark for implementing a streaming version of word count and an example using Twitter streaming.
- At **HOME** you will
  - test other Spark functions including machine learning and graph algorithms with streams.
  - Implement Bloom filter in Hadoop.

2. Getting started with Spark streaming

Spark streaming is an extension of the core Spark API that enables stream processing of live data streams. Data can be harvested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to file systems, databases, and live dashboards[1].

Internally Spark streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.

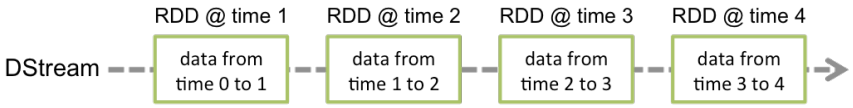
Spark Streaming is based on the notion of *discretized stream* or *DStream*, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.

2.1 Basic concepts

2.1.1 Discretized streams (DStreams)

Discretized Stream is the basic abstraction provided by Spark Streaming. It represents a continuous stream of data, either the input data stream received from source, or the processed data stream generated by transforming the input stream. Internally, a DStream is represented by a continuous series of RDDs, which is Spark's abstraction of an immutable, distributed dataset. Each RDD in a DStream contains data from a certain interval, as shown in the following figure.

[1] http://spark.apache.org/docs/latest/streaming-programming-guide.html#overview



Any operation applied on a DStream translates to operations on the underlying RDDs. These underlying RDD transformations are computed by the Spark engine. The DStream operations hide most of these details and provide the developer with a higher-level API for convenience.

2.1.2 Input DStreams and Receivers

Input DStreams are DStreams representing the stream of input data received from streaming sources. Every input DStream (except file stream) is associated with a **Receiver** object which receives the data from a source and stores it in Spark's memory for processing. Spark Streaming provides two categories of built-in streaming sources.

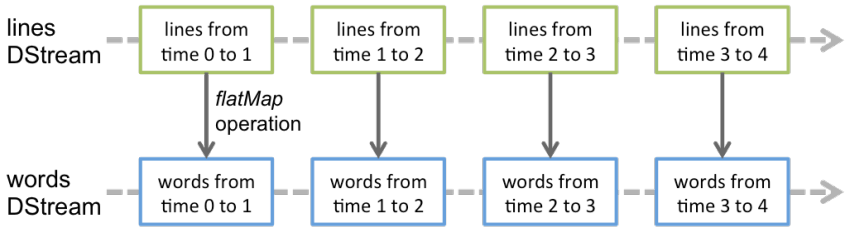
- *Basic sources*: directly available in the StreamingContext API. Examples: file systems, socket connections, etc.
- *Advanced sources*: Sources like Kafka, Flume, Kinesis, Twitter, etc. are available through extra utility classes. These require linking against extra dependencies (see the Twitter exercise next).

Note that, if you want to receive multiple streams of data in parallel in your streaming application, you can create multiple input DStreams. This will create multiple receivers which will simultaneously receive multiple data streams. Yet, note that a Spark worker/executor is a long-running task, hence it occupies one of the cores allocated to the Spark Streaming application. Therefore, it is important to remember that a Spark Streaming application needs to be allocated enough cores (or threads, if running locally) to process the received data, as well as to run the receiver(s).

2.2 Counting words from streams

We want to count the number of words in text data received from a data server listening on a TCP socket.

Note that for converting a stream of lines to words, the flatMap operation is applied on each RDD in the lines DStream to generate the RDDs of the words DStream as shown in the following figure.



2.2.1 Importing Spark Streaming classes and conversions from `StreamingContext`

`StreamingContext` is the main entry point for all streaming functionality. We create a local `StreamingContext` with two execution threads, and a batch interval of 1 second.

```
import org.apache.spark._
```

```scala
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._ // not necessary since Spark 1.3

// Create a local StreamingContext with two working thread and batch interval of 1 second.
// The master requires 2 cores to prevent from a starvation scenario.

val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

2.2.2 Creating a DStream

Using this context, we can create a `DStream` that represents streaming data from a TCP source, specified as hostname (e.g. `localhost`) and port (e.g. `9999`).

```scala
// Create a DStream that will connect to hostname:port, like localhost:9999

val lines = ssc.socketTextStream("localhost", 9999)
```

This `lines` DStream represents the stream of data that will be received from the data server. Each record in this DStream is a line of text. Next, we want to split the lines by space characters into words.

