Spark/(i)Python/NLTK EMR cluster for NLP

This document describes how to create and use an Apache Spark cluster for Python-based (and iPython) natural language processing tasks using the NLTK library. The Spark cluster is based on Amazon's (AWS) Elastic Map Reduce (EMR) service. A PDF conversion step is also included.

Table of Contents

S3 Storage Setup

In this demonstrations S3 storage is used for:

- Keeping custom bootstrap actions for EMR/Spark/iPython cluster
- As a staging area for incoming documents to be processed on the cluster
- As an output area for tasks/jobs executed on the cluster
- As storage for PDF converter Lambda function (optional)

For simplicity, all of the above will be served by a single S3 bucket (called **`spnei-demo`**).

To create S3 bucket:

1. Sign in to AWS console
2. Navigate to “S3 - Scalable Storage in the Cloud”
3. Choose “Create Bucket”
4. Enter bucket name (e.g. “**`spnei-demo`**”) and pick “**`US Standard`**” region
5. Click “**`Create`**”

Uploading sample data for processing

While we’re in S3 dashboard, we can also upload sample data to be processed on our cluster.
For the purpose of this exercise, let take the following OReilly ebook:
http://www.oreilly.com/data/free/files/data-driven.pdf

Alternatively, the demoer can upload a file using this simple tool at:
http://gmelli-web.s3-website-us-east-1.amazonaws.com

Sample PDF Corpus

There are several public PDF repositories in S3.
We will use `s3n://og-data-public /budgets/unorganized/*.pdf`
(visible here http://og-data-public.s3.amazonaws.com/ ideally with an XML/JSON formatter/extension)

Set-up EMR Cluster

This section assumes the creation of a cluster in the us-east-1 region.

For short-lived clusters such as this demo project, we will use EC2 Classic space. However, for production grade clusters, which require higher level of security, it's advisable to provision EMR cluster in a VPC. VPC setup and configuration is outside the scope of this project.

Uploading custom bootstrap action

To create an EMR/Spark cluster with iPython notebook (and NLTK) support we will utilize a custom bootstrap action. Bootstrap actions simplify the addition of libraries, modules as well as any other additional dependencies required on the cluster at runtime. Typically, bootstrap action is a shell script which will execute on every node (master and all core nodes) of the EMR cluster at provision time. The script can perform all the additional tasks needed to bring the cluster to the level of packages/modules/software required.

Copy the script file below into a file. Name it "**`SPNEi-bootstrap.sh`**"

```
#!/bin/bash

# EMR Bootstrap Action Script
# - It prepares the cluster for an NLP demo
# - Version: 160716

# Prepare the Unix shell environment
export PATH="$PATH:/usr/local/bin"
. /home/hadoop/.bashrc

# Upgrade pip, restore softlink to /usr/bin (seems to break during upgrade)
sudo pip install --upgrade pip
sudo ln -s /usr/local/bin/pip /usr/bin/pip
sudo pip install s3cmd  # optional, allows command line S3 access

# Any additional python packages/modules can go here
sudo pip install jupyter
# Typical python data science
sudo pip install matplotlib
sudo pip install pandas
sudo pip install matplotlib
sudo pip install scikit-learn
# Typical python NLP
sudo pip install textblob
# supporting services
sudo pip install boto3
sudo pip install pdfminer

# Deploy jupyter
mkdir -p /emr/jupyter
cd /emr/jupyter

# Add spark kernel
aws s3 cp s3://elasticmapreduce.bootstrapactions/jupyter/kernel.json .
aws s3 cp s3://elasticmapreduce.bootstrapactions/jupyter/spark-kernel.tar.gz .
```

```
#Generate Config for jupyter
/usr/local/bin/jupyter notebook --generate-config

echo "c = get_config()" >  /home/hadoop/.jupyter/jupyter_notebook_config.py
echo "c.NotebookApp.ip = '*'" >>  /home/hadoop/.jupyter/jupyter_notebook_config.py
echo "c.NotebookApp.open_browser = False"  >>  /home/hadoop/.jupyter/jupyter_notebook_config.py
echo "c.NotebookApp.port =  8192" >>  /home/hadoop/.jupyter/jupyter_notebook_config.py

# Generate pyspark Config the old style for iPython
ipython profile create pyspark
cat <<END1 >>/home/hadoop/.ipython/profile_pyspark/ipython_notebook_config.py
c = get_config()
c.NotebookApp.ip = '*'
c.NotebookApp.open_browser = False
c.NotebookApp.port =  8192
END1

# Add pyspark setup script
cat <<END2 >>/home/hadoop/.ipython/profile_pyspark/startup/00-pyspark-setup.py
import os
import sys
spark_home = os.environ.get('SPARK_HOME', None)
sys.path.insert(0, spark_home + "/python")
# Add the py4j to the path.
# You may need to change the version number to match your install

sys.path.insert(0, os.path.join(spark_home, 'python/lib/py4j-0.9-src.zip'))
# Initialize PySpark to predefine the SparkContext variable 'sc'
execfile(os.path.join(spark_home, 'python/pyspark/shell.py'))
END2

# Define pyspark kernel
mkdir -p /home/hadoop/.ipython/kernels/pyspark
cat <<END3 >>/home/hadoop/.ipython/kernels/pyspark/kernel.json
{
 "display_name": "pySpark (Spark 1.6.1)",
 "language": "python",
 "argv": [
  "/usr/bin/python",
  "-m",
  "IPython.kernel",
  "--profile=pyspark",
  "-f",
  "{connection_file}"
 ]
}
END3

# Environment vars required for pyspark and jupyter
export SPARK_HOME=/usr/lib/spark
#export PYSPARK_SUBMIT_ARGS="--master local[2] pyspark-shell"
export PYSPARK_SUBMIT_ARGS="--master yarn-client pyspark-shell"
export JUPYTER_CONFIG_DIR=/home/hadoop/.ipython/profile_pyspark

echo "Untar spark Kernel and move config"
tar -xzf spark-kernel.tar.gz
mkdir -p /home/hadoop/.ipython/kernels/spark/
cp kernel.json /home/hadoop/.ipython/kernels/spark/

# Start jupyter notebook
cd /home/hadoop/
#nohup /usr/local/bin/jupyter notebook > /emr/jupyter/jupyter.log &
nohup /usr/bin/python2.7 /usr/local/bin/jupyter-notebook > /emr/jupyter/jupyter.log &
```

```
# Download and install nltk data libraries
sudo mkdir /mnt1/py
sudo ln -s /mnt1/py /usr/share/nltk_data
sudo /usr/bin/python2.7 -m nltk.downloader -d /usr/share/nltk_data all
```

The script draws from this blog post (plus modifications for EMRv4 and jupyter): http://blogs.aws.amazon.com/bigdata/post/TxX4BY5T1PQ7BQ/Using-IPython-Notebook-to-Analyze-Data-with-Amazon-EMR

Upload the script to the S3 bucket.

1. Open "**`speni-demo`**" bucket (e.g. https://console.aws.amazon.com/s3/home?region=us-east-1&bucket=**speni-demo**)
2. Click "Upload"
3. Click "Add files"
4. Pick the "**`speni-bootstrap.sh`**"
5. And then "Start upload" (at the bottom right corner)

NLTK on EMR (local storage requirements)

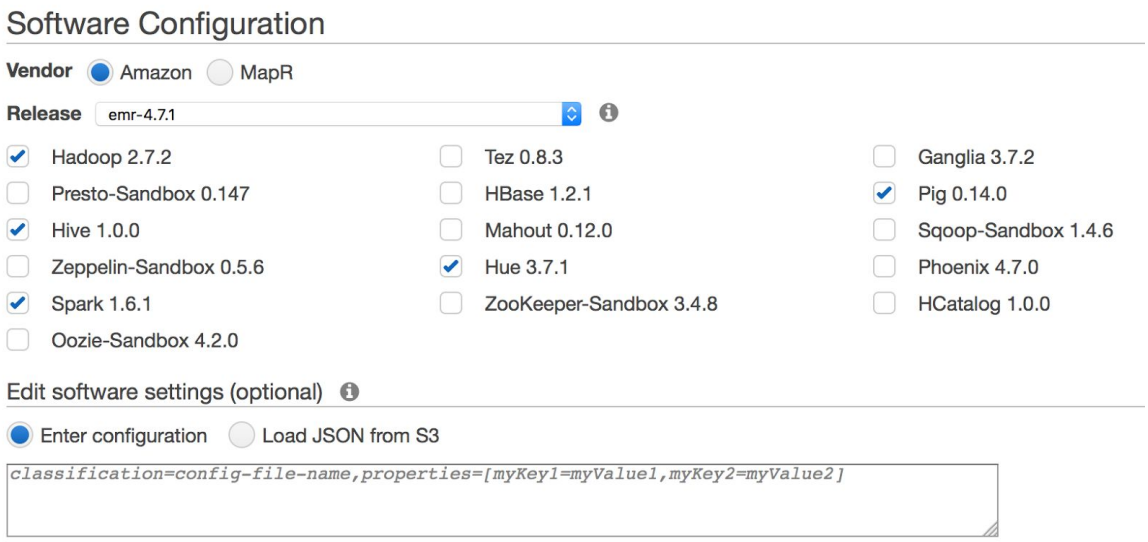
NLTK includes extensive data libraries. However, each AWS EMR cluster node comes with limited (~80 GiB) and uncustomizable root volume space. To download and install NLTK data libraries the last few lines of the script tap into local ephemeral storage allocated for HDFS. A soft link to ephemeral volume from `/usr/share/nltk_data` will be used to store NLTK data libraries. Because of the use of ephemeral storage, make sure that the master and worker node instance types come with ephemeral storage. (See: "SSD storage" in aws.amazon.com/ec2/instance-types/)

Create Cluster with AWS's Web Console

In AWS console, navigate to EMR dashboard (http://console.aws.amazon.com/elasticmapreduce/home?region=us-east-1#)
Click "Create Cluster"
Choose "**`Advanced options`**"

Step 1: Software and Steps

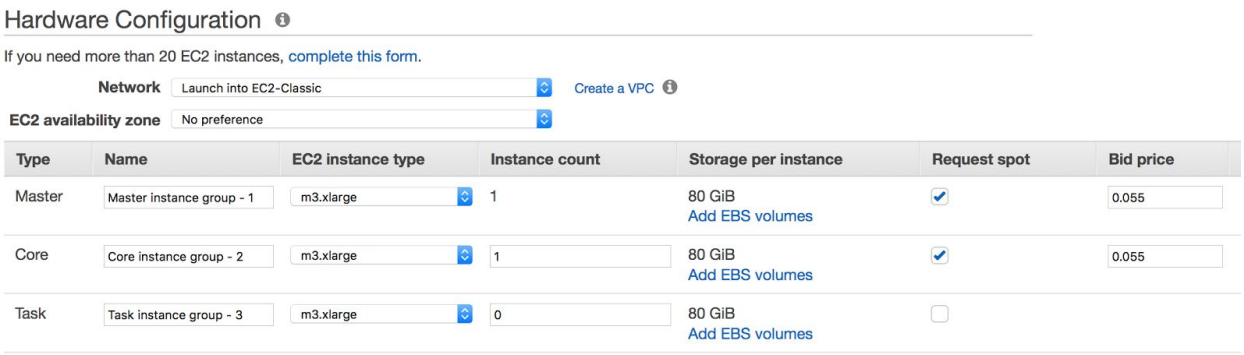
- Select Release: AMI Version **`emr-4.7.1`**
- Select "**`Spark 1.6.1`**" (no additional configuration required)



