

# Big Data Management (BDMA & MIRI Masters)

## Session: Spark Streaming (Training)

Alberto Abelló & Sergi Nadal

The purpose of this document is that you get familiar with the environment and tools required for the lab session. This is an optional task, thus no delivery is expected and no grade will be assigned. You are free to check any external reference, for instance the Spark programming guide (<https://spark.apache.org/docs/latest/rdd-programming-guide.html>) or the Spark Streaming programming guide (<https://spark.apache.org/docs/latest/streaming-programming-guide.html>).

### Instructions

In this session, as Spark jobs will run locally, you do not need to start the Spark service and everything can be run from eclipse. Thus, you can dismiss the virtual machine and directly use the eclipse version provided in Linux or Windows on the lab session computers. Once you have fetched the code, it can be imported as a *Maven Project* into eclipse. All executions can be run locally, and thus you do not need to be concerned with the host connection. In the *Main* class, you can see that different arguments are expected, and will drive the behaviour of the program. The simplest way to define such arguments is by creating a *Run Configuration* of type *Java Application* from the *Run* menu. In the tab menu called *Arguments* you will be able to edit such arguments, in each exercise you will be guided with the required arguments.

### Exercise 1: Setup

In this session we will ingest data from the Twitter API, to this end first create your Twitter App following the guidelines in Appendix A. Do not forget to modify the variable `TWITTER_CONFIG_PATH` in `Main.java` with the path of the generated config file as described in appendix A.

Now, run the main method which should display in console the stream of tweets. Once this works, analyze the code provided in `Exercise_1.displayAllTweets`. The method `displayAllTweets` receives an object `JavaDStream<Status>` which is a batch of tweets, i.e. `Status`. A `map` function is applied, this converts each `Status` to a `String` extracting only the user's and text information. Afterwards, it is printed in console.

### Exercise 2: Computing Popular Hashtags

Next, let's try something more interesting, say, 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. After every step, you can see the contents of new `DStream` you created by using the `print()` operation. The implementation of this exercise can be found in `Exercise_2.get10MostPopularHashtagsInLast5min`, you will see all transformations, but commented and without the required logic (you need to parametrize the methods). In the next steps, you will be guided on how to complete each of them. The code is divided for each of the following subsections.

**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 "#". 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 should be 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. You should see something like this (assuming no other `DStream` has print on it):

```
-----  
Time: 1359886521000 ms  
-----
```

```
#njbng  
#njpw  
#?????  
#algeria  
#Annaba
```

**Count the hashtags over a 5 minute window** Next, 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. Note that `reduceByKeyAndWindow` is a *stateful transformation* as it combines data across multiple batches (generated *RDDs* depends on *RDDs* of previous batches), thus it requires enabling checkpointing (do not worry about this, as it is handled in `Main.java`).

There are two functions that should be defined for adding and subtracting the counts. `new Duration(5 * 60 * 1000)` specifies a window size of 5 minutes and `new Duration(1 * 1000)` specifies a movement of the window of 1 second. Note that only 'invertible' `reduce` operations that have 'inverse' functions (like subtraction is the inverse of addition) can be optimized in this manner. The generated counts *DStream* will have records that are (hashtag, count) tuples. If you print *counts* and run this program, you should see something like this:

```
-----  
Time: 1359886694000 ms  
-----
```

```
(#epic,1)  
(#WOWSetanYangTerbaik,1)  
(#recharged,1)  
(#???????????,1)  
(#jaco,1)  
(#Blondie,1)  
(#TOKIO,1)  
(#fili,1)  
(#jackiechanisamazing,1)  
(#DASH,1)  
...
```

**Find the top 10 hashtags based on their counts** Finally, these counts have to be used to find the 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.

