# MIDS W205

| Lab # | 6 | Lab Title | Apache Storm Introduction |
|---|---|---|---|
| Related Module(s) | 6 | Goal | Get you started on Storm |
| Last Updated | 1/20/2017 | Expected duration | 90–120 minutes |

## Introduction

A Storm application is designed as a "topology" represented as a direct acyclic graph (DAG) with *spouts* and *bolts* acting as graph vertices. Edges on the graph are named streams, and they direct data from one node to another. Together, the topology acts as a data-transformation pipeline. At a superficial level the general topology structure is similar to a MapReduce job, with the main difference being that data are processed in real time as opposed to in individual batches. Additionally, Storm topologies run indefinitely until killed, whereas a MapReduce job must eventually end.

Storm can be used with many different languages. To avoid introducing a new language we will be using Python for our implementations of spouts and bolts. To get an example running quickly, we will use streamparse. Although the installation of streamparse is somewhat cumbersome, we believe it will simplify repeated usage of Storm and Python.

If you run `sparse quickstart`, streamparse will quick-start a local Storm and Python project using the streamparse framework. The following basic example will implement a simple word count against a stream of words. Going into that directory and executing `sparse run` will create a local Apache Storm cluster and execute your topology of Python code against the local cluster.

> **Note: You need to use a different AMI for this Lab. Here is the information for the new AMI:**
>
> **AMI Name: UCB MIDS W205 EX2-FULL**
>
> **AMI ID: ami-d4dd4ec3**

### Here are the steps we will cover in this lab:

- Review of the definition and implementation of a simple word-count application that uses Storm.
- Run the sample word-count Storm application.
- Create and explore a new application simulating a tweet-analysis application.

## Instructions, Resources, and Prerequisites

In the following table you will find references to resources related to programs and components used and mention in subsequent sections.

| Resource | What |
|---|---|
| http://storm.apache.org/documentation.html | Apache Storm documentation |
| http://streamparse.readthedocs.org/en/latest/quickstart.html | Introduction to streamparse topology definitions |
| https://streamparse.readthedocs.org/en/latest/api.html | Streamparse documentation |
| http://www.pixelmonkey.org/2014/05/04/streamparse | Short description of streamparse |
| https://drive.google.com/file/d/0B6706xGNaPPyZ1l4bjFUMHJWdkk/view?usp=sharing | Instruction video referred to in this lab |

## Step 1: Environment and Tool Setup

You have streamparse installed in this new AMI. It will greatly simplify the creation of Python Storm projects and help you get a simple example up and running quickly. The following video tutorial show you how to run a word-count Storm application.

https://drive.google.com/file/d/0B6706xGNaPPyZ1l4bjFUMHJWdkk/view?usp=sharing

The following command creates an installation of the `wordcount` example.

```
$sparse quickstart wordcount
```

Assuming you know the structure of topology definitions and the actual spout and bolt as explained in Async videos, run the word-count example use the following commands:

```
$cd wordcount
$sparse run
```

## Step 2: Implementation of a Tweet Word-Count Topology

In this step, your task is to use the following topology to create *one spout* and *two bolts* that parse the tweets and *one bolt* that counts the number of a given word in a tweet stream.

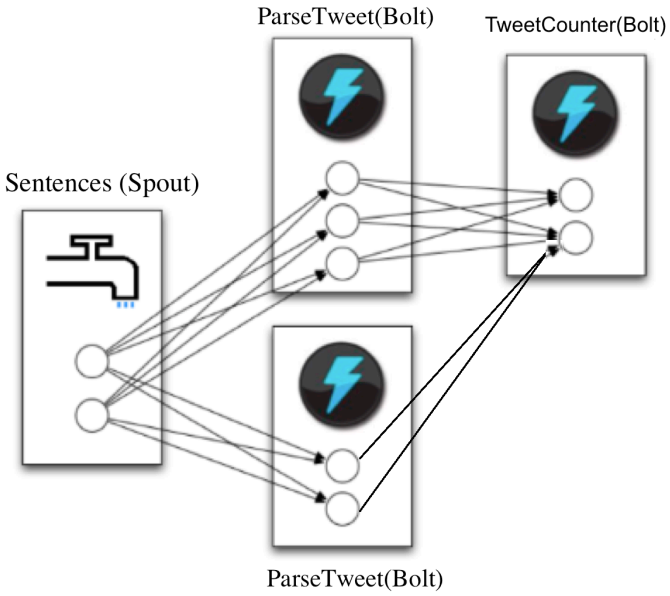


Figure 1: Task Topology

Create a project by running the following command:

```
$sparse quickstart tweetcount
```

This command provides a basic `wordcount` topology example, as seen in Step 1. You can modify this topology according to Figure 1 by modifying the file `wordcount.clj` in `tweetcount/topologies/`.

When constructing your topology it is important to remember that the topology is a function definition. This function must return an array with two dictionaries and take one argument called `options`. The first dictionary holds a named mapping of all the spouts that exist in the topology; the second holds a named mapping of all the bolts. Observe that the array is defined with square brackets [], and each dictionary is defined as a list with curly brackets {}. The options argument contains a mapping of topology settings.