- Click "Next" (to "Step 2: Hardware")

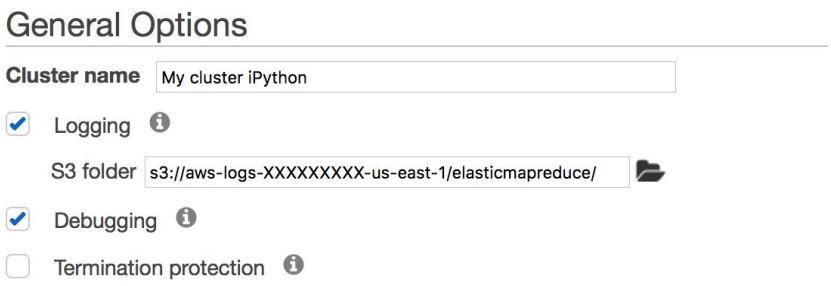
Step 2: Hardware

- Master: Instance_type: m3.xlarge (remember to pick instances with built in storage)
- Core: Instance_type: m3.xlarge Instance_count:1
- Task: Instance_type: m3.xlarge Instance_count:0
- Choose "Request Spot", use "i" icon to see current prices and bid accordingly
  (Note: Instance Group is useful when running high performance production workflows. Durig "shuffle" phase of Hadoop EMR, it helps to have all nodes on the same "rack" to reduce data transfer latencies. Placing instances into a Group allows to achieve just that)
- Click "Next" (to "Step 3: General Cluster)

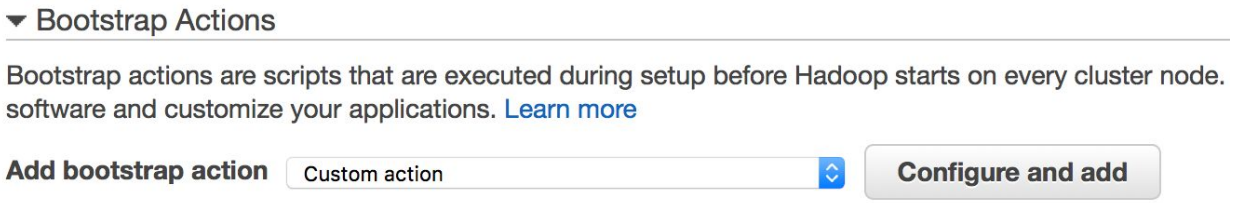


Step 3: General Cluster

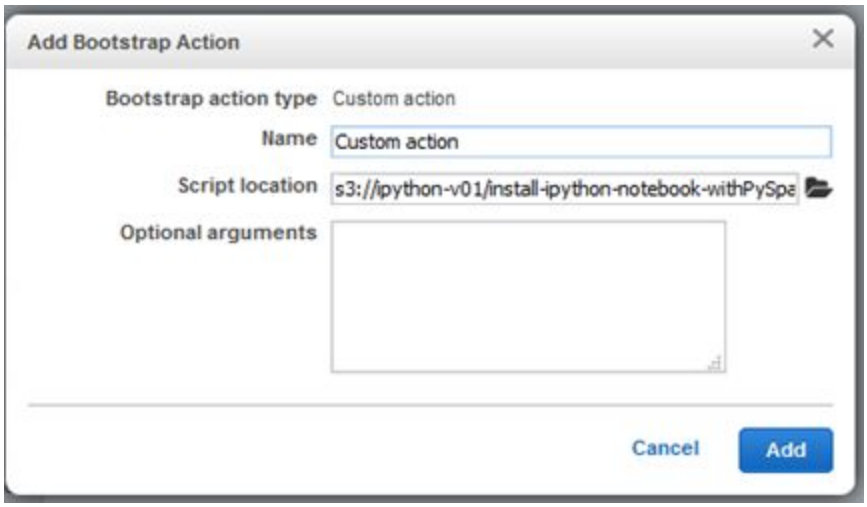
- Name the cluster (e.g. "**`SPNEi`**-160716")
- Uncheck "Termination Protection" (only useful in production)
- (Accept default S3 folder for logging)



- At the bottom, in Bootstrap Actions choose "`Custom action`" and select "`Configure and add`"



- In "**`Script location`**", enter the full path to the bootstrap script (including S3 bucket name) for example, **`s3://speni-demo/speni-bootstrap.sh`**



- Click “**`Add`**” and then Click "**`Next`**" (to "Step 4: Security")

Step 4: Security

- Choose your EC2 Key Pair (it will be used for SSH tunneling).
- Default EMR* roles are good choice for the demo cluster. They will ensure the cluster has access to S3 buckets (for reading and writing data).
- Click “Create cluster”

Create the Cluster with AWS's CLI Unix Tool

This section presents a poweruser method to create the cluster, via AWS' command line interface (CLI) tool.

Install and configure the CLI tool,
if you have not yet done so.

```
ubuntu@/home/ubuntu:~$ \
       sudo pip install awscli

# if you have not yet configured the tool
ubuntu@/home/ubuntu:~$ \
       aws configure
       AWS Access Key ID [****************222Q]:
       AWS Secret Access Key [****************JdF/]:
       Default region name [us-east-1]:
       Default output format [None]:
```

Next,
request the creation of a cluster

```
# ubuntu@/home/ubuntu/:~$ \
       aws emr create-cluster \
          --name 'SPENi' \
          --applications Name=Hadoop Name=Spark Name=Ganglia \
          --release-label emr-4.7.1 \
          --region us-east-1 \
          --tags Name=EMR \
          --bootstrap-actions '[{"Path":"s3://speni-demo/speni-bootstrap.sh","Name":"Custom
       action"}]' \
          --ec2-attributes
       '{"KeyName":"GM-EMR","InstanceProfile":"EMR_EC2_DefaultRole","AvailabilityZone":"us-east-1a",
       "EmrManagedSlaveSecurityGroup":"sg-b6fb13a0","EmrManagedMasterSecurityGroup":"sg-4efb1358"}'
       \
          --service-role EMR_DefaultRole \
          --enable-debugging \
          --log-uri 's3n://aws-logs-747579156672-us-east-1/elasticmapreduce/' \
          --instance-groups
       '[{"InstanceCount":1,"BidPrice":"0.10","InstanceGroupType":"CORE","InstanceType":"m3.xlarge",
       "Name":"Core instance group -
       2"},{"InstanceCount":1,"BidPrice":"0.10","InstanceGroupType":"MASTER","InstanceType":"m3.xlar
       ge","Name":"Master instance group - 1"}]'

       # for future reference
       # --termination-protected
       # --use-default-roles
```

This command should output the cluster id.
For example:

```
       {
              "ClusterId": "j-2E9XITP7P9TNT"
       }
```

This ClusterId, can be used to query the cluster.
For example on the AWS web console

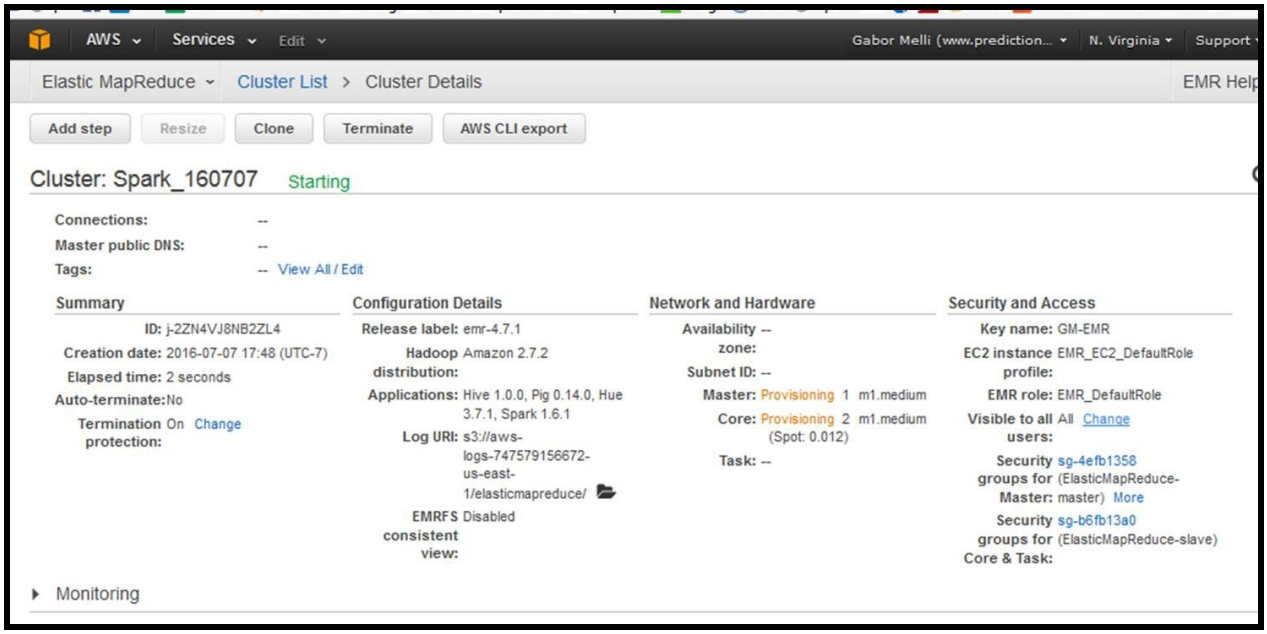
https://console.aws.amazon.com/elasticmapreduce/home?region=us-east-1#cluster-details:**j-2E9XITP7P9TNT**

Or,
from the command line using AWS's CLI

```
$ aws emr describe-cluster --cluster-id j-1TCRZQMPC9R64
```

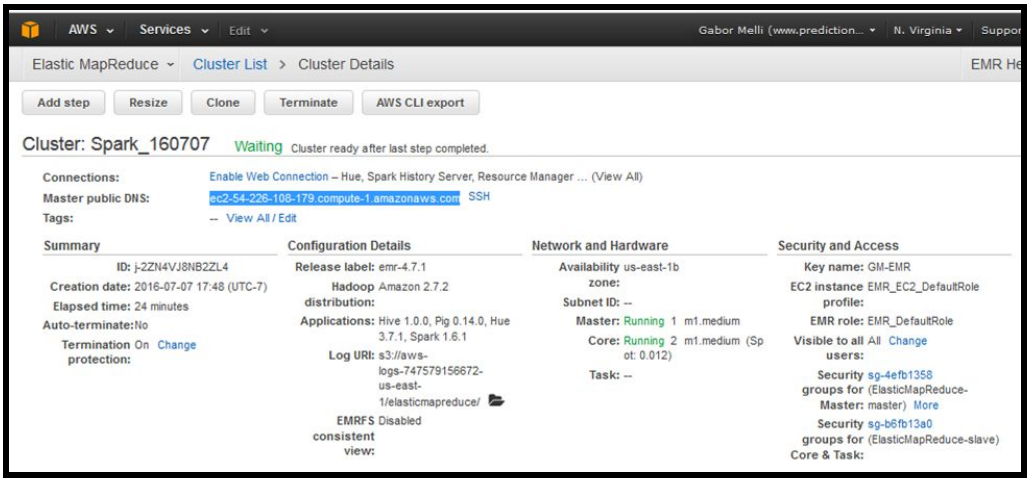
Get the Public Master DNS Address

Wait for the cluster to be provisioned. It' status will change to "Waiting" (for jobs) or "Running" when it's ready.
It takes on average 20-25 minutes for the cluster (with spot instances of this size) to fully initialize.



