MIDS W205

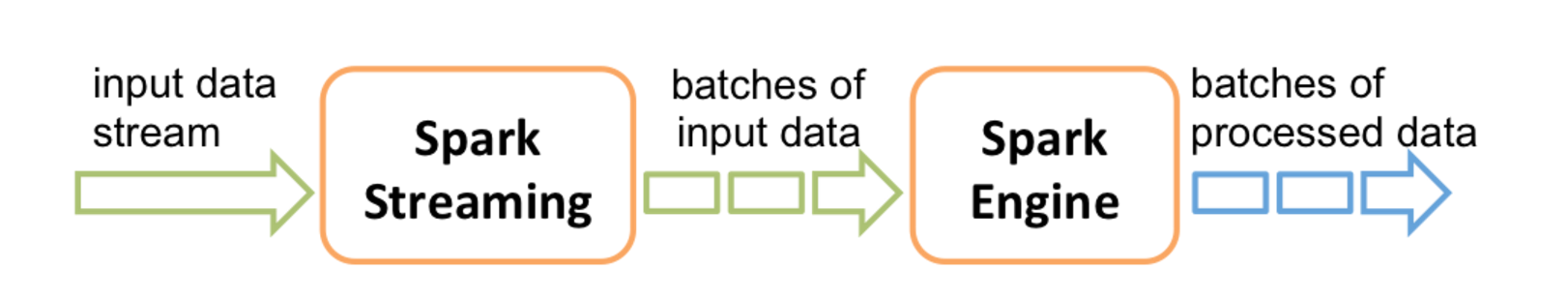
|  |  |  |  |
| --- | --- | --- | --- |
| **Lab #** | 12 | **Lab Title** | Spark Streaming Introduction |
| **Related Module(s)** | 9 | **Goal** | Introduction to Spark Streaming. |
| **Last Updated** | 10/20/15 | **Expected duration** | 60 minutes |

# Introduction

Spark Streaming is an extension to Spark that enables processing of streaming data. By streaming data processing we mean that data arrives continuously to an analytics process, and that we have a need to process the data as it comes in. Spark Streaming can receive data from any sources, and for testing purposes we can even feed data from a few “poor” mans streams.

Before we start with the Lab we will briefly introduce Spark Streaming and its main concepts.

Spark Streaming differs from Storm in several respects. Firstly Spark Streaming divides an incoming stream into batches. Each batch is processed as one unit using the core spark infrastructure. This is in contrast to Storm were data is processed as it comes in. Due to this difference Spark Streaming is sometimes called a micro batching solution. Micro batching has both advantages and disadvantages compared to a real-time streaming solution.

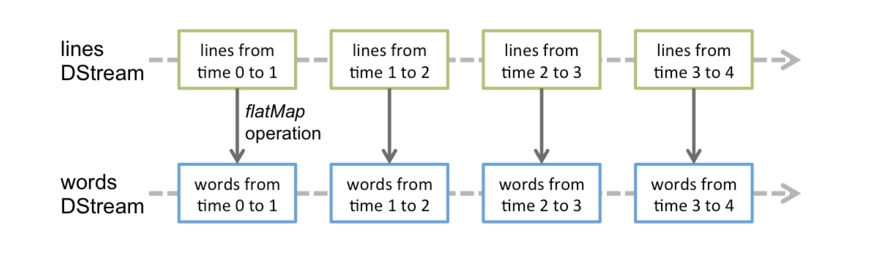


There are a few basic concepts that you need to understand in Spark Streaming: DStream, Transformations and Output Operations.

Discretized Streams, or DStreams for short, is an architectural concept that essentially captures the fact that an incoming continuous stream is chunked in to discrete RDDs for Spark processing. DStreams support many of the transformations supported by RDD’s. Here is a list of some of them. The full list is available in the programming guide.