```scala
// Split each line into words
val words = lines.flatMap(_.split(" "))
```

`flatMap` is a one-to-many DStream operation that creates a new DStream by generating multiple new records from each record in the source DStream. In this case, each line will be split into multiple words and the stream of words is represented as the words DStream.

2.2.3 Counting words

Next, we want to count these words.

```scala
import org.apache.spark.streaming.StreamingContext._ // not necessary since Spark 1.3
// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

// Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.print()
```

The `words` DStream is further mapped (one-to-one transformation) to a DStream of `(word, 1)` pairs, which is then reduced to get the frequency of words in each batch of data. Finally, `wordCounts.print()` will print a few of the counts generated every second.

Note that when these lines are executed, Spark Streaming only sets up the computation it will perform when it is started, and no real processing has started yet. To start the processing after all the transformations have been setup, we finally call

```
ssc.start()             // Start the computation
ssc.awaitTermination()  // Wait for the computation to terminate
```

The complete code can be found in the Spark Streaming example NetworkWordCount.

2.2.4 Running the program

You can run this example as follows. You will first need to run Netcat (a small utility found in most Unix-like systems) as a data server by using

```
$ nc -lk 9999
```

Then, in a different terminal, you can start the example by using

```
$ ./bin/run-example streaming.NetworkWordCount localhost 9999
```

Then, any lines typed in the terminal running the `netcat` server will be counted and printed on screen every second. It will look something like the following:

```
# TERMINAL 1:
# Running Netcat

$ nc -lk 9999

hello world



...
```

```
# TERMINAL 2: RUNNING NetworkWordCount

$ ./bin/run-example streaming.NetworkWordCount localhost 9999
...
-------------------------------------------
Time: 1357008430000 ms
-------------------------------------------
(hello,1)
(world,1)
...
```

2.3 Technical considerations

2.3.1 Linking

Similar to Spark, Spark Streaming is available through Maven Central. To write your own Spark Streaming program, you will have to add the following dependency your Maven project.

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.10</artifactId>
    <version>1.5.2</version>
</dependency>
```

For harvesting data from sources like Kafka, Flume, and Kinesis that are not present in the Spark Streaming core API, you will have to add the corresponding artifact `spark-streaming-xyz_2.10` to the dependencies. For an up-to-date list, please refer to the Maven repository for the full list of supported sources and artifacts.

2.3.2 Initializing StreamingContext

To initialize a Spark Streaming program, a **`StreamingContext`** object has to be created which is the main entry point of all Spark Streaming functionality. A `StreamingContext` object can be created from a `SparkConf` object.

```scala
import org.apache.spark._
import org.apache.spark.streaming._

val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

The `appName` parameter is a name for your application to show on the cluster UI. `master` is a Spark, Mesos or YARN cluster URL, or a special "**`local[*]`**" string to run in local mode.

In practice, when running on a cluster, you will not want to hardcode `master` in the program, but rather launch the application with `spark-submit` and receive it there. However, for local testing and unit tests, you can pass "`local[*]`" to run Spark Streaming in-process (detects the number of cores in the local system). Note that this internally creates a `SparkContext` (starting point of all Spark functionality) which can be accessed as `ssc.sparkContext`.

The batch interval must be set based on the latency requirements of your application and available cluster resources.

A `StreamingContext` object can also be created from an existing `SparkContext` object.