Once provisioned locate the Master public DNS

E.g. the cluster below the address was: **ec2-54-226-108-179.compute-1.amazonaws.com**.



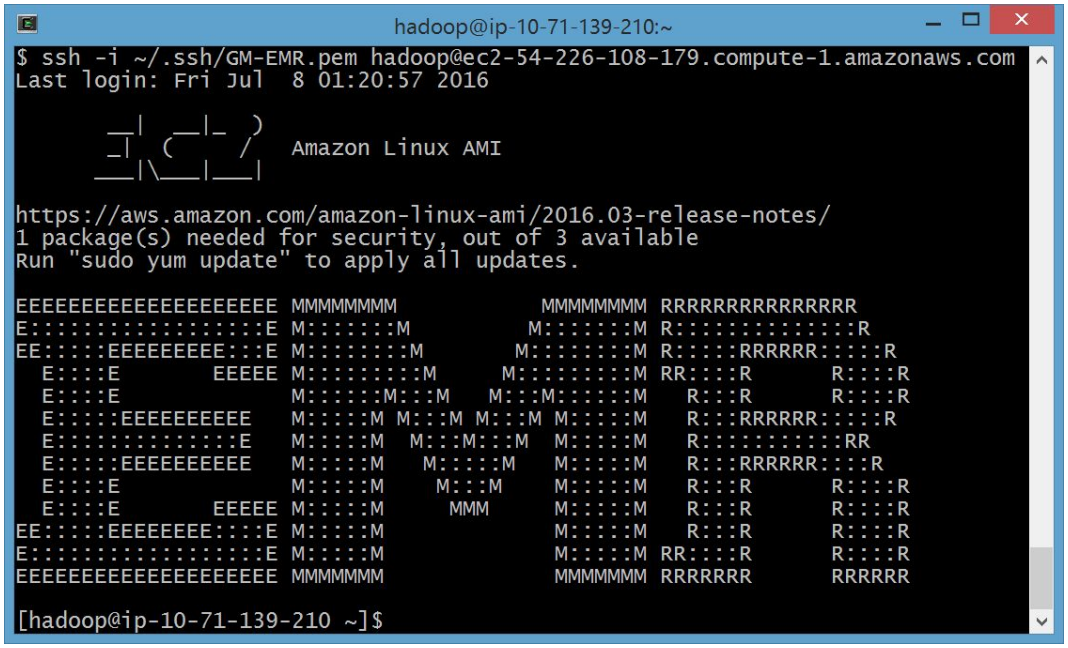
Or from the command line using AWS's CLI

```
$ aws emr describe-cluster --cluster-id j-1TCRZQMPC9R64 | grep -i MasterPublicDnsName
      "MasterPublicDnsName": "ec2-54-211-23-27.compute-1.amazonaws.com",
```

You can now connect to the Amazon EMR master node using SSH to run interactive queries, examine log files, submit Linux commands, and so on. Learn more.

Test the connection by logging into the server.

E.g. `$ EMR_Master=ec2-54-226-108-179.compute-1.amazonaws.com`
`$ ssh -i ~/.ssh/GM-EMR.pem hadoop@${EMR_Master}`



Adding Python Packages

Python packages should be added by updating the custom bootstrap action described earlier (by adding **`sudo pip install`** … command into the “Installing dependencies” block (which already has few pip installs). You will have to re-upload the updated bootstrap script to S3, and re-provision the cluster.

Alternatively, additional modules can be loaded directly in the iPython notebook as follows:

```
! pip install <module/package>
```

Connect to the Spark Cluster

A secure way to interact with the Spark cluster from your computer's web browser is via SSH tunneling. This section describes the setup of SSH tunneling that is restricted to only one port/application. In our case that port connects to the Jupyter/iPython service on the cluster. A more flexible arrangement that is not limited to a single port (e.g. to also view the Spark manager and/or Ganglia) look for the section below on "Dynamic" tunneling.

Suppose that the cluster's

- EMR public address is **`ec2-54-211-23-27.compute-1.amazonaws.com`**
- EMR EC2 private access key file is called **`myemr.pem`** (and is stored in your ~/.ssh directory).
- Recall from above that we have selected 8192 as iPython's port
- Finally, we have also selected **`8192`** as the communication port on your local computer.

Establish SSH tunnel enter the following command:

```
$ EMR_Master_DNS=ec2-54-211-23-27.compute-1.amazonaws.com
$ SSH_Tunnel_Port=8192
$ EMR_Port=8192
$ ssh -vvv \
   -o ServerAliveInterval=10 \
   -i ~/.ssh/myemr.pem \
   -N -L ${SSH_Tunnel_Port}:${EMR_Master_DNS}:${EMR_Port} \
   hadoop@${EMR_Master}
```

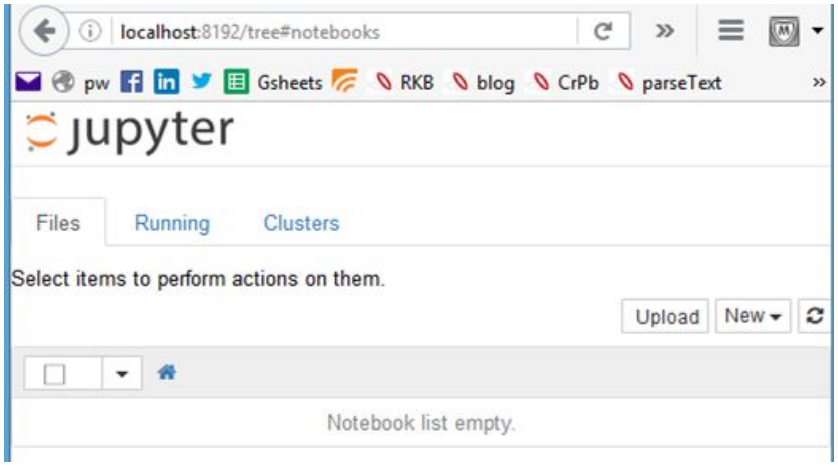
If prompted, answer "yes".
The command will persist while the "SSH tunnel" is in place.

```
The authenticity of host 'ec2-54-211-23-27.compute-1.amazonaws.com (54.211.23.27)' can't be
established.
ECDSA key fingerprint is SHA256:CHCa110Ajis1SWmpbYkOc6Ysl0WR3/EaPhO3OnstPts.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-54-211-23-27.compute-1.amazonaws.com,54.211.23.27' (ECDSA) to
the list of known hosts.
```

Finally, to connect to the iPython service, point your browser to

http://127.0.0.1:8192/tree

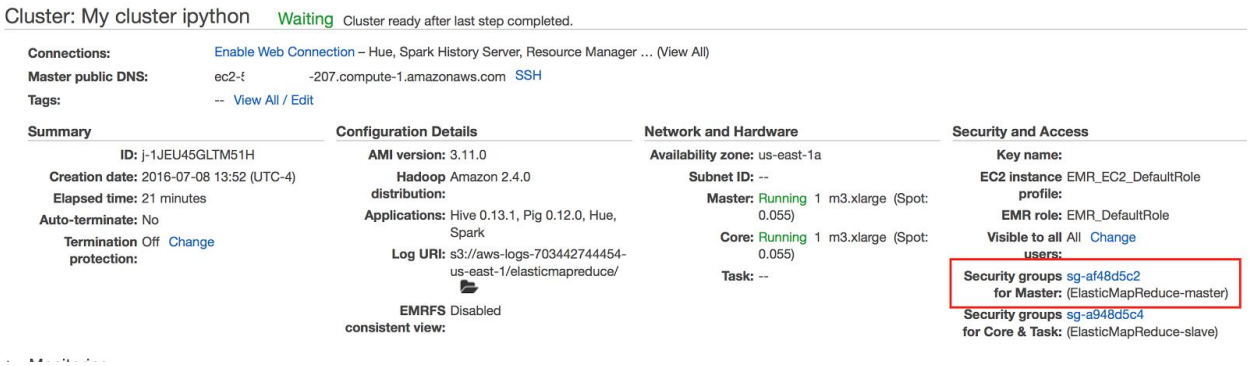
Your notebook should load on your browser, and appear roughly as follows:



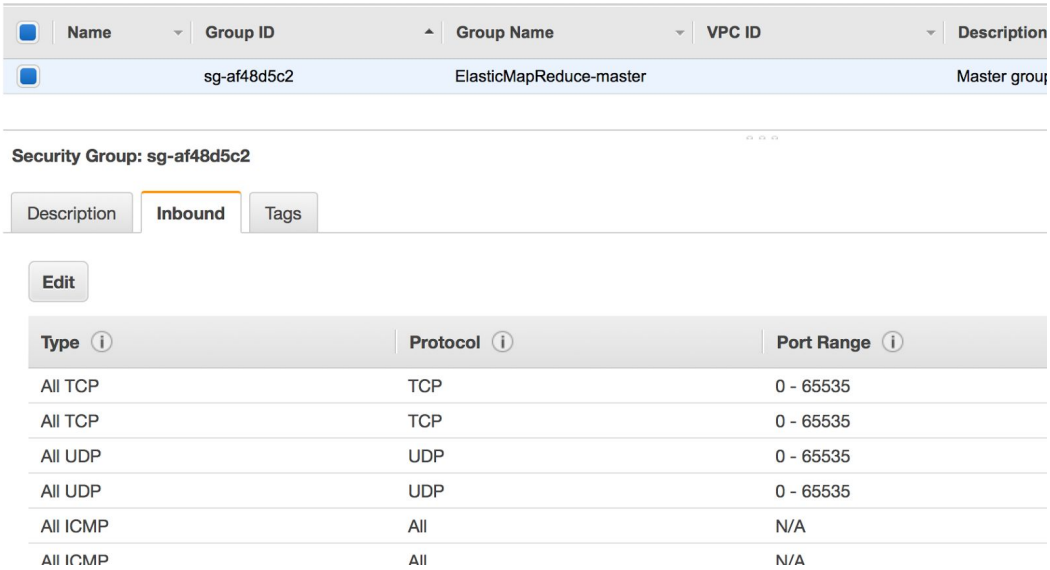
Alternative Security

If you were unable to establish SSH tunneling to your instance, you can try to access your iPython notebook by directly connecting to port 8192 of the Master instance of your EMR cluster. Although, a modification to its security group will be needed to allow that.