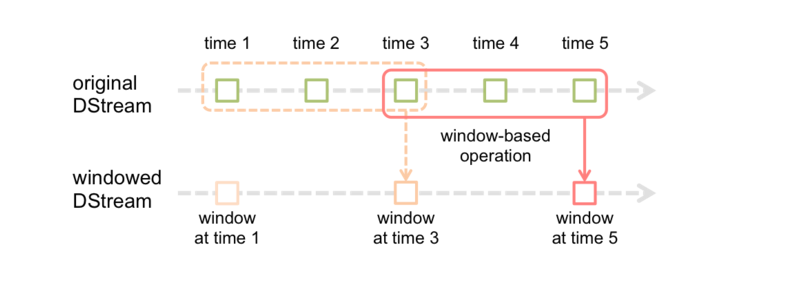
* **map(func):** Return a new DStream by passing each element of the source DStream through a function func.
* **flatMap(func)**: Similar to map, but each input item can be mapped to 0 or more output items.
* **filter(func)**: Return a new DStream by selecting only the records of the source DStream on which func returns true.
* **repartition(numPartitions)**: Changes the level of parallelism in this DStream by creating more or fewer partitions.
* **union(otherStream):** Return a new DStream that contains the union of the elements in the source DStream and otherDStream.
* **count()** : Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
* **reduce(func):** Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function func (which takes two arguments and returns one). The function should be associative so that it can be computed in parallel.
* **countByValue()**: When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
* **join(otherStream, [numTasks]):** When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.

The following figure from the programming guide illustrates DStreams and transformations well. Each DStream is partitioned into a set of RDD’s. When you apply a stateless transformation you will create a new DStream of RDD’s with the data that resulted from each transformation.



The above transformations are all state-less. This means that they create new DStream, but there is no state information maintained that can be accesses or updated by subsequent transformations or other operations. Spark Streaming allows you to define a function that updates a to a new defined state based on the current state and values from an stream. Stateful transformations requires checkpoints, so we consider that somewhat outside of this introductory lab.

Spark Streaming provides a concept of windowing. This essentially means that you can define window that spans more than one RDD in a DStream. Spark Streaming would combine all the RDD’s in the window and allow you to apply a transformation on the data in the window. It can be useful in many types of real-time computations. Lets say you receive new Stock quotes every second captured from as incoming stream socket as a Distinct RDD. But you want to calculate some value based on the last 30 second window. Windowing would allow you to combine the one second DStream RDD’s into 30 second windows and slide the window for you. It allows you to more easily compute the value you are seeking based on the last 30 seconds of real-time data. The figure below from the programming guide depicts the concept of sliding windows.



Below we provide a sample of operations available on windows.

* **window(windowLength, slideInterval)** : Return a new DStream which is computed based on windowed batches of the source DStream.
* **countByWindow(windowLength, slideInterva**l) : Return a sliding window count of elements in the stream.
* **reduceByWindow(func, windowLength, slideInterval)** : Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using func. The function should be associative so that it can be computed correctly in parallel.
* **reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])**: When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function func over batches in a sliding window. Note: By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property spark.default.parallelism) to do the grouping. You can pass an optional numTasks argument to set a different number of tasks.

Finally, we like to discuss check pointing.

Check pointing, linage

# Spark Streaming

Simple stream example from port

Example from file

Filtering of streams.

Connect to tcp stream

Check pointing

Change the window

# Instructions, Resources, and Prerequisites

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. This lab will cover the followings:

1. Pre-Requirements to start Spark Streaming
2. How to generate streaming data using Python
3. How to Access & Initialize dependencies for Spark Streaming on Spark shell
4. Files live stream example using Spark Streaming from Spark Shell