```scala
import org.apache.spark.streaming._

val sc = ...                // existing SparkContext
val ssc = new StreamingContext(sc, Seconds(1))
```

After a context is defined, you have to do the following.

1. Define the input sources by creating input `DStreams`.
2. Define the streaming computations by applying transformation and output operations to `DStreams`.
3. Start receiving data and processing it using `streamingContext.start()`.
4. Wait for the processing to be stopped (manually or due to any error) using `streamingContext.awaitTermination()`.
5. The processing can be manually stopped using `streamingContext.stop()`.

Points to remember:

- Once a context has been started, no new streaming computations can be set up or added to it.
- Once a context has been stopped, it cannot be restarted.
- Only one `StreamingContext` can be active in a JVM at the same time.
- `stop()` on `StreamingContext` also stops the `SparkContext`. To stop only the `StreamingContext`, set the optional parameter of `stop()` called `stopSparkContext` to **`false`**.
- A `SparkContext` can be re-used to create multiple `StreamingContexts`, as long as the previous `StreamingContext` is stopped (without stopping the `SparkContext`) before the next `StreamingContext` is created.

2.4. Processing Twitter streams

These exercises are designed as standalone Scala programs which will receive and process Twitter's real sample tweet streams. This section will first introduce you to the basic system setup of the standalone Spark Streaming programs, and then guide you through the steps necessary to create Twitter authentication tokens necessary for processing Twitter's real time sample stream. This hands-on corresponds to the one proposed by Databricks[2] with some variants.

2.4.1 System setup

You should find the following items in the directory [usb root directory]/streaming/. For Scala users:

- `scala/build.sbt`: SBT project file
- `scala/Tutorial.scala`: Main Scala program that you are going to edit, compile and run
- `scala/TutorialHelper.scala`: Scala file containing few helper functions for Tutorial.scala
- `getCheckpointDirectory()` is a helper function that figures out the checkpoint directory that Spark Streaming should use.
- `configureTwitterCredential()` is another helper function that configures Twitter's authentication detail using the file `/root/streaming/twitter.txt`

2.4.2 Twitter credential setup

Our hand-on is based on Twitter's sample tweet stream, so we need to configure authentication with a Twitter account. To do this, you need to setup a consumer `key+secret` pair and an access `token+secret` pair using a Twitter account.

Creating a temporary Twitter access keys