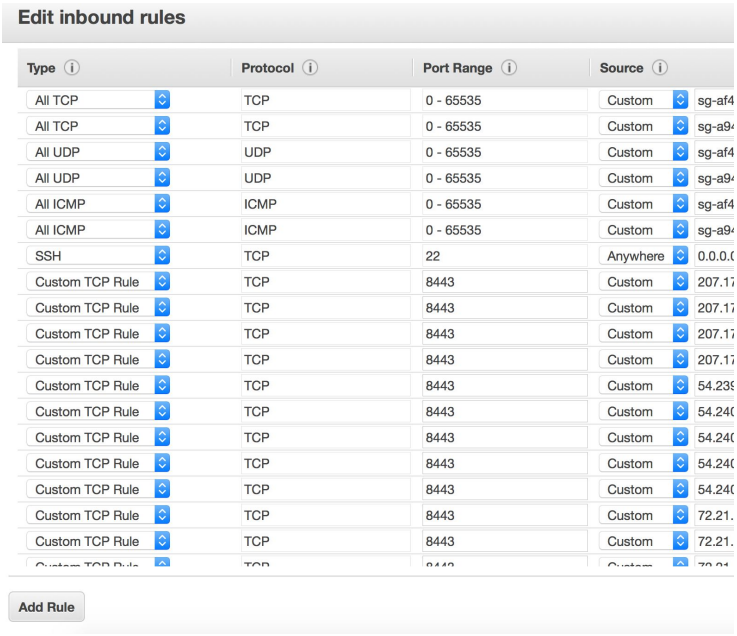
In the main EMR cluster details screen, locate "**`Security groups for Master`**": sg-xxxxxxx



Click on the group id, and you will be taken to the Security Groups screen.



Choose “Inbound” tab, then hit “Edit” right underneath. On the next screen (“Edit inbound rules”), hit “Add rule” button at the bottom.



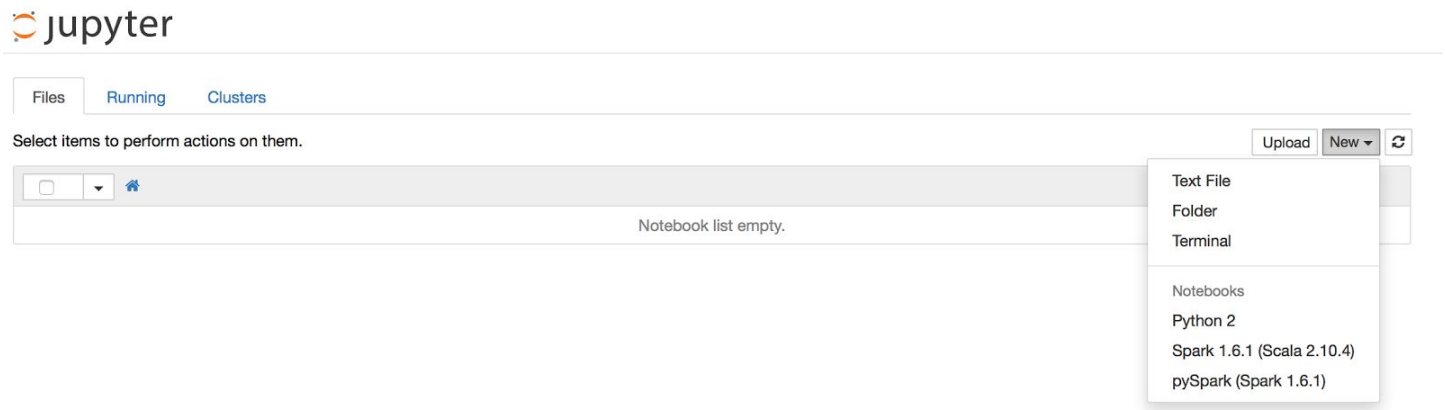
This will insert additional rule at the bottom of the list:

Pick “Custom TPC Rule” TCP, enter port 8192, pick “My IP” which should populate the field next to it with your IP. Next, hit “Save” and you should be able to access your notebook by using public address of your EMR cluster (which you used for EMR access testing):
http://ec2-54-XXX-XXX-207.compute-1.amazonaws.com:8192

Using iPython

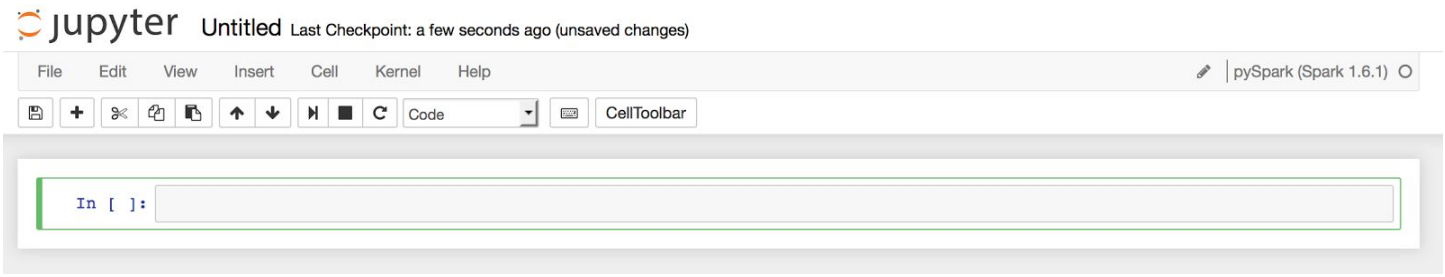
Creating a new iPython notebook:

To create a brand new iPython notebook use the "**`New`**" pull-down menu in the right hand corner.
Select "**`pySpark (Spark 1.6.1)`**".



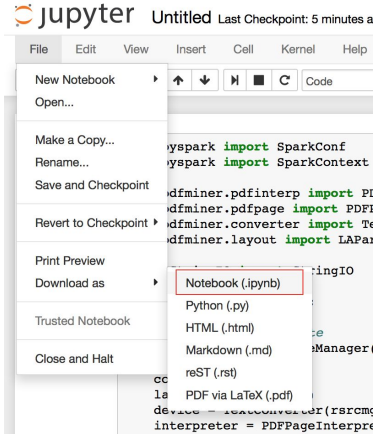
(btw, the Spark/Scala kernel may also be available, but this is not demonstrated in this document)

The pySpark notebook itself will look like this:



Saving work locally

Saving work to your local workstation (as opposed to keeping it on EMR cluster) can be done via `File->Download As->Notebook` (into an .ipynb file)



Uploading work back to the cluster

In order to upload the work previously saved locally back to EMR cluster, from the main screen choose Upload

Once uploaded, your notebook file will show up in the main catalog (similar to the “Untitled” notebook above). Simply click on the uploaded file to open it in the corresponding notebook.

Notes on Spark/EMR and iPython

When running a Spark job in notebook you're somewhat limited to enforcing number of executors/cores. Spark assigns each job default number of executors/cores and then what happens inside that job is limited to that. So if the notebook is started with 2 executors default then all tasks will execute in that restriction. One way to get around this restriction is to submit the job from a command line so that spark and yarn defaults are used and which can manually overridden (as described in Appendix B).

Examples of Performing Jobs

This section includes some sample Spark jobs that become every more complex to ultimately show an NLP task.

Cut/paste the code below into the iPython notebook.

Sample Job 1 - Estimating pi

This is a standard (non-NLP) script to test the cluster
It is based on http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-spark-application.html#d0e42771

```
import sys
from random import random
from operator import add
from pyspark import SparkContext

if __name__ == "__main__":
    """
        Usage: pi [partitions]
    """
    #sc = SparkContext(appName="PythonPi")
    #partitions = int(sys.argv[1]) if len(sys.argv) > 1 else 2
    partitions = 2
    n = 100000 * partitions

    def f(_):
        x = random() * 2 - 1
        y = random() * 2 - 1
        return 1 if x ** 2 + y ** 2 < 1 else 0

    count = sc.parallelize(xrange(1, n + 1), partitions).map(f).reduce(add)
    print "Pi is roughly %f" % (4.0 * count / n)

    sc.stop()
```

Another sample Job
http://spark.apache.org/examples.html

```
def sample(p):
        x, y = random(), random()
        return 1 if x*x + y*y < 1 else 0

NUM_SAMPLES=100000000
count = sc.parallelize(xrange(0, NUM_SAMPLES)).map(sample) \
             .reduce(lambda a, b: a + b)
print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
```

Sample Job 2 - Copy the pdf file locally

This job ingests a pre-staged PDF file, converts it to text, and then uses Spark with NLTK to identify parts of speech for every word. The result will be saved back to S3.

#

Step 1 - define PDF miner based function for converting PDF to text:

```
from pyspark import SparkConf
from pyspark import SparkContext

from pdfminer.pdfinterp import PDFResourceManager, PDFPageInterpreter#process_pdf
from pdfminer.pdfpage import PDFPage
from pdfminer.converter import TextConverter
from pdfminer.layout import LAParams

from cStringIO import StringIO

import boto3

import nltk

def pdf_to_text(pdfname):

    # PDFMiner boilerplate
    rsrcmgr = PDFResourceManager()
    sio = StringIO()
    codec = 'utf-8'
    laparams = LAParams()
    device = TextConverter(rsrcmgr, sio, codec=codec, laparams=laparams)
    interpreter = PDFPageInterpreter(rsrcmgr, device)

    # Extract text
    fp = file(pdfname, 'rb')
    for page in PDFPage.get_pages(fp):
        interpreter.process_page(page)
    fp.close()

    # Get text from StringIO
    text = sio.getvalue()

    # Cleanup
    device.close()
    sio.close()

    return text
```

#

Step 2 - In this step, we will import the PDF file from S3, into our notebook, storing on the local node to /tmp

```
s3 = boto3.resource('s3')
s3.Object('ipythonv01', 'data-driven.pdf').download_file( "/tmp/data-driven.pdf")
```

#

Step 3 - Now that the file was stored locally, we can call PDF converter (to extract text from the PDF):

```
data=pdf_to_text('/tmp/data-driven.pdf')
```

#

Step 4 - Next, we will convert “data” string variable (which has the content of our PDF converted to text) to Spark RDD:

```
lines = sc.parallelize(data.split())
```

#

Step 5 - Now we can invoke nltk part of speech tagging against our RDD:

```
words = lines.flatMap(lambda x: nltk.word_tokenize(x))
print words.take(100) #10

pos_word = words.map(lambda x: nltk.pos_tag([x]))
print pos_word.take(100) #5
```

First print statement will output 100 sample words from the data we extracted:

```
['Data', 'Driven', 'Creating', 'a', 'Data', 'Culture', 'DJ', 'Patil', 'and', 'Hilary', 'Mason', 'Data', 'Driven', 'by
', 'DJ', 'Patil', 'and', 'Hilary', 'Mason', 'Copyright', '\xc2\xa9', '2015', 'O\xe2\x80\x99Reilly', 'Media,', 'Inc',
'.', 'All', 'rights', 'reserved', '.', 'Printed', 'in', 'the', 'United', 'States', 'of', 'America', '.', 'Published',
 'by', 'O\xe2\x80\x99Reilly', 'Media,', 'Inc.,', '1005', 'Gravenstein', 'Highway', 'North,', 'Sebastopol,', 'CA', '95
472', '.', 'O\xe2\x80\x99Reilly', 'books', 'may', 'be', 'purchased', 'for', 'educational,', 'business,', 'or', 'sales
', 'promotional', 'use', '.', 'Online', 'editions', 'are', 'also', 'available', 'for', 'most', 'titles', '(', 'http',
 ':', '//safaribooksonline.com', ')', '.', 'For', 'more', 'sales', 'department:', '800-998-9938', 'or', 'corporate',
'@', 'oreilly.com', '.', 'corporate/institutional', 'information,', 'contact', 'our', 'Editor:', 'Timothy', 'McGovern
', 'Copyeditor:', 'Rachel', 'Monaghan', 'Interior', 'Designer:']
```

Second print statement will output NLTK tagging:

