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.

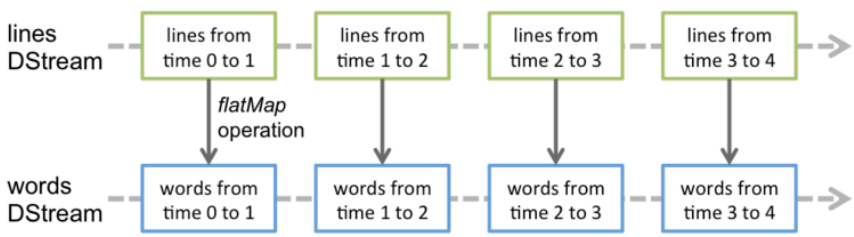
There are a few basic concepts that you need to understand in Spark Streaming: DStream, Transformations and Output Operations.

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.

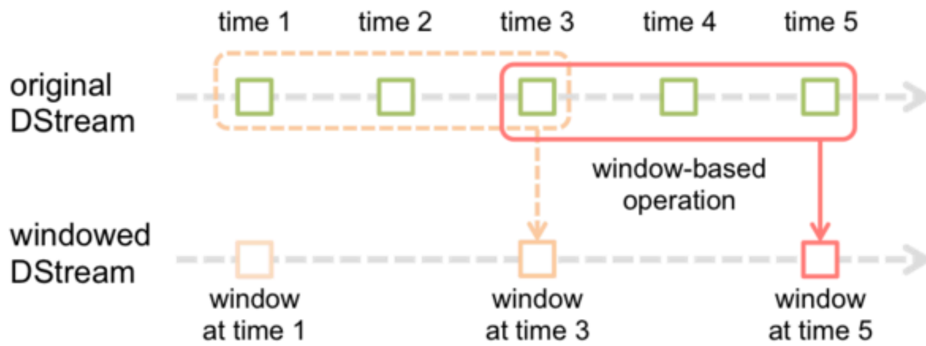
- **repartition(numPartitions)**: Changes the level of parallelism in this DStream by creating more or fewer partitions.
- **union(otherStream):** Return a new DStream that contains the union of the elements in the source DStream and otherDStream.
- **count()** : Return a new DStream of single-element 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.
JSON Introduction	http://www.w3schools.com/json/json_syntax.asp
Meetup API	http://www.meetup.com/meetup_api/
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-0. Pre-Requirements to start Spark Streaming

Update the repository from git hub. It should contain a directory called "`somedata`". We will be using this to illustrate a few things.

First, if an example starts with the "`$`" prompt, it is run in the Linux shell. If it starts with the "`>>>`" prompt, it is run in `pyspark` shell. If code is within a "box" you should consider it a code fragment for illustration purposes and not a command you should execute.

Step-1. 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 called `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
```

SUBMISSION 1: Provide a screenshot of the output from the Spark Streaming process.

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

We will now parse a more complicated data structure. We will parse data feed from the Meetup API. Specifically we will be looking at RSVP event. Meetup provides several streaming API's. For the purpose of this Lab we will be creating a simple test stream. In real application you may need to implement a Spark Streaming end point using the Custom Receiver framework.

What is JSON is format for a human readable representation of data that is transferred between distributed components. The following is a very simple example. An object consists of name-value pairs or sub objects. The components in a name value pair are separated by a colon. An object is enclosed by curly brackets. A value that also be an array that contains multiple objects.

```
{
"firstName": "John",
"lastName": "Smith",   "isAlive": true,   "age": 25,
"address": {
"streetAddress": "21 2nd Street",     "city": "New York",
"state": "NY",     "postalCode": "10021-3100"   },
"children": [],
"spouse": null
}
```

The following is an example of a rsvp JSON object. As you can see it has sub-objects such as `venue`. And value such as `visibility`.