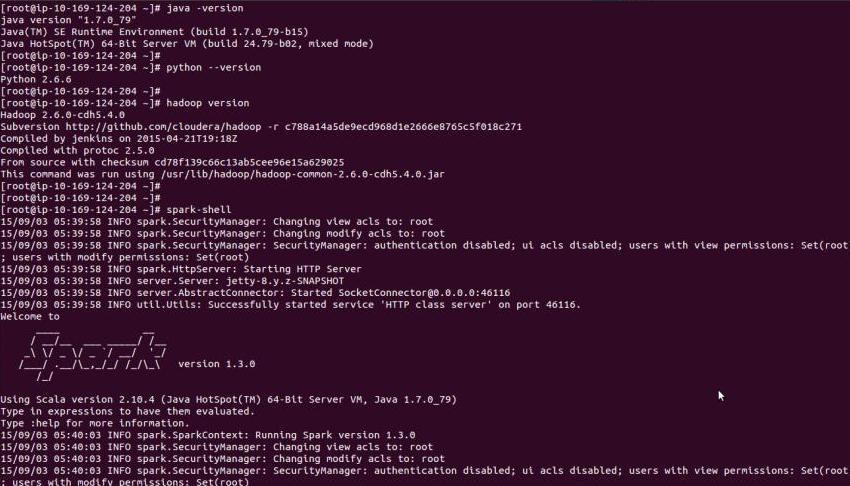
|  |  |
| --- | --- |
| **Resource** | **What** |
| http://spark.apache.org/docs/latest/streaming-programming-guide.html#transformations-on-dstreams | Spark Streaming Programming Guide. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| http://www.meetup.com/meetup\_api/docs/stream/2/rsvps/#websockets | Meetup websocket API. |
| http://www.slideshare.net/spark-project/deep-divewithsparkstreaming-tathagatadassparkmeetup20130617 | Technical presentation on Spark Streaming. |

# Step-1. Pre-Requirements to start Spark Streaming:

You must have the followings on your server to start Spark Streaming.

1. Java 1.6+
2. Hadoop 1.X or 2.X - optional (for file storage or we can use local file system also)
3. Spark 1.X
4. Data source (we will be using file stream)
5. Python

Please check on your server if these exist. You can refer to the following Unix log for the exact code.



# Step-2. Getting Started

$pyspark

>>>from pyspark import SparkContext

>>>from pyspark.streaming import StreamingContext

>>>sc

Pyspark already has a spark context so we will not create one. If you were to implement this as a program you would need to add a statement such as:

sc = SparkContext("local[2]", "MyApp")

For the time being we are using the interactive shell to make it easier for you to experiment and retry. We will start by doing a very simple streaming operation by converting all incoming words to upper case. We will first show how you do it from a file to simulate a stream. Secondly we will who how to have the Streaming application listen to a stream socket.

First create a directory of you choice. You will be submitting files to this directory for processing by the application. For the purpose of this lab I will name it /tmp/datastreams. You can call anything of your choosing, but make sure to use your path in place of this path.

Create a streaming context.

>>>ssc = StreamingContext(sc, 1)

Tell the context to read data from files in a directory. This means it will be monitoring the directory for new files and read them as they arrive. All files in the directory must have the same format, and must be created atomically. You can create them by moving or copying them there. You can not open a file and incrementally add to it and expect that the updates will be read.

>>>lines= ssc.textFileStream("/tmp/datastreams")

Now lets do a simple transformation that converts all the words in an RDD to upper case.

>>>uclines = lines.map(lambda word: word.upper())

We will output the words by printing the top results.

>>>uclines.pprint()

Finally we start the process. After this command you should see an output confirming the processing of a nee RDD. We have not moved any data to the directory yet, so each result will be empty.

>>>ssc.start()

Open a separate Unix terminal window. Lets assume you have a simple files call words with the following content.

hej

kalle

kula

nisse

Hello

coloraDO

DOrado

eldorade

gatrorade

Copy the file to the datastream directory.

$ cp words /tmp/datastreams/

If you look in the streaming window you will the result of the processing and that the word were converted to upper case words after the processing. Try copying the file to the directory again. What happens? If you instead copy to a new file you will see a different result.

$ cp words /tmp/datastreams/w1

You can continue copying to a different file and the will see that the spark process will pick the data up as it arrives in the directory.

$ cp words /tmp/datastreams/w2

$ cp words /tmp/datastreams/w3

You can stop the Spark Streaming application by typing the following stop command in te pyspark shell.

>>>ssc.stop()

It is very useful to understand how to process files from during development of your processing logic. This way you can test different logic based on some specific data and so forth.

Now lets do the same operation but have the process listen to a streaming socket. You will need to restart pyspark we it will not allow you to have a new StreamingContext associated with the same Spark Context. After restart execute the import statements and the creation of the StreamingContext.