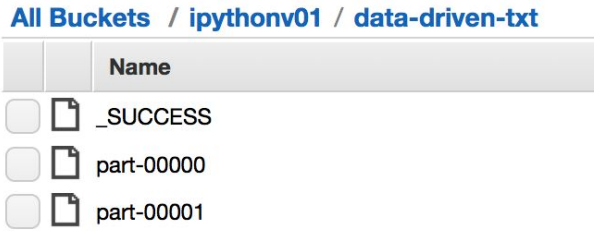
```
[[('Data', 'NNP')], [('Driven', 'RB')], [('Creating', 'VBG')], [('a', 'DT')], [('Data', 'NNP')], [('Culture', 'NN')],
 [('DJ', 'NNP')], [('Patil', 'NNP')], [('and', 'CC')], [('Hilary', 'NNP')], [('Mason', 'NNP')], [('Data', 'NNP')], [(
'Driven', 'RB')], [('by', 'IN')], [('DJ', 'NNP')], [('Patil', 'NNP')], [('and', 'CC')], [('Hilary', 'NNP')], [('Mason
', 'NNP')], [('Copyright', 'NNP')], [('\xc2\xa9', 'NN')], [('2015', 'CD')], [('O\xe2\x80\x99Reilly', 'RB')], [('Media
,', 'NN')], [('Inc', 'NNP')], [('.', '.')], [('All', 'DT')], [('rights', 'NNS')], [('reserved', 'VBD')], [('.', '.')]
, [('Printed', 'NNP')], [('in', 'IN')], [('the', 'DT')], [('United', 'NNP')], [('States', 'NNS')], [('of', 'IN')], [(
'America', 'NNP')], [('.', '.')], [('Published', 'JJ')], [('by', 'IN')], [('O\xe2\x80\x99Reilly', 'RB')], [('Media,',
 'NN')], [('Inc.,', 'NN')], [('1005', 'CD')], [('Gravenstein', 'NNP')], [('Highway', 'NN')], [('North,', 'NN')], [('S
ebastopol,', 'NN')], [('CA', 'NNP')], [('95472', 'CD')], [('.', '.')], [('O\xe2\x80\x99Reilly', 'RB')], [('books', 'N
NS')], [('may', 'MD')], [('be', 'VB')], [('purchased', 'VBD')], [('for', 'IN')], [('educational,', 'NN')], [('busines
s,', 'NN')], [('or', 'CC')], [('sales', 'NNS')], [('promotional', 'JJ')], [('use', 'NN')], [('.', '.')], [('Online',
'NNP')], [('editions', 'NNS')], [('are', 'VBP')], [('also', 'RB')], [('available', 'JJ')], [('for', 'IN')], [('most',
 'JJS')], [('titles', 'NNS')], [('(', 'NN')], [('http', 'NN')], [(':', ':')], [('//safaribooksonline.com', 'NN')], [(
')', 'NN')], [('.', '.')], [('For', 'IN')], [('more', 'JJR')], [('sales', 'NNS')], [('department:', 'NN')], [('800-99
8-9938', 'CD')], [('or', 'CC')], [('corporate', 'JJ')], [('@', 'IN')], [('oreilly.com', 'NN')], [('.', '.')], [('corp
orate/institutional', 'JJ')], [('information,', 'NN')], [('contact', 'NN')], [('our', 'PRP$')], [('Editor:', 'NN')],
[('Timothy', 'NNP')], [('McGovern', 'NNP')], [('Copyeditor:', 'NN')], [('Rachel', 'NNP')], [('Monaghan', 'NNP')], [('
Interior', 'NNP')], [('Designer:', 'NN')]]
```

#

Step 6 - Save result to S3

```
pos_word.saveAsTextFile("s3n://ipython-v01/data-driven-txt/")
#data.saveAsTextFile("s3n://ipython-v01/data-driven.txt")
```

Note, the output is a folder (slash on the end), not a file. Spark will save output in multiple files (chunks) in the provided folder.



Sample Job 3 - In-notebook PDF conversion and processing

Sample Job 2 copied PDF files from S3 to local Spark node for conversion. This will not parallelize very well. This example uses pyspark's (possibly still experimental) binaryFiles() loader and performs conversion of PDF data as if it came from a local file (while in fact, it is stored in Spark's RDD distributed computing structure)

#

Step 1 - Converter function, changed to deal with byte stream as opposed to file name

```
from pdfminer.pdfinterp import PDFResourceManager, PDFPageInterpreter#process_pdf
from pdfminer.pdfpage import PDFPage
from pdfminer.converter import TextConverter
from pdfminer.layout import LAParams
from cStringIO import StringIO

def pdf_to_text(fp):

    # PDFMiner boilerplate
    rsrcmgr = PDFResourceManager()
    sio = StringIO()
    codec = 'utf-8'
    laparams = LAParams()
    laparams = None    # had to zero it out, otherwise some PDFs caused issues
    device = TextConverter(rsrcmgr, sio, codec=codec, laparams=laparams)
    interpreter = PDFPageInterpreter(rsrcmgr, device)

    # Extract text
    for page in PDFPage.get_pages(fp):
        interpreter.process_page(page)

    # Get text from StringIO
    text = sio.getvalue()

    # Cleanup
    device.close()
    sio.close()

    return text
```

#

Step 2 - upload PDFs in parallel (this example likely points to 3 documents).

```
data = sc.binaryFiles('s3n://og-data-public/budgets/unorganized/*_NM_11.pdf')
```

#

Step 3 - Let's verify which files are being picked up (OPTIONAL)

```
data.keys().collect()
```

```
In [23]: data.keys().collect()

Out[23]: [u's3n://og-data-public/budgets/unorganized/alamogordo_NM_11.pdf',
          u's3n://og-data-public/budgets/unorganized/farmington_NM_11.pdf',
          u's3n://og-data-public/budgets/unorganized/lascruces_NM_11.pdf']
```

Spark appears to default to a single partition. In this example we are loading three files, so the last line increases the number of partitions to a more reasonable three(3).

```
        data.getNumPartitions()
        data1 = data.repartition(3)
#
#  Step 4 - The conversion (we're using "data1" RDD because of the optional re-partitioning above. When skipped, use "data" RDD instead.)

        textrdd = data1.map(lambda (x,y): (x,pdf_to_text(StringIO(y))))
#
#  Step 5 - convert textrdd to map
#  NOTE:  collectAsMap executes a Spark job and gets back the results from each partition from the workers and aggregates them with a reduce/concat phase on the driver. Depending on how many files are being converted, this step can take a long time. This can be sped up by adding more executors/nodes/cores to the cluster, etc.

        m=textrdd.collectAsMap()
#
#  Step 6 - perform the NLP and save results (change first two lines accordingly)

        out_bucket="s3n://ipython-dataexp"
        out_region = 'us-east-1'

        import nltk

        # we will use boto3 s3 bucket/files manipulations
        import boto3
        s3 = boto3.resource('s3')
        objects = list()

        # bucket name without leading s3n://
        bucket = s3.Bucket(out_bucket.split("/")[-1])

        # Check if bucket exists, create if it doesn't
        if bucket not in s3.buckets.all():
            if out_region == 'us-east-1':
                s3.create_bucket(Bucket=out_bucket.split("/")[-1])
            else:
                s3.create_bucket(Bucket=out_bucket.split("/")[-1], \
                    CreateBucketConfiguration={'LocationConstraint': out_region})

        # List bucket content, will be used later to delete existing files before output
        for object in bucket.objects.all():
            objects.append(object)

        # Iterate through map of converted PDFs, perform NLP and store results
        for k in m:

            # form new file name (replace .pdf with .txt), and produce info output
            print ("Original PDF:" + k)
            out_file=k.split("/")[-1].replace(".pdf",".txt")
            print ("NLP result:" + out_bucket + "/" + out_file)

            # delete if exists
            to_delete = [x for x in objects if x.key.startswith(out_file)]
            for o in to_delete:
                o.delete()

            # perform NLP
            dat = sc.parallelize(m[k].decode('utf-8').split())
            words = dat.flatMap(lambda x: nltk.word_tokenize(x))
```

```
pos_word = words.map(lambda x: nltk.pos_tag([x]))

# Save to output bucket
pos_word.saveAsTextFile(out_bucket + "/" + out_file)

# Output first 20 categorized words
print (pos_word.take(20))
print ""
```

Output should look as follows:

```
Original PDF:s3n://og-data-public/budgets/unorganized/lascruces_NM_11.pdf
NLP result:s3n://ipython-data-exp2/lascruces_NM_11.txt
[[(u'Table', 'NN')], [(u'of', 'IN')], [(u'Contents', 'NNS')], [(u'i', 'NN')], [(u'ii', 'NN')], [(u'iii', 'NN')], [(u'
iv', 'NN')], [(u'v', 'NN')], [(u'viii', 'NN')], [(u'xi', 'NN')], [(u'Mayor', 'NN')], [(u'and', 'CC')], [(u'City', 'NN
P')], [(u'Council', 'NN')], [(u'District', 'NNP')], [(u'Maps', 'NNS')], [(u'Executive', 'NN')], [(u'Staff', 'NN')], [
(u'City', 'NNP')], [(u'Organizational', 'JJ')]]

Original PDF:s3n://og-data-public/budgets/unorganized/alamogordo_NM_11.pdf
NLP result:s3n://ipython-data-exp2/alamogordo_NM_11.txt
[[(u'Officials', 'NNS')], [(u'CITY', 'NN')], [(u'COMMISSION', 'NN')], [(u'Ron', 'NN')], [(u'Griggs', 'NNS')], [(u',',
 ',')], [(u'Mayor', 'NN')], [(u',', ',')], [(u'District', 'NNP')], [(u'Seven', 'RB')], [(u'Ed', 'NN')], [(u'Cole', 'N
N')], [(u',', ',')], [(u'Mayor', 'NN')], [(u'Pro-Tem', 'NN')], [(u',', ',')], [(u'District', 'NNP')], [(u'Six', 'NN')
], [(u'Marion', 'NN')], [(u'L', 'NN')]]

Original PDF:s3n://og-data-public/budgets/unorganized/farmington_NM_11.pdf
NLP result:s3n://ipython-data-exp2/farmington_NM_11.txt
[[(u'Prepared', 'JJ')], [(u'by', 'IN')], [(u':', ':')], [(u'Administrative', 'JJ')], [(u'Services', 'NNS')], [(u'Depa
rtment', 'NNP')], [(u'Steve', 'NNP')], [(u'Ellison', 'NN')], [(u',', ',')], [(u'Budget', 'VB')], [(u'Officer', 'NN')]
, [(u'Molly', 'RB')], [(u'Bondow', 'NN')], [(u',', ',')], [(u'Financial', 'JJ')], [(u'Analyst', 'NN')], [(u'With', 'I
N')], [(u'Special', 'JJ')], [(u'Thanks', 'NNS')], [(u'to', 'TO')]]
```