```
{"venue":{"venue_name":"Couchbase San Francisco","lon":-
122.397354,"lat":37.790005,"venue_id":21741962},"visibility":"pub
lic","response":"yes","guests":0,"member":{"member_id":8301128,"p
hoto":"http:\/\/photos3.meetupstatic.com\/photos\/member\/9\/0\/a
\/8\/thumb_29557032.jpeg","member_name":"Che
Hsu"},"rsvp_id":1577033972,"mtime":1446516373612,"event":{"event_
name":"Full Stack Development with NoSQL & Node.js or
GoLang","event_id":"226259580","time":1446602400000,"event_url":"
http:\/\/www.meetup.com\/The-San-Francisco-Couchbase-Meetup-
Group\/events\/226259580\/"},"group":{"group_topics":[{"urlkey":"
java","topic_name":"Java"},{"urlkey":"php","topic_name":"PHP"},{"
urlkey":"newtech","topic_name":"New
Technology"},{"urlkey":"ria","topic_name":"Rich Internet
Applications"},{"urlkey":"mobile-
development","topic_name":"Mobile
Development"},{"urlkey":"nosql","topic_name":"NoSQL"},{"urlkey":"
databasepro","topic_name":"Database
Professionals"},{"urlkey":"database-
development","topic_name":"Database
Development"},{"urlkey":"softwaredev","topic_name":"Software
Development"},{"urlkey":"web-development","topic_name":"Web
Development"},{"urlkey":"ia","topic_name":"Information
Architecture"}],"group_city":"San
Francisco","group_country":"us","group_id":1693125,"group_name":"
The San Francisco Couchbase Group","group_lon":-
122.42,"group_urlname":"The-San-Francisco-Couchbase-Meetup-
Group","group_state":"CA","group_lat":37.75}}
```

Before we start receiving using a stream we want to make sure we can parse the data properly. Initially we just like to extract the venue from each incoming record. You can understand the rsvp stream call and response formats by looking at the API definition of rsvps.

Connect to a socket directly using the curl command below.

```
$curl -i http://stream.meetup.com/2/rsvps
```

You should see rsvp events being printed in your terminal window as they arrive from the stream.

Before we parse the stream, lets read data from the file system and try out a few transformations. In the `somedata` directory you will have a few files called : `meetup.data.1`, `meetup.data.2` and so forth. Each containing 200 rsvp events, except the last one that contains 162 rsvp events. The data in these files contains more that 1000 rsvp's that we retrieved from the meetup api. We will use the by copying them to `/tmp/datastreams` so that a streaming application will pick them up.

The statement below parses the incoming JSON objects. It takes each row (which is a JSON rsvp object) and tests if it has a `venue` object. If it does it sends the object to the `json.loads` function for parsing. Once the parsing is done it returns just the `venue` object.

```
slines = lines.flatMap(lambda x: [ j['venue'] for j in
json.loads('['+x+']') if 'venue' in j] )
```

Load the following spark streaming program:

```
>>>from pyspark import SparkContext
>>>from pyspark.streaming import StreamingContext
>>>import json
>>>ssc = StreamingContext(sc, 10)
>>>lines = ssc.textFileStream("/tmp/datastreams")
>>>slines = lines.flatMap(lambda x: [ j['venue'] for j in
json.loads('['+x+']') if 'venue' in j] )
>>>cnt=jslines.count()
>>>ccnt.pprint()
>>>jslines.pprint()
```

Finally, start the process.

```
>>>ssc.start()
```

We are importing a `json` library. We are using this library to parse the json structure for an rsvp using the `json.loads` function. We then extract the `venue` element out of the structure and push that into a new RDD. We will print the total amount of rsvp events received in the batch, the number of events with a `venue`, and the top venues with `pprint`.

We can test this by just copying the `meetup.data.*` files into the `/tmp/datastreams` directory.

```
$cp meetup.data.1 /tmp/datastreams/
$cp meetup.data.2 /tmp/datastreams/
$cp meetup.data.3 /tmp/datastreams/

and so forth…
```

Everything you copy a file you it will be detected by Spark Streaming, picked up and included in the current batch.

> SUBMISSION 2: What are the number of venue objects in each processed batch correponsinding to meetup.data.{1,2,3,4,5,6}?

Step-3. Hooking up to a simple stream.

In this step we will analyze incoming streaming data. The way we integrate is not production grade, but it provides is with an easy way of actually analyzing a real stream.

First we will need to change our script so that it reads from a socket.

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
import json
ssc = StreamingContext(sc, 10)
lines = ssc.socketTextStream("localhost", 9999)
jslines = lines.flatMap(lambda x: [ j['venue'] for j in
json.loads('['+x+']') if 'venue' in j] )
lcnt=lines.count()
lcnt.pprint()
c=jslines.count()
c.pprint()
jslines.pprint()
ssc.start()
```

In a separate terminal window run the command below.