Follow the instructions below to setup these temporary access keys with your Twitter account. These instructions will not require you to provide your Twitter username/password. You will only be required to provide the consumer key and access token pairs that you will generate, which you can easily destroy once you have finished the tutorial. So, your Twitter account will not be compromised in any way.

Open Twitter's Application Settings page. This page lists the set of Twitter-based applications that you own and have already created consumer keys and access tokens for. This list will be empty if you have never created any applications. For this tutorial, create a new temporary application. To do this, click on the blue "*Create a new application*" button. The new application page should look the page shown below.

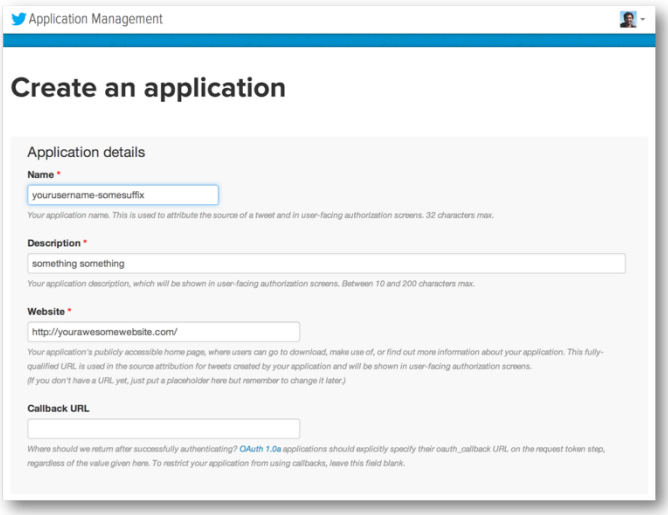
Provide the required fields:

- The **`Name`** of the application must be globally unique, so using your Twitter username as a prefix to the name should ensure that. For example, set it as `[your-twitter-handle]-test`.
- For the **`Description`** , anything longer than 10 characters is fine.
- For the **`Website`** , similarly, any website is fine, but ensure that it is a fully-formed URL with the prefix *`http://`* .

Then, click on the "*Yes, I agree*" checkbox below the **`Developer Rules of the Road`**. Finally, fill in the `CAPTCHA` and click on the blue "*Create your Twitter application*" button.

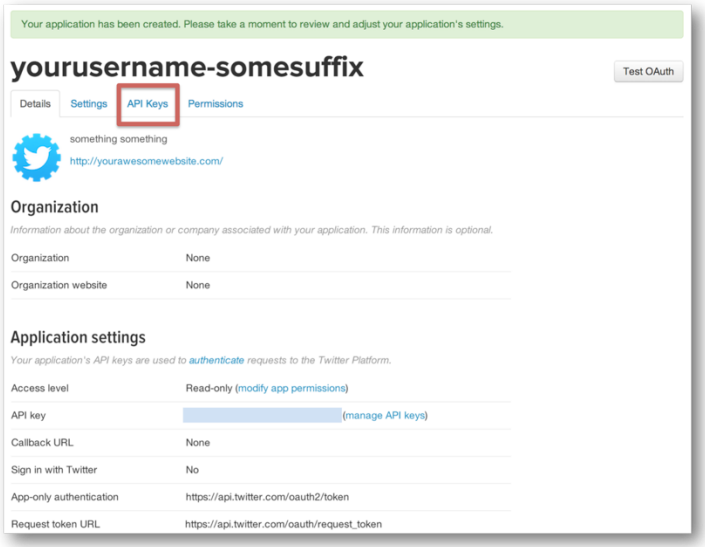
---

[2] https://databricks-training.s3.amazonaws.com/realtime-processing-with-spark-streaming.html



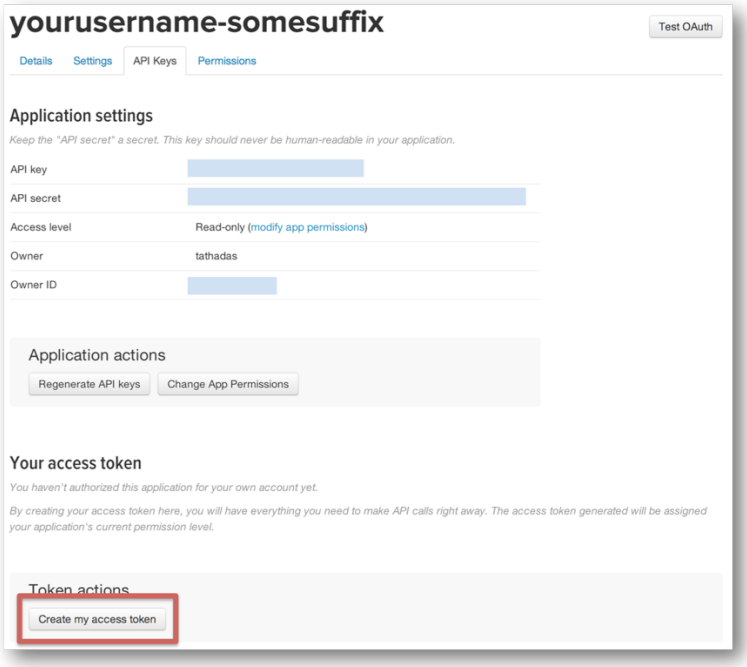
Confirmation

Once you have created the application, you will be presented with a confirmation page similar to the one shown below. Click on the **`API Key`** tab.



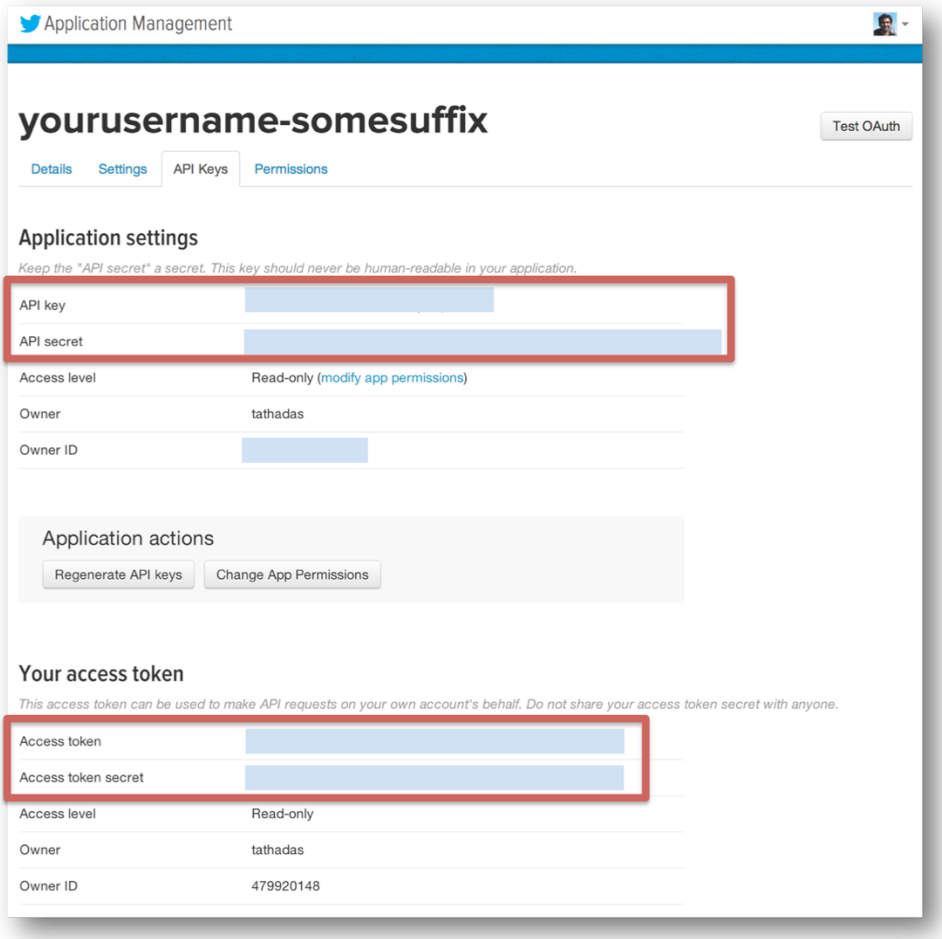
Application settings

You should be able to see the API key and the API secret that have been generated. To generate the access token and the access token secret, click on the "*Create my access token*" button at the bottom of the page (marked in red in the figure below). Note that there will be a small green confirmation at the top of the page saying that the token has been generated.



Final result of the setup process

Finally, the page should look like the following. Notice the API Key, API Secret, Access Token and Access Token Secret. We are going to use these 4 keys in the next section.



2.4.3 First Spark streaming program

Let us write a very simple Spark Streaming program that prints a sample of the tweets it receives from Twitter every second.

Add your authentication keys

First add the authentication keys that you generated in the previous step. After entering the keys in the corresponding strings, your file should look something like this.