>>>from pyspark import SparkContext

>>>from pyspark.streaming import StreamingContext

>>>ssc = StreamingContext(sc, 1)

Instead of reading from files in a directory we will listen to a socket. The process terminating the socket will be running on you local host and we will use the port number 9999.

>>>lines = ssc.socketTextStream("localhost", 9999)

Define the transformation and output statement as previously and start the process.

>>>uclines= lines.map(lambda word: word.upper())

>>>uclines.pprint()

>>>ssc.start()

You will see some errors from Spark since there is no active port to connect to. But the process will continue to try to connect to the port. To create a port to which the streaming process can connect we will use the Unix nc command. The name nc stands for netcat. You can think about it as the Unix cat but for pushing data to a socket rather than a file. If you try this Lab on Windows, you can use the Windows netcat command in place of nc.

In a terminal window type the command below. The l option tells nc to listen for incoming connections. This is the right behavior as we expect Spark Streaming to connect to the socket. The k command tell nc to continue listening even if a connection is completed. This way you can have nc running and restart you spark streaming application without needing to restart nc.

$nc -lk 9999

In the terminal window were you started nc, type some words. What happens on the streaming application side?

The batch duration for our simple streaming application is one second. This is a short time when a human is providing the input. Lets change that to 30 seconds and see what happens. Restart pyspark and run the following commands.

>>>from pyspark import SparkContext

>>>from pyspark.streaming import StreamingContext

>>>ssc = StreamingContext(sc, 30)

>>>lines = ssc.socketTextStream("localhost", 9999)

>>>uclines= lines.map(lambda word: word.upper())

>>>uclines.pprint()

>>>ssc.start()

Type some words in the nc terminal window. As you can see, spark is now batching things up in RDD representing 30 seconds of incoming data.

# Step-2. Parsing JSON data

Form a file filter some data.

curl -i http://stream.meetup.com/2/rsvps | nc -lk 9999

# Step-2. How to generate streaming data using Python:

Here is the small snippet of code to generate data using python.

1. Create a python file name “generate\_data.py” to generate live streaming data. Make sure there is no error copying and pasting this code to your script.

#!/usr/bin/python

import threading

import time

import string

import random

import os

import uuid

class Generate\_events(threading.Thread):

def \_\_init\_\_(self, events\_count, file\_name):

threading.Thread.\_\_init\_\_(self)

if os.path.exists(file\_name):

self.file\_name = open(file\_name, 'a')

else:

self.file\_name = open(file\_name, 'w')

self.events\_count = events\_count

def \_\_gen\_sal(self):

while True:

sal = range(1000000)

yield random.choice(sal)

def \_\_gen\_emp\_number(self):

while True:

sal = range(1000000)

yield random.choice(sal)

def \_\_gen\_bonus(self):

while True:

bonus = range(500000)

yield random.choice(bonus)

def \_\_gen\_name(self):

alphabets = list(string.ascii\_lowercase)

while True:

yield ''.join(random.choice(alphabets) for \_ in range(6))

def \_get\_name(self):

return self.\_\_gen\_name().next()

def \_get\_sal(self):

return self.\_\_gen\_sal().next()

def \_get\_bonus(self):

return self.\_\_gen\_bonus().next()

def \_get\_emp\_number(self):

return self.\_\_gen\_emp\_number().next()

def run(self):

try:

for event in range(self.events\_count):

self.file\_name.write("\n" + str(self.\_get\_emp\_number()) + ',' + self.\_get\_name() + ',' + str(self.\_get\_sal()) + ',' + str(self.\_get\_bonus()))

self.file\_name.close()

except Exception, e:

print e

while(True):

#id = os.urandom(32)

id = uuid.uuid4().int & (1<<64)-1

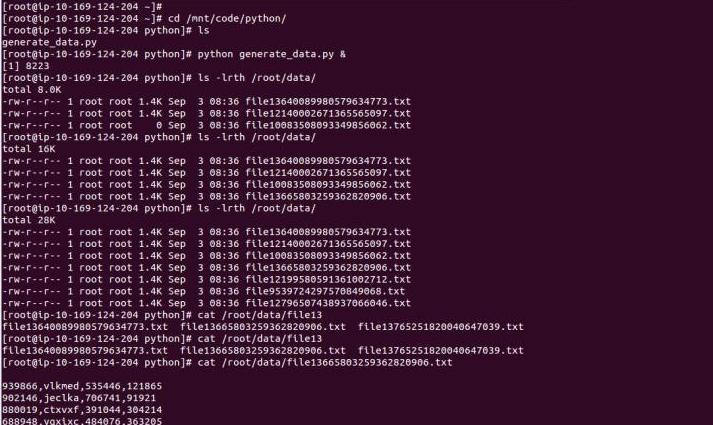
file = 'file'+str(id)+'.txt'