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

Note: you may encounter a parsing error when you first start parsing the stream. This is because a websocket stream send an HTTP header before it send the data. This is obviously not JSON and the Spark Streaming JSON parsing will not like that.

This command used `curl` to grab the content from the Meetup websocket api and feeds it into `nc`. The `nc` process will make the data available on the socket on port 9999 on `localhost`. The behavior of this stream can become artificially bursty. This is due to that the unix pipe command `(|)` buffer data before forwarding it. To more closely mimic the incoming rate of rsvp's you can type the following on OSX.

```
script -q /dev/null curl -
i http://stream.meetup.com/2/rsvps  | nc -lk 9999
```

On Linux it would be the following command.

```
script –q -c/dev/null curl
     -i http://stream.meetup.com/2/rsvps  | nc -lk 9999
```

You would probably not see that much difference on the spark streaming end since it is batching the incoming events up anyways. But you can see in the logging that the spark streaming process receives many more messages of data. This is because the sending process forwards them directly as they arrive, rather than letting the pipe buffer them.

Step-4. Running with spark-submit

Until now we have been running the command in a pyspark shell. Clearly one also need to be able to submit and execute streaming jobs and standalone programs. Put the following commands into a file called `venuecounter.py`. For the description I will assume you put the script in `$HOME`

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
import json
sc = SparkContext("local[2]", "MyApp")
ssc = StreamingContext(sc, 10)
lines = ssc.socketTextStream("localhost", 9999)
jslines = lines.flatMap(lambda x: [ j['venue'] for j in
json.loads('['+x+']') if 'venue' in j] )
lcnt=lines.count()
lcnt.pprint()
c=jslines.count()
c.pprint()
jslines.pprint()
ssc.start()
```

```
ssc.awaitTermination()
```

Start the “poor mans stream” using either of the following command.

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

Or if you want to avoid the pipe buffering (OSX command) you can use the following.

```
$script -q /dev/null curl -
i http://stream.meetup.com/2/rsvps  | nc -lk 9999
```

Run the script with the submit command:

```
spark-submit $HOME/venuecounter.py localhost 9999
```

You should see output emerge looking something similar to what is show in the screenshot below.

```
-------------------------------------------

-------------------------------------------
Time: 2015-11-08 10:42:50
-------------------------------------------

15/11/08 10:42:52 WARN BlockManager: Block input-0-1447008172200 replicated to only 0 peer(s) instead of 1 peers
15/11/08 10:42:52 WARN BlockManager: Block input-0-1447008172400 replicated to only 0 peer(s) instead of 1 peers
15/11/08 10:42:53 WARN BlockManager: Block input-0-1447008172800 replicated to only 0 peer(s) instead of 1 peers
15/11/08 10:42:54 WARN BlockManager: Block input-0-1447008174200 replicated to only 0 peer(s) instead of 1 peers
15/11/08 10:42:56 WARN BlockManager: Block input-0-1447008175800 replicated to only 0 peer(s) instead of 1 peers
15/11/08 10:42:56 WARN BlockManager: Block input-0-1447008176200 replicated to only 0 peer(s) instead of 1 peers
15/11/08 10:42:56 WARN BlockManager: Block input-0-1447008176600 replicated to only 0 peer(s) instead of 1 peers
15/11/08 10:42:57 WARN BlockManager: Block input-0-1447008177000 replicated to only 0 peer(s) instead of 1 peers
15/11/08 10:42:57 WARN BlockManager: Block input-0-1447008177200 replicated to only 0 peer(s) instead of 1 peers
15/11/08 10:42:58 WARN BlockManager: Block input-0-1447008177800 replicated to only 0 peer(s) instead of 1 peers
15/11/08 10:42:58 WARN BlockManager: Block input-0-1447008178000 replicated to only 0 peer(s) instead of 1 peers
15/11/08 10:42:58 WARN BlockManager: Block input-0-1447008178200 replicated to only 0 peer(s) instead of 1 peers
15/11/08 10:42:58 WARN BlockManager: Block input-0-1447008178600 replicated to only 0 peer(s) instead of 1 peers
15/11/08 10:42:59 WARN BlockManager: Block input-0-1447008179000 replicated to only 0 peer(s) instead of 1 peers
15/11/08 10:42:59 WARN BlockManager: Block input-0-1447008179400 replicated to only 0 peer(s) instead of 1 peers
15/11/08 10:43:00 WARN BlockManager: Block input-0-1447008180400 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2015-11-08 10:43:00
-------------------------------------------
603