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` should be a stream of RDDs having sorted hashtags. The `foreachRDD` operation applies a given function on each RDD in a `DStream`, that is, on each batch of data. In this case, `foreachRDD` should be used to get the first 10 hashtags from each RDD in *sortedCounts* and print them, every second. If you print the contents of *sortedCounts*, you should see something like this:

Top 10 hashtags:

```
(2,#buzzer)
(1,#LawsonComp)
(1,#wizkidleftEMECos)
(1,#???????)
(1,#NEVERSHUTMEUP)
(1,#reseteo.)
(1,#casisomoslamismapersona)
(1,#job)
(1,#????_??_????_?????)
(1,#????RT(*^~*))
```

Note that this implementation is not scalable, as we are saving the window in memory (which is limited), while we cannot make any assumption on the window's size.

### Exercise 3: Analyzing hashtags historically

In the previous exercise, we implemented the counting of hashtags over a sliding window to get the top 10 most populars. This approach had a major limitation as we were only able to track the count of hashtags over the fixed time window. Now, the goal of this exercise is to loosen such limitation and enable to periodically obtain insights about the whole stream. Precisely, we are interested in obtaining, based on the frequency of appearance, the *Top 10* hashtags and the *Median* hashtag. They should be printed at each microbatch.

To this end, in class *Exercise\_3.java* implement the necessary processing using the `updateStateByKey` operator. This operator uses a state function that generates a new state based on the previous one. Note that it uses the class `org.apache.spark.api.java.Optional` from Scala. Such class represents a value of a given type that may or may not exist. It is used in methods that might need to return empty values, instead of returning null. See the details of the API in <https://spark.apache.org/docs/latest/api/java/org/apache/spark/api/java/Optional.html>.

**Important note.** We are aware that the operator `updateStateByKey` is not the most suitable for this task, as it updates the whole state at each microbatch. Alternatively, the operator `mapWithState` is better as it only updates the state for newly incoming keys.

## A Twitter Credentials Setup

Since all of the exercises are based on Twitter's sample tweet stream, it is necessary to configure OAuth authentication with a Twitter account. To do this, you will need to setup a consumer key+secret pair and an access token+secret pair using a Twitter account. Please 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.

### A.1 Create Your Application

Go into <https://apps.twitter.com/>. 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 "Create a new application" button. The new application page should look the page shown in figure 1. 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 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 Agreement. Finally, click on the "Create your Twitter application" button.

## Create an application

### Application Details

**Name \***


Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

**Description \***


Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

**Website \***


Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.  
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

**Callback URL**


Where should we return after successfully authenticating? [OAuth 1.0a](#) applications should explicitly specify their `oauth_callback` URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

Figure 1: Application creation screen

## A.2 Token Generation

Once you have created the application, you will be presented with a confirmation page similar to the one shown in figure 2. You should be able to see the consumer key and the consumer secret that have been generated. To generate the access token and the access token secret go to the tab “Keys and Access Tokens”, and click on the “Create my access token” button at the bottom of the page (lower green arrow in the figure). Note that there will be a small green confirmation at the top of the page saying that the token has been generated.