```
    // Configure Twitter credentials
    val consumerKey = "//Your credentials go here"
    val consumerSecret = "//Your credentials go here"
    val accessToken = "//Your credentials go here"
    val accessTokenSecret = "//Your credentials go here"
```

Setup the system properties for using Twitter4j libraries

```
// Set the system properties so that Twitter4j library used by twitter
stream
// can use them to generat OAuth credentials
    System.setProperty("twitter4j.oauth.consumerKey", consumerKey)
    System.setProperty("twitter4j.oauth.consumerSecret",
consumerSecret)
    System.setProperty("twitter4j.oauth.accessToken", accessToken)
    System.setProperty("twitter4j.oauth.accessTokenSecret",
accessTokenSecret)
```

Create a StreamingContext

To express any Spark Streaming computation, a `StreamingContext` object needs to be created. This object serves as the main entry point for all Spark Streaming functionality. For the rest of this exercise, insert the code samples after "`//Your Code goes here`" placeholder.

```
val ssc = new StreamingContext(new SparkConf(), Seconds(1))
```

We created a `StreamingContext` object by providing a Spark configuration, and the batch duration we'd like to use for streams. Next, we use this context to create a stream of tweets:

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

The object tweets is a DStream of tweet statuses. More specifically, it is continuous stream of RDDs containing objects of type `twitter4j.Status`. As a very simple processing step, let us try to print the status text of the some of the tweets.

```
val statuses = tweets.map(status => status.getText())
// Print first 10 records in theDStream (1 second batches of received
status)
statuses.print()
```

Using DStreams

Similar to RDD transformation in the earlier Spark exercises, the map operation on tweets maps each Status object to its text to create a new 'transformed' DStream named statuses. The print output operation tells the context to print first 10 records in each RDD in a DStream, which in this case are 1 second batches of received status texts.

Finally, we need to tell the context to start running the computation we have setup.

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

Note that all DStream operations must be done before calling this statement.

Running Tweets consumption

After saving `Tutorial.scala`, let us compile the code. To do this, follow these instructions. **Make sure you replace `[usb root directory]` in the following instructions with your actual USB root directory path.**

In the console, make sure you are in the directory:

```
[usb root directory]/streaming/scala . ../../sbt/sbt assembly
```

This command will compile the `Tutorial class` and create a JAR file in :