Sample Job 4 - In-notebook PDF conversion and processing

```
#
#  Step 1 - Defines the pdf-to-text converter function (nothing happens in Spark)
      from pdfminer.pdfinterp import PDFResourceManager, PDFPageInterpreter#process_pdf
      from pdfminer.pdfpage import PDFPage
      from pdfminer.converter import TextConverter
      from pdfminer.layout import LAParams

      from cStringIO import StringIO

      def pdf_to_text(fp):

          # PDFMiner boilerplate
          rsrcmgr = PDFResourceManager()
          sio = StringIO()
          codec = 'utf-8'
          laparams = LAParams()
          laparams = None
          device = TextConverter(rsrcmgr, sio, codec=codec, laparams=laparams)
          interpreter = PDFPageInterpreter(rsrcmgr, device)

          # Extract text (catch exceptions, such as password protected files)
          try:
              for page in PDFPage.get_pages(fp, check_extractable=False):
                  interpreter.process_page(page)
          except:
              return ""

          # Get text from StringIO
          text = sio.getvalue()

          # Cleanup
          device.close()
          sio.close()

          return text
#
#  Step 2  - ingest PDFs (Spark is asked to ingest PDFs into an RDD).
      data = sc.binaryFiles('s3n://og-data-public/budgets/unorganized/*_NM_*.pdf')
#
#  Step 3  - re-partition RDD in such a way that each ingested PDF is placed in its own partition.
      document_count = len(data.keys().collect())
      print "document_count =",document_count
      data1 = data.repartition(document_count)
#
#  Step 4 - convert PDFs to text
      textrdd = data1.map(lambda (x,y): (x,pdf_to_text(StringIO(y))))
#
#  Step 5 - perform NLP
      import nltk
      words = textrdd.map(lambda (f,x): (f,nltk.word_tokenize(x.decode('utf-8'))))
      pos_word = words.map(lambda (f,x): (f,nltk.pos_tag(x)))

#
```

Step 6 - Output results (workers begin to work here).

```
import boto3
import os
out_bucket="ipython-data-exp4"
out_region = 'us-east-1'

# Check if bucket exists, create if it doesn't
s3 = boto3.resource('s3')
bucket = s3.Bucket(out_bucket)

if bucket not in s3.buckets.all():
    if out_region == 'us-east-1':
        s3.create_bucket(Bucket=out_bucket)
    else:
        s3.create_bucket(Bucket=out_bucket, \
        CreateBucketConfiguration={'LocationConstraint': out_region})

def f(i):

    s3 = boto3.resource('s3') # Connect to S3

    # output file in /tmp/ with 'txt' suffix
    out_file= str(i[0]).split("/")[-1].replace(".pdf",".txt")
    out_loc_file="/tmp/" + out_file

    # Write to the local file (in /tmp on the executor node)
    open(out_loc_file,'w').write(str(i[1]))

    # Remove files in S3 bucket if the file already exist.
    bucket = s3.Bucket(out_bucket)
    for o in bucket.objects.all():
        if str(o.key) == out_file:
            o.delete()

    # Upload local file from /tmp to S3
    s3.Object(out_bucket, out_file).upload_file(out_loc_file)

    # Remove the file from local fs (optional if we don't have a lot files)
    os.remove(out_loc_file)


# Write each element of RDD to local FS and upload to S3 (and time it)
import timeit; start_time = timeit.default_timer()
pos_word.foreach(f)
elapsed = timeit.default_timer() - start_time
print "elapsed centiseconds =", elapsed
```

.

Comments:

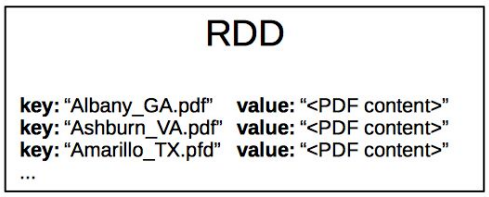
1) Benefits of this approach: 100% parallel. All actions are done on RDDs and with the exception of Step 3, where we count number of PDFs to be ingested (so that we can properly partition RDD) - there are no aggregations of any kind, which means, Master node is never the bottleneck, everything is done on local nodes.

2) The `pdf_to_text` converter function disregards password protection when allowed/possible and handles any other types of exception by simply skilling the file (instead of crashing)

3) Execution of the script:

Step 3 - `data.keys().collect()` forces first action in Spark, it instructs RDD to collect keys:

- Unlike “regular” text based RDD, binaryFiles RDD is slightly different. Instead of storing lines of text, it stores <key,value> pairs, where <key> is the name of the binary file and <value> is its content



- So, in this step, RDD is loaded and keys are collected on each Worker node and shipped to the Driver (Master Node). Usually an expensive operation, but because we’re only talking a hundred bytes per file - this only takes seconds.
- Finally, all executors/nodes are instructed to re-partition RDD in such a way that each key/value pair (basically each PDF file in our case) gets its own RDD Partition

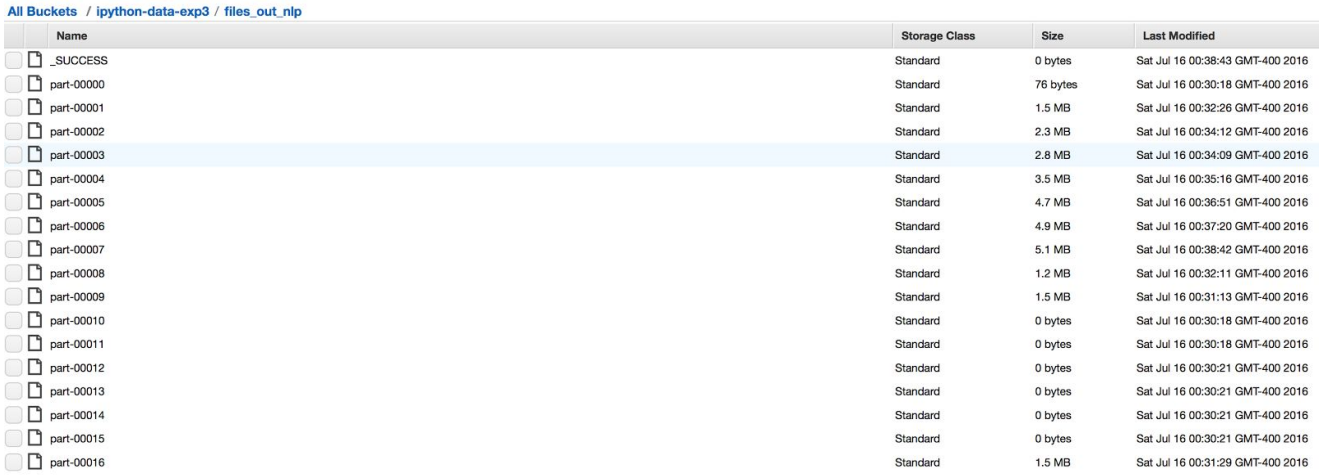
- Basically, this step controls parallelism. Each partition seems to be mapped to one executor, which means, if our RDD has only 1 partition - all files will be converted by single executor one by one. And with small number of PDFs to convert, I’m observing Spark being very conservative and doing just that. By forcing each file into its own Partition we are ensuring that all PDFs will be distributed equally among all available executors

Step 4 - We are converting each Value (of the key/value pairs in our RDD) from PDF to text. The entire pdf_to_text function is “shipped” to each Worker node (to all executors) and it will be applied locally, on PDFs that are local to each executor

Step 5 - Similar to Step 4, we’re applying two NLP functions, again, to each Value of all the pairs in our RDD. Again, this will take place locally, on each worker node.

Step 6 - Very last line of Step 6 - `pos_word.foreach(f)` , finally forces Spark to act. This command “ships” the “f” function (defined in this step) to all nodes and make them apply to all data elements. It’s just now that Spark is forced to “catch up” on the “lazy evaluation” and finally perform steps 3,4,5 and then apply function “f”. Once again, function “f” is applied on each Node, all data is local to all the nodes, there is no shuffle, no transfers to the driver/master node.

1. Files are “converted” to (and saved as) partitions:



2. Original PDF file name is stored inside each partition:

(u'**s3n://og-data-public/budgets/unorganized/albuquerque_NM_12.pdf**', [(u'Mayor', 'NNP'), (u'Richard', 'NNP'), (u'J.', 'NNP'), (u'BerryMayor', 'NNP'), (u'Richard', 'NNP'), (u'J.', 'NNP'), (u'BerryMayor', 'NNP'), (u'Richard', 'NNP'), (u'J.', 'NNP'), (u'Berry', 'NNP'), (u'Where', 'NNP'), (u'the', 'DT'), (u'money', 'NN'), (u'comes', 'VBZ'), (u'from', 'IN'), (u':', ':'), (u'And', 'CC'), (u',', ','), (u'where', 'WRB'), (u'the', 'DT'), (u'money', 'NN'), (u'goes', 'VBZ'), (u':', ':'), (u'THE', 'DT'), (u'CITY', 'NNP'), (u'OF', 'NNP'), (u'ALBUQUERQUE', 'NNP'), (u'ACKNOWLEDGES', 'NNP'), (u'IT\u2019S', 'NNP'), (u'CONTINUING', 'NNP'), (u'COMMITMENT', 'NNP'), (u'TO', 'NNP'), (u'PROTECTING', 'NNP')....

This can be addressed by a relatively simply python routine which will be executed at the end, scan the bucket and rename each file accordingly (should only take seconds).

3. This approach was to load as many PDFs into memory of the cluster as possible, work on them, store them, grab next set of PDFs. This limited number of “containers” which could work on the PDFs in parallel, because by default each container is given a lot more memory than it really needs. A possible alternative approach may be to load files one by one, but spread processing of each file among all available nodes.

4. Finally, both the cluster’s configuration as well as pyspark job’s settings can possibly be adjusted for better efficiency in terms of memory usage and parallelism