15/11/08 10:43:01 WARN BlockManager: Block input-0-1447008181000 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2015-11-08 10:43:00
-------------------------------------------
465

-------------------------------------------
Time: 2015-11-08 10:43:00
-------------------------------------------
{u'lat': 33.391479, u'venue_id': 17582512, u'lon': -111.788322, u'venue_name': u'India Oven'}
{u'lat': 43.251106, u'venue_id': 1556769, u'lon': -79.870781, u'venue_name': u'The Undermount'}
{u'lat': 39.570477, u'venue_id': 23912315, u'lon': 2.637716, u'venue_name': u'Bar Carrera23'}
{u'lat': 29.689941, u'venue_id': 23713095, u'lon': -95.827621, u'venue_name': u'Westheimer Lakes '}
{u'lat': -27.399618, u'venue_id': 24017706, u'lon': 153.440933, u'venue_name': u'Amity point Stradbroke'}
{u'lat': 51.420166, u'venue_id': 22443312, u'lon': -0.987627, u'venue_name': u'Grow @Green Park a ConnectTVT Innovation Hub'}
{u'lat': 37.975342, u'venue_id': 24137559, u'lon': 23.710804, u'venue_name': u'Found.ation'}
{u'lat': 21.30485, u'venue_id': 22696332, u'lon': -157.857758, u'venue_name': u'Makua Beach (Pray for Sex Beach)'}
{u'lat': 36.095478, u'venue_id': 7666372, u'lon': -86.80249, u'venue_name': u'JT Moore Middle School'}
{u'lat': 39.570477, u'venue_id': 23912315, u'lon': 2.637716, u'venue_name': u'Bar Carrera23'}
...
```

SUBMISSION 3: Provide a screenshot showing the running Spark Streaming application.

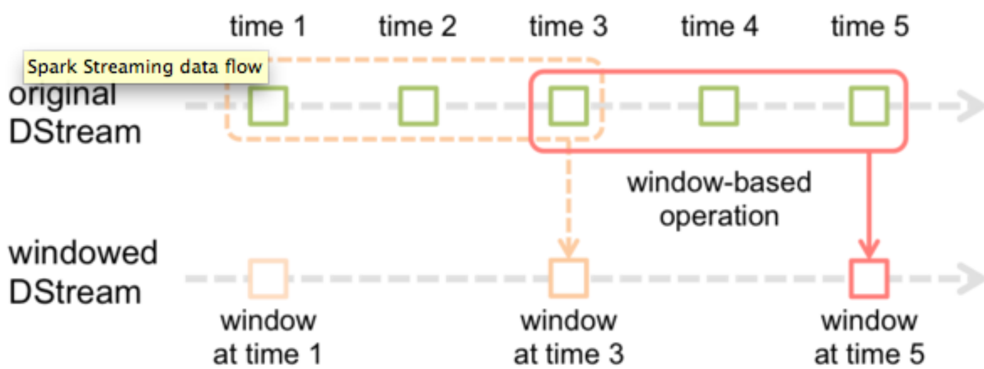
Step-5 Sliding Window

In this step we will do a simple sliding window to illustrate the concept. For the purpose of this application we like to know how many rsvp's we received in the last batch (10 seconds), the aggregated number of rsvp's the last 30 seconds and finally the number of rsvp's with a `venue` object in the last batch. To achieve this we will use windowing.

Windowing allows us to perform certain transformations for a sliding window over a number of batches. Our batch size is 10 seconds. We will be calling the DStream transform **`countByWindow`**`(`*`windowLength,slideInterval`*`).` We want to perform the count calculation of last 30 seconds so the `windowLength` argument will be 30. We want to perform the calculation every 10 seconds, so the `slideInterval` will be set to 10.

```
wlcnt=lines.countByWindow(30,10)
```

Using the below illustration each green box represents a 10 second batch, the red box represents the sliding window of 30 seconds.



In order to enable windowing we will need to turn on checkpointing. This is done by defining a checkpoint directory. I will assume you have created the checkpoint directory in `/tmp/checkpointing`, but you can change that as you prefer. The statement looks as follows.