```
[usb root directory]/streaming/scala/target/scala-2.10/.
```

Finally, the program can be executed as using the spark-submit script. At the command line, run the following. Again, **make sure you replace `[usb root directory]` with your actual USB root directory path.**

```
../../spark/bin/spark-submit --class Tutorial
../../streaming/scala/target/scala-2.10/Tutorial-assembly-0.1-
SNAPSHOT.jar
```

You will soon find a sample of the received tweets being printed on the screen (can take 10 seconds or so before it start appearing). Use `Ctrl + c`, to stop the application.

2.4.4 Further mini-challenges

Let us try printing the 10 most popular hashtags in the last 5 minutes. These next steps explain the set of the `DStream` operations required to achieve our goal. As mentioned before, the operations explained in the next steps must be added in the program before `ssc.start()`. After every step, you can see the contents of new `DStream` you created by using the `print()` operation and running the program in the same way as explained earlier.

Get the stream of hashtags from the stream of tweets

To get the hashtags from the status string, we need to identify only those words in the message that start with "#". This can be done as follows:

```
val words = statuses.flatMap(status => status.split(" "))
val hashtags = words.filter(word => word.startsWith("#"))
```

The `flatMap` operation applies a one-to-many operation to each record in a `DStream` and then flattens the records to create a new `DStream`. In this case, each status string is split by space to produce a `DStream` where each record is a word. Then we apply the `filter` function to retain only

the hashtags. The resulting hashtags `DStream` is a stream of RDDs having only the hashtags. If you want to see the result, add `hashtags.print()` and try running the program.

Count the hashtags over a 5 minutes window

We would like to count these hashtags over a 5 minute moving window. A simple way to do this would be to gather together the last 5 minutes of data and process it in the usual map-reduce way:

- map each tag to a `(tag, 1)` key-value pair and then
- reduce by adding the counts.

However, in this case, counting over a sliding window can be done more intelligently. As the window moves, the counts of the new data can be added to the previous window's counts, and the counts of the old data that falls out of the window can be 'subtracted' from the previous window's counts. This can be done using `DStreams`.

```
val counts = hashtags.map(tag => (tag, 1))
                     .reduceByKeyAndWindow(_ + _, _ - _, Seconds(60 *
5), Seconds(1))
```

- The `_ + _` and `_ - _` are Scala shorthands for specifying functions to add and subtract two numbers.
- `Seconds(60 * 5)` specifies the window size
- `Seconds(1)` specifies the movement of the window.

The generated counts `DStream` will have records that are `(hashtag, count)` tuples.

Find the top 10 hashtags based on their counts

These counts are used to find popular hashtags. A simple (but not the most efficient) way to do this is to sort the hashtags based on their counts and take the top 10 records. Since this requires sorting by the counts, the count (i.e., the second item in the `(hashtag, count)` tuple) needs to be made the key. Hence, we need to first use a map to flip the tuple and then sort the hashtags. Finally, we need to get the top 10 hashtags and print them. All this can be done as follows:

```
val sortedCounts = counts.map { case(tag, count) => (count, tag) }
                         .transform(rdd => rdd.sortByKey(false))
sortedCounts.foreach(rdd =>
  println("\nTop 10 hashtags:\n" + rdd.take(10).mkString("\n")))
```

The `transform` operation allows any arbitrary `RDD-to-RDD` operation to be applied to each `RDD` of a `DStream` to generate a new `DStream`. The resulting '`sortedCounts`' DStream is a stream of RDDs having sorted hashtags. The `foreach` operation applies a given function on each RDD in a DStream, that is, on each batch of data. In this case, `foreach` is used to get the first 10 hashtags from each RDD in `sortedCounts` and print them, every second.

2.5 Further reading

- More about Spark streaming http://spark.apache.org/docs/latest/streaming-programming-guide.html#overview

3. Homework

- Exploit Spark streaming using transformation operations for windows and joins. (see http://spark.apache.org/docs/latest/streaming-programming-guide.html#overview)
- Use Spark streaming extensions for processing graphs or using machine learning algorithms on data streams.
- Implement Bloom filter using Hadoop and HBase (see challenge)
- Propose a Bloom filter application for streams. For example, deal with Twitter restriction of the number of requests by bloom filtering queries.