Sample Job 5 - Spark & Lambda code combined.

```
import os
import urllib
import boto3

#------------------------------
# PDF Conversion function (PDF miner)
#------------------------------

from pdfminer.pdfinterp import PDFResourceManager, PDFPageInterpreter#process_pdf
from pdfminer.pdfpage import PDFPage
from pdfminer.converter import TextConverter
from pdfminer.layout import LAParams

from cStringIO import StringIO
```

```
def pdf_to_text(fp):

    # PDFMiner boilerplate
    rsrcmgr = PDFResourceManager()
    sio = StringIO()
    codec = 'utf-8'
    laparams = LAParams()
    laparams = None
    device = TextConverter(rsrcmgr, sio, codec=codec, laparams=laparams)
    interpreter = PDFPageInterpreter(rsrcmgr, device)

    # Extract text (catch exceptions, such as password protected files)
    try:
        for page in PDFPage.get_pages(fp, check_extractable=False):
            interpreter.process_page(page)
    except:
        return ""

    # Get text from StringIO
    text = sio.getvalue()

    # Cleanup
    device.close()
    sio.close()

    return text

#------------------------------
# NLP Tasks function
#------------------------------

def do_nlp (plain_text):

    if type(plain_text) == str:
        os.environ["NLTK_DATA"] = "."       # Location of nltk_data (means "current directory")
        import nltk                          # Has to be imported after NLTL_DATA defined
        words = nltk.word_tokenize(plain_text.decode('utf-8'))
        pos_words = nltk.pos_tag(words)
    else:
        import nltk
        words = plain_text.map(lambda (f,x): (f,nltk.word_tokenize(x.decode('utf-8'))))
        pos_words = words.map(lambda (f,x): (f,nltk.pos_tag(x)))

    return pos_words


#------------------------------
# Saving to S3 function
#------------------------------

def save_to_s3(bucket, file_name, content):

    # output file in /tmp/ with 'txt' suffix
    out_rem_file= file_name.split("/")[-1].replace(".pdf",".txt")
    out_loc_file="/tmp/" + out_rem_file

    # Write to the local file (in /tmp on the executor node)
    open(out_loc_file,'w').write(content)

    # Establish connection to S3
    s3 = boto3.resource('s3')

    # Upload local file from /tmp to S3
    s3.Object(bucket, out_rem_file).upload_file(out_loc_file)
```

```
    # Remove the file from local fs (optional if we don't have a lot files)
    os.remove(out_loc_file)

#-----------------------------
# Writing out RDDs function
#-----------------------------

def save_rdds(rdd_element):

    # Output region/bucket
    out_bucket="ipython-data-exp61"
    out_region = 'us-east-1'

    # Each RDD element has file name [0] and file content [1]
    out_file=str(rdd_element[0])
    out_content=str(rdd_element[1])

    # Check if bucket exists, create if it doesn't
    s3 = boto3.resource('s3')
    bucket = s3.Bucket(out_bucket)

    if bucket not in s3.buckets.all():
        if out_region == 'us-east-1':
            s3.create_bucket(Bucket=out_bucket)
        else:
            s3.create_bucket(Bucket=out_bucket, \
                CreateBucketConfiguration={'LocationConstraint': out_region})

    # Save to S3
    save_to_s3(out_bucket, out_file, out_content)


#-------------------------------------------------------------------------
# Main (Lambda)
#-------------------------------------------------------------------------

# This is the main function (the event handler)
# It will be called every time 'S3 event' we define takes place
# We will set it up in such a way that every time a new PDF is uploaded to our S3 bucket,
# an event is triggered and dispatched to this lambda function

def lambda_handler(event, context):

    s3r = boto3.resource('s3')

    # Get the bucket name and object key from the event details
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.unquote_plus(event['Records'][0]['s3']['object']['key']).decode('utf8')
    try:
        # Extract name of the pdf file from the event
        pdf_name = event['Records'][0]['s3']['object']['key']

        # download pdf file to our local environment (container in which lambda is running)
        s3r.Object(bucket, pdf_name).download_file( "/tmp/" + pdf_name)

        # convert pdf to text, keep text in 'txt' variable
        fp = file("/tmp/" + pdf_name, 'rb')
        txt = pdf_to_text(fp)
        fp.close()

        # Perform NLP
        pos_word=do_nlp(str(txt.decode('utf-8')))
```

```
        # Save to S3
        save_to_s3(bucket, pdf_name, str(pos_word))

        print "Processed " + pdf_name
        return "success"
    except Exception as e:
        print(e)
        print('Error while processing {0} in {1}'.format(key, bucket))
        raise e



#------------------------------------------------------------------------
# Main (Spark)
#------------------------------------------------------------------------
def spark_handler():

    # Input bucket/mask
    data = sc.binaryFiles('s3n://og-data-public/budgets/unorganized/*_NM_11.pdf')

    # Re-partition according to the number of PDFs to be processed
    document_count = len(data.keys().collect())
    print "document_count =",document_count
    data1 = data.repartition(document_count)

    # PDF to text conversion
    textrdd = data1.map(lambda (x,y): (x,pdf_to_text(StringIO(y))))

    # NLP tasks
    pos_word=do_nlp(textrdd)

    # Save results back to S3
    import timeit; start_time = timeit.default_timer()
    pos_word.foreach(save_rdds)
    elapsed = timeit.default_timer() - start_time
    print "elapsed centiseconds =", elapsed


spark_handler()
```

Notes:
- When in Spark, lambda_handler function is optional (won't cause any problems if present). Also, don't forget to adjust bucket name and region in save_rdds function.
- When in Lambda, spark_handler and save_rdds functions are optional (won't cause any problems if present); however, the very last line (call to spark_handler) **must be removed**. Lambda doesn't seem to tolerate any code which is not encapsulated into a function of some sort.

APPENDIX A - Monitoring the Cluster

This section describes ways to monitor the clusters performance.

Dynamic SSH tunneling

First, to also connect to Ganglia through an SSH tunnel you need to access the Spark cluster via more than one port. This section presents an approach that uses “dynamic” ssh tunneling and a “Proxy” (specifically, FoxyProxy extension).

On your browser’s computer start a “dynamic” ssh tunnel with this command:

```
$ EMR_Master=ec2-54-204-218-87.compute-1.amazonaws.com
$ Tunnel_Port=8157
$ ssh -i ~/GM-EMR.pem -ND $EMR_Port hadoop@${EMR_Master}
```

Then,
install the following file into FoxyProxy

```
<?xml version="1.0" encoding="UTF-8"?>
<foxyproxy>
    <proxies>
        <proxy name="emr-socks-proxy" id="2322596116" notes="" fromSubscription="false" enabled="true" mode="manual" selectedTabIndex="2"
lastresort="false" animatedIcons="true" includeInCycle="true" color="#0055E5" proxyDNS="true" noInternalIPs="false" autoconfMode="pac"
clearCacheBeforeUse="false" disableCache="false" clearCookiesBeforeUse="false" rejectCookies="false">
            <matches>
                <match enabled="true" name="*ec2*.amazonaws.com*" pattern="*ec2*.amazonaws.com*" isRegEx="false" isBlackList="false"
isMultiLine="false" caseSensitive="false" fromSubscription="false" />
                <match enabled="true" name="*ec2*.compute*" pattern="*ec2*.compute*" isRegEx="false" isBlackList="false"
isMultiLine="false" caseSensitive="false" fromSubscription="false" />
                <match enabled="true" name="10.*" pattern="http://10.*" isRegEx="false" isBlackList="false" isMultiLine="false"
caseSensitive="false" fromSubscription="false" />
                <match enabled="true" name="*10*.amazonaws.com*" pattern="*10*.amazonaws.com*" isRegEx="false" isBlackList="false"
isMultiLine="false" caseSensitive="false" fromSubscription="false" />
                <match enabled="true" name="*10*.compute*" pattern="*10*.compute*" isRegEx="false" isBlackList="false" isMultiLine="false"
caseSensitive="false" fromSubscription="false" />
                <match enabled="true" name="*.compute.internal*" pattern="*.compute.internal*" isRegEx="false" isBlackList="false"
isMultiLine="false" caseSensitive="false" fromSubscription="false" />
                <match enabled="true" name="*.ec2.internal*" pattern="*.ec2.internal*" isRegEx="false" isBlackList="false"
isMultiLine="false" caseSensitive="false" fromSubscription="false" />
            </matches>

            <manualconf host="localhost" port="8157" socksversion="5" isSocks="true" username="" password="" domain="" />
        </proxy>
    </proxies>
</foxyproxy>
```

1. Click the FoxyProxy icon in the toolbar and select Options.
2. Click File > Import Settings.
3. Browse to foxyproxy-settings.xml, select it, and click Open.
4. To confirm that you wish to overwrite current settings with those stored in the new file, click Yes.
5. Click Yes to restart Firefox.
6. From the Firefox menu, choose Add-ons.
7. Beside the FoxyProxy Standard add-on, click Options.
8. In the FoxyProxy Standard dialog, for Select Mode, choose Use proxies based on their predefined patterns and priorities.
9. Click Close.

Now connect to iPython/Jupyter via the master's public DNS

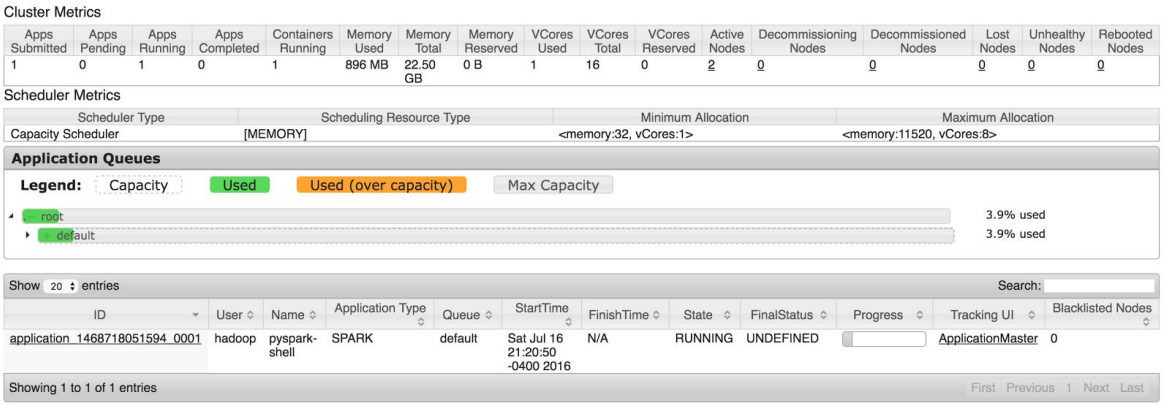
http://ec2-54-204-218-87.compute-1.amazonaws.com:8192/tree

And to Ganglia

http://ec2-54-204-218-87.compute-1.amazonaws.com/ganglia/

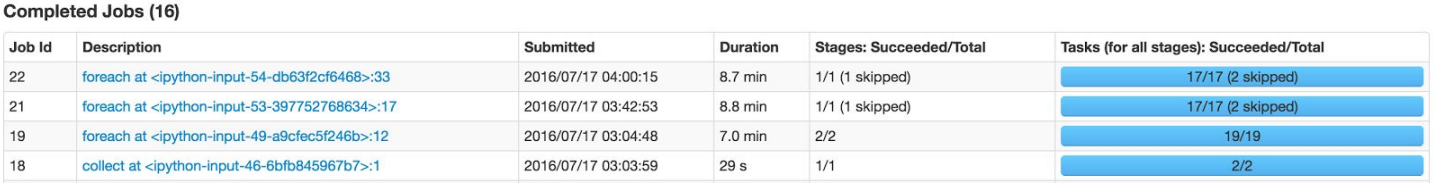
Spark Scheduler

3) A very useful monitoring console (requires setting up foxy proxy as per AWS instructions):
http://ec2-54-163-XXX.XXX.compute-1.amazonaws.com:8088/cluster/scheduler



At the bottom, where it shows "application_142……._0001", State: RUNNING - that's the pyspark-shell launched by Notebook. In that same line, further to the right there is a link to "Application Master" - open it in another window.

The link from application master will take you to the screen showing all jobs executed from the Notebook. At first, the screen will be empty. As you click through the steps in the Notebook, keep refreshing this screen. You'll notice first short job will appear on Step 3, as we collecting keys. It'll run for about 20 secs or less (with just a dozen of files, might be longer if more). Each job will first appear in "Active" category and later move to either "Completed" or "Failed"



For example, on the screen above, job 18 was the "collect keys" step (Step 2). As you notice, steps 3, 4 and 5 didn't kick off any jobs at all. It wasn't until Step 6 (foreach) that another job was dispatched (which included 3,4,5 and 6). I re-executed Step 6 three times (making changes to the code), hence it shows up 3 times.

This was a two m3.xlarge workers cluster and converting 17 files took less than 9 minutes (as opposed to yesterday's version, where I was running a cluster of 3 m3.xlarge workers and it was taking same or slightly even more time).