You need to make sure that the topology and code are consistent with respect to the names for emitted tuples, dependencies, and so on. Be careful because starting a topology can take a long time. If you have multiple bolts, the structure is similar to the following code. Observe that when referring to the spout or bolt, the first component is the filename `<filename>` and the second the class name `<classname>`. The following snippet is only an outline and not a fully functional example:

```
(:use   [streamparse.specs])
 (:gen-class))

(defn tweetcount [options]
  [
  ;; spout configuration
      {"X-spout" (python-spout-spec
     options
     "spouts.<filename>.<classname>"
     [<emitted name>]
     )      }
  ;; bolt configuration 1
      {"Y-bolt"
            …
  ;; bolt configuration 2
      "Z-bolt"
           …
      }
 ]
)
```

## Code Base

The code snippets that you can use for your spout and bolts follow. Remove all the `words.py` from your spouts directory and `wordcount.py` from your bolts folder in `tweetcount/src/`.

**Spout Name**: Sentences(Spout)
Create a file called `sentences.py` using the following sample code. This is the spout code that will continuously generate tweet-like data.

```
from __future__ import absolute_import,print_function, unicode_literals
import itertools
from streamparse.spout import Spout
class Sentences(Spout):
  def initialize(self, stormconf, context):
    self.sentences = [
"She advised him to take a long holiday, so he
immediately quit work and took a trip around the world",
      "I was very glad to get a present from her",
      "He will be here in half an hour",
      "She saw him eating a sandwich",
    ]
    self.sentences = itertools.cycle(self.sentences)
  def next_tuple(self):
    sentence = next(self.sentences)
    self.emit([sentence])
  def ack(self, tup_id):
    pass # if a tuple is processed properly, do nothing
```

```
  def fail(self, tup_id):
    pass # if a tuple fails to process, do nothing
```

This Storm spout has the following methods:

- `initialize`: Initializes the storm spout and generates the data.
- `next_tuple`: Passes the events to bolts one by one.
- `ack`: Acknowledges the event delivery success.
- `fail`: If event fails to deliver to bolts, this method will be called.

Now you can put `sentences.py` into the `/src/spouts/` directory.

**Bolt 1 Name**: ParseTweet(Bolt)
This bolt will capture the input coming from the `Sentences` spout, filter out specific formats, and pass it to the next bolt of the topology, called `tweetcount`. Create a file called `parse.py` using the following sample code:

```
from __future__ import absolute_import, print_function,unicode_literals
import re
from streamparse.bolt import Bolt
def ascii_string(s):
 return all(ord(c) < 128 for c in s)
class ParseTweet(Bolt):
  def process(self, tup):
    tweet = tup.values[0] # extract the tweet
    # Split the tweet into words
    words = tweet.split()
    valid_words = []
    for word in words:
      if word.startswith("#"): continue
      # Filter the user mentions
      if word.startswith("@"): continue
      # Filter out retweet tags
      if word.startswith("RT"): continue
      # Filter out the urls
      if word.startswith("http"): continue
      # Strip leading and lagging punctuations
      aword = word.strip("\"?><,'.:;)")
      # now check if the word contains only ascii
      if len(aword) > 0 and ascii_string(word):
        valid_words.append([aword])
    if not valid_words: return
    # Emit all the words
    self.emit_many(valid_words)
    # tuple acknowledgment is handled automatically.
```

`ParseTweet`(bolt) will filter out input data that represents URLs, user mentions, hash tags, and so on and will emit each word to the tweet-count bolt.

`ParseTweet` bolt methods:

- `process`: Actual programming logic is applied in this method.
- Tuple acknowledgment is handled automatically.

**Bolt 2 Name**: TweetCounter(Bolt)

This bolt will capture the input coming from the `ParseTweet` bolt, update the count of a given input word, and print the result into a log with the format `self.log('%s: %d' % (word, self.counts[word]))`. Create a file named `tweetcounter.py` using the following sample code:

```
from __future__ import absolute_import, print_function,
unicode_literals
from collections import Counter
from streamparse.bolt import Bolt

class TweetCounter(Bolt):
  def initialize(self, conf, ctx):
    self.counts = Counter()

  def process(self, tup):
    word = tup.values[0]
    # Increment the local count
    self.counts[word] += 1
    self.emit([word, self.counts[word]])
    # Log the count - just to see the topology running
    self.log('%s: %d' % (word, self.counts[word]))
```

`TweetCounter` bolt methods:

- `initialize`: Initializes the bolt method with required variable initialization.
- `process`: Actual programming logic is applied in this method.
- Tuple acknowledgment is handled automatically.

Now you can put both `parse.py` and `tweetcounter.py` into your `bolts/` directory.

### Run the Storm Application

The final step is to run your application. You need to go inside `tweetcount` folder and run:

```
$cd tweetcount
$sparse run
```

## Submission

Submit a PDF that includes your topology file based on Figure 1 (`wordcount.clj`) and a screenshot of your running application that shows the stream of tweet counts on screen.