```
ssc.checkpoint("/tmp/checkpointing")
```

The final program looks as follows, save this in `$HOME/venuecounter2.py`:

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
import json
sc = SparkContext("local[2]", "MyApp")
ssc = StreamingContext(sc, 10)
ssc.checkpoint("/tmp/checkpointing")
lines = ssc.socketTextStream("localhost", 9999)
wlcnt=lines.countByWindow(30,10)
jslines = lines.flatMap(lambda x: [ j['venue'] for j in
json.loads('['+x+']') if 'venue' in j] )
lcnt=lines.count()
wlcnt.pprint()
lcnt.pprint()
c=jslines.count()
c.pprint()
jslines.pprint()
```

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

Now start the stream using the command below.

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

Or if you want to avoid the pipe buffering (OSX command) you can use the following.

```
$script -q /dev/null curl -
i http://stream.meetup.com/2/rsvps  | nc -lk 9999
```

Then start the stream processing application using the command:

```
$park-submit $HOME/venuecounter2.py localhost 9999
```

SUBMISSION 4: Provide a screenshot of the running Spark Streaming application that shows that the CountByWindow indeed provides an sum of the counts from the 3 latest batches. See example screenshot below:

```
-------------------------------------------
Time: 2015-11-08 12:56:50
-------------------------------------------
87

15/11/08 12:56:51 WARN BlockManager: Block input-0-1447016210800 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:56:51 WARN BlockManager: Block input-0-1447016211000 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2015-11-08 12:56:50
-------------------------------------------
26

15/11/08 12:56:51 WARN BlockManager: Block input-0-1447016211400 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2015-11-08 12:56:50
-------------------------------------------
24

15/11/08 12:56:52 WARN BlockManager: Block input-0-1447016212400 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:56:53 WARN BlockManager: Block input-0-1447016213000 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:56:54 WARN BlockManager: Block input-0-1447016213800 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:56:55 WARN BlockManager: Block input-0-1447016215000 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:56:55 WARN BlockManager: Block input-0-1447016215600 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:56:56 WARN BlockManager: Block input-0-1447016215800 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:56:56 WARN BlockManager: Block input-0-1447016216400 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:56:57 WARN BlockManager: Block input-0-1447016217000 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:56:57 WARN BlockManager: Block input-0-1447016217200 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:56:57 WARN BlockManager: Block input-0-1447016217600 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:56:58 WARN BlockManager: Block input-0-1447016217800 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:56:58 WARN BlockManager: Block input-0-1447016218400 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:56:59 WARN BlockManager: Block input-0-1447016219600 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:00 WARN BlockManager: Block input-0-1447016220200 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2015-11-08 12:57:00
-------------------------------------------
61

15/11/08 12:57:01 WARN BlockManager: Block input-0-1447016221000 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2015-11-08 12:57:00
-------------------------------------------
22

-------------------------------------------
Time: 2015-11-08 12:57:00
-------------------------------------------
22

15/11/08 12:57:01 WARN BlockManager: Block input-0-1447016221600 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:02 WARN BlockManager: Block input-0-1447016222200 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:03 WARN BlockManager: Block input-0-1447016223000 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:03 WARN BlockManager: Block input-0-1447016223600 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:04 WARN BlockManager: Block input-0-1447016223800 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:04 WARN BlockManager: Block input-0-1447016224600 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:05 WARN BlockManager: Block input-0-1447016224800 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:05 WARN BlockManager: Block input-0-1447016225200 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:05 WARN BlockManager: Block input-0-1447016225600 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:06 WARN BlockManager: Block input-0-1447016225800 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:06 WARN BlockManager: Block input-0-1447016226000 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:07 WARN BlockManager: Block input-0-1447016226800 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:08 WARN BlockManager: Block input-0-1447016227800 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:09 WARN BlockManager: Block input-0-1447016229200 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:10 WARN BlockManager: Block input-0-1447016229800 replicated to only 0 peer(s) instead of 1 peers
15/11/08 12:57:10 WARN BlockManager: Block input-0-1447016230400 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2015-11-08 12:57:10
-------------------------------------------
70

-------------------------------------------
Time: 2015-11-08 12:57:10
-------------------------------------------
22

15/11/08 12:57:11 WARN BlockManager: Block input-0-1447016231400 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2015-11-08 12:57:10
-------------------------------------------
19
```