Clicking on the job will show Stages:

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
35	foreach at <ipython-input-49-a9cfec5f246b>:12 +details	2016/07/17 03:04:48	7.0 min	17/17			423.4 KB	
34	repartition at NativeMethodAccessorImpl.java:-2 +details	2016/07/17 03:04:48	0.2 s	2/2				423.5 KB

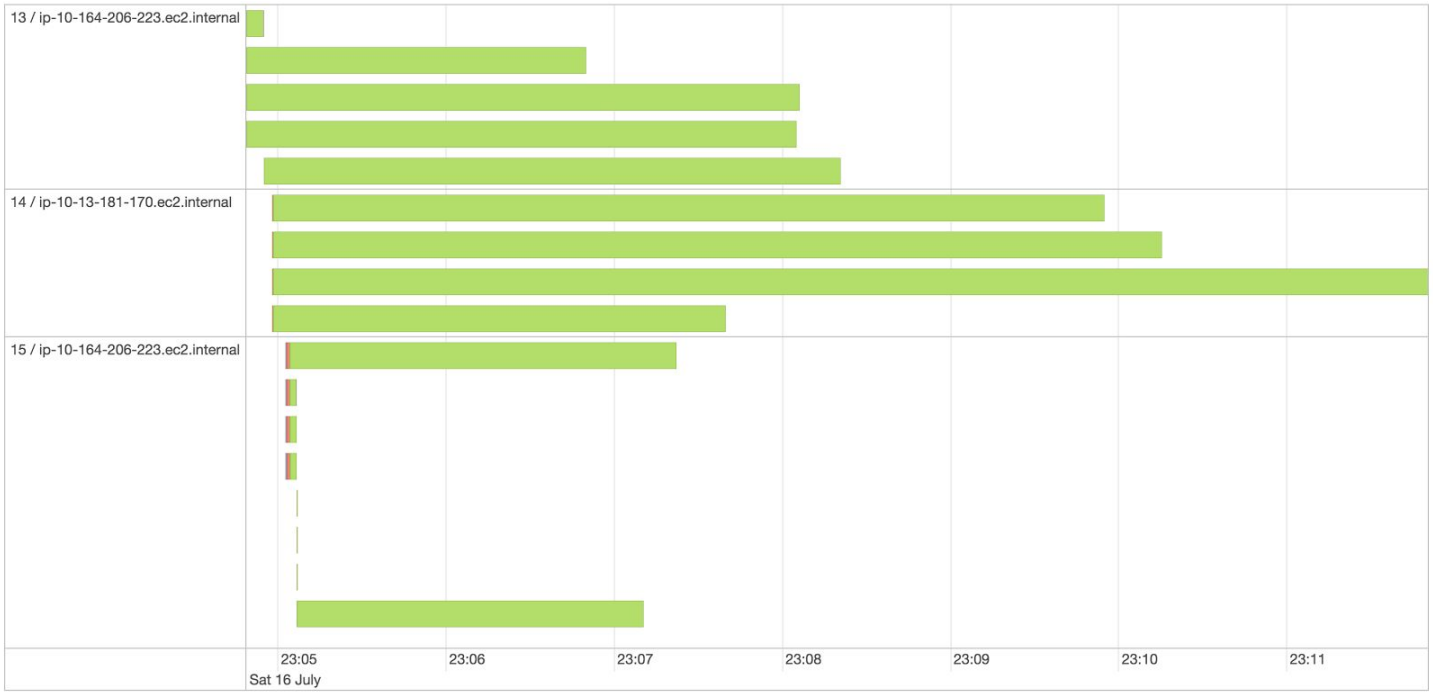
This shows how re-partitioning took place (was quick, no data was really moved, seems like only meta-data of some sort was updated). Then the foreach step. Clicking on it will show lots of interesting information. First Summary, then, "Aggregated metric by executor"

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records
13	<IP1>	12 min	5	0	5	199.1 KB / 8
14	<IP2>	20 min	4	0	4	174.2 KB / 7
15	<IP1>	4.6 min	8	0	8	50.1 KB / 2

It shows how there were 3 executors, 13 and 15 were running on the same worker node (same IP), executors 13 and 15 converted a total of 13 files (out of 17) and the executor 14 did only 4.

Expanding “Event Timeline” at the tops shows this graph:



Here we can see how each executor gets 3-4 tasks and then as soon as it finishes one, it gets next one, etc. Looks like executor 15 was getting alot of small files (or empties), because it was processing them very quickly, getting assigned more work.. That’s why 13 and 15 together did so many more tasks on the same node, comparing to 14, which was running on a node of its own.. But now if you look on the number of “heavy” tasks processed by each worker node (server), they come to about the same.

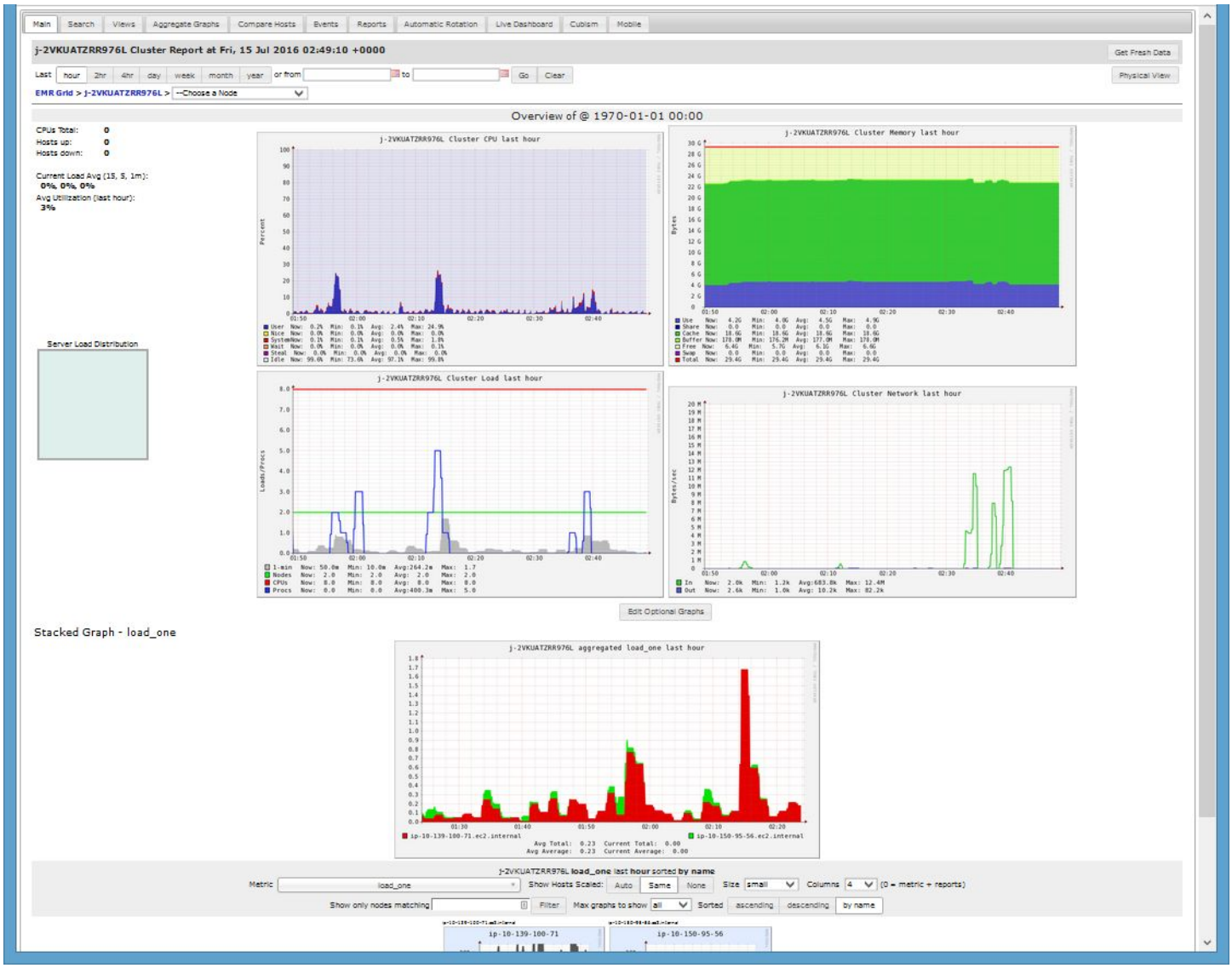
Finally there are details for all 17 tasks, with lots of interesting information:

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	Scheduler Delay	Task Deserialization Time	GC Time	Result Serialization Time	Getting Result Time	Peak Execution Memory	Shuffle Read Blocked Time	Shuffle Read Size / Records	Shuffle Remote Reads	Errors
0	78	0	SUCCESS	NODE_LOCAL	13 / ip-10-164-206-223.ec2.internal	2016/07/17 03:04:48	6 s	15 ms	36 ms		1 ms	0 ms	0.0 B	0 ms	24.9 KB / 1	0.0 B	
1	79	0	SUCCESS	NODE_LOCAL	13 / ip-10-164-206-223.ec2.internal	2016/07/17 03:04:48	2.0 min	11 ms	35 ms	65 ms	0 ms	0 ms	0.0 B	0 ms	24.9 KB / 1	0.0 B	
2	80	0	SUCCESS	NODE_LOCAL	13 / ip-10-164-206-223.ec2.internal	2016/07/17 03:04:48	3.3 min	12 ms	33 ms	65 ms	0 ms	0 ms	0.0 B	0 ms	49.8 KB / 2	0.0 B	
3	81	0	SUCCESS	NODE_LOCAL	13 / ip-10-164-206-223.ec2.internal	2016/07/17 03:04:48	3.3 min	14 ms	32 ms	65 ms	0 ms	0 ms	0.0 B	0 ms	49.8 KB / 2	0.0 B	
4	82	0	SUCCESS	NODE_LOCAL	13 / ip-10-164-206-	2016/07/17 03:04:54	3.4 min	10 ms	26 ms	65 ms	0 ms	0 ms	0.0 B	0 ms	49.8 KB / 2	0.0 B	

4) Final thought - this particular cluster could probably use more executors. Unfortunately this is something that’s controlled during spark context creation, which means either during starting notebook (pyspark shell) or programmatically, when job is submitted via command line, but not possible from inside notebook because notebook gets passed pre-created spark context.

Performance Analysis/Reporting with Ganglia



APPENDIX B - pyspark jobs via CLI command line

To submit work to Spark using the AWS CLI

http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-spark-submit-step.html

To execute pyspark job via command line:

1) SSH to EMR master node
2) By default you will find yourself in hadoop's home directory (/home/hadoop)
3) Place (via copy or vi+cut/paste) your python job here
4) If transferring from pyspark notebook, simply assemble all steps into the same file and add the following two lines at the beginning:

```
from pyspark import SparkContext
sc = SparkContext(appName="my_pyspark_job")
```

5) Assuming we placed the script to hadoop's home and called it "my_pyspark_job.py", the following command will submit this job to Spark cluster (the command will execute the job and duplicate its output to /tmp/out.txt in addition to console):

```
cd /usr/lib/spark
bin/spark-submit ~/my_pyspark_job.py --master yarn --deploy-mode client 2>&1 | tee /tmp/out.txt
```

6) Additional parameters that can be provided:
   `--num-executors <N>`- how many executors to dedicate to the job
   `--executor-cores <N>`- how many threads to run per each executor
   `--executor-memory <N>Gb`- how much memory to allocate for each executor

   Each worker node can run one or more executors.
   Each executor can run one or more "cores" or "tasks" or sometimes they're also referred as "threads". I believe it's best to give each thread its own CPU core.
   Memory on each worker node needs to be divided among all executors with small percentage to be reserved for OS.

   Example:
   