Generate\_events(50, '/root/data/'+file).start()

time.sleep(5)

1. Create a folder call “data” in /root/ directory
2. Input Data set contains these columns “emp\_number,name,salary,bonus,filename”
3. You can run this python file using this command “python generate\_data.py”
4. You can find result data in /root/data/. You can use this command to see your data “ls -lrth /root/data/”. Don’t keep running this script as it can fill up your space quickly.

Please refer to the Unix log of these steps.



# Step-3. How to Access and Initialize Dependencies for Spark Streaming on Spark shell:

**How to Access:**

You can access spark shell using below command on terminal

spark-shell

You can refer to this Unix log for the same.



#### How to build Dependencies for Spark Streaming :

Below listed imports statements are necessary for Spark Streaming. You can directory use these commands and run in Spark-Shell.

import org.apache.hadoop.io.\_

import org.apache.hadoop.mapred.OutputFormat

import org.apache.spark.\_

import org.apache.spark.streaming.\_

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

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat

You can refer to this Unix log for the same.



#### How to Initialize Spark Streaming:

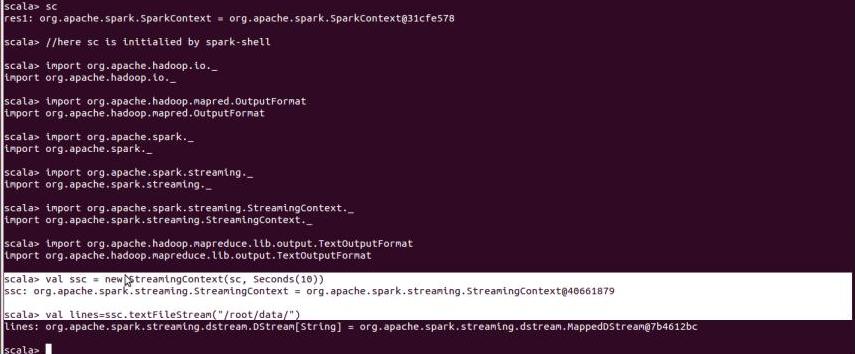
You can use the following commands to do the same.

val ssc = new StreamingContext(sc, Seconds(10))

val lines=ssc.textFileStream(“/root/data/”)

Here sc is sparkContext, which has been initialized by spark-shell.

You can refer below the Unix log for the same.



#### Step-4. Live streaming data processing example using Spark Streaming on Spark-Shell :

**on Spark shell:**

**Please follow the following steps:**

* Start the data generate process using this command  
   python generate\_data.py & (refer to step-2)
* Make a folder using “mkdir /root/outputdata” to store output results

The below code simply adds run time the salaries and bonuses and create a employee monthly income result set. You can use the following code on Spark-Shell and execute.

import org.apache.hadoop.io.\_

import org.apache.hadoop.mapred.OutputFormat

import org.apache.spark.\_

import org.apache.spark.streaming.\_

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

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat

val ssc = new StreamingContext(sc, Seconds(10))

val lines=ssc.textFileStream("file:///root/data/")

val total= lines.map(line=>if(line.contains(",")){(line+","+(line.split(",")(2)).toLong+(line.split(",")(3)).toLong,null)})

total.saveAsTextFiles("file:///root/outputdata/","output")

ssc.start()

ssc.awaitTermination();

Once “sss.start ” processes started on spark shell, you can see result on “/root/outputdata” folder. You might want to open two termnials to see data generation process and spark streaming real time calculations.

You can refer to the following Unix log for the same. Also, please check in your output folder for the output results.