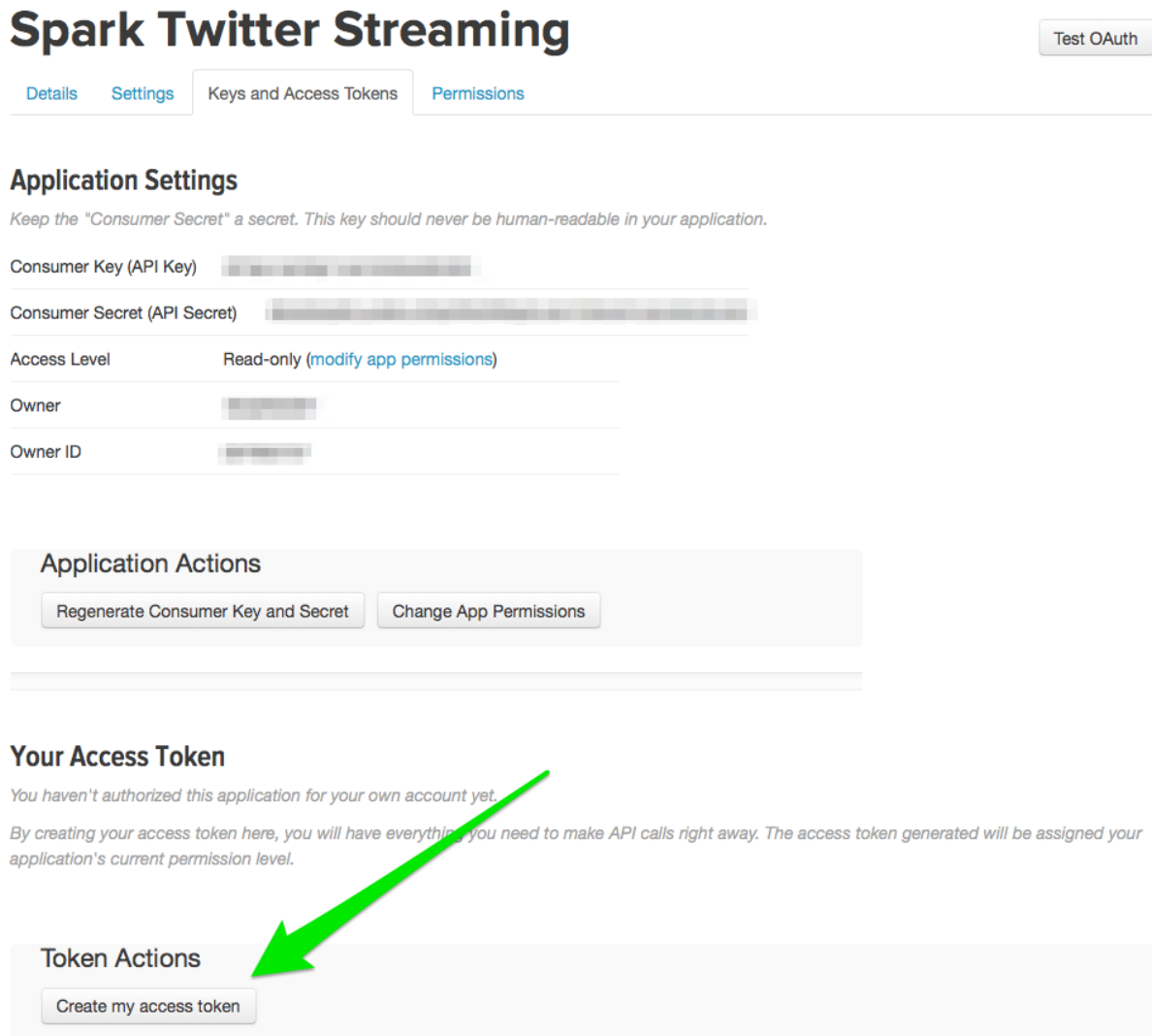


Figure 2: Token generation screen

## A.3 Get OAuth Credentials (Access Token)

Wait for the token to be generated (refresh the screen if required), and in the bottom section you will see that both access token and secret have appeared like in figure 3.

## A.4 Wrapping Everything

Finally, generate a `.txt` document named `twitter_configuration.txt` using your favorite text editor. The document should have the following structure:

## Your Access Token

*This access token can be used to make API requests on your own account's behalf. Do not share your access token secret with anyone.*





Access Token	
Access Token Secret	
Access Level	Read-only
Owner	
Owner ID	

Figure 3: Access Token Screen

```
consumerKey =
consumerSecret =
accessToken =
accessTokenSecret =
```

Copy the values from the "Keys and Access Tokens" into this appropriate keys in this file. After copying, it should look something like the following:

```
consumerKey = z25xt02zcaadf12 ...
consumerSecret = gqc9uAkjla13 ...
accessToken = 8mitfTqDrgAzasd ...
accessTokenSecret = 479920148 ...
```

Double-check that the right values have been assigned to the right keys. You might save the file in the resources directory of your Java project. Save the file in a USB flash drive or any cloud-based repository in order to use it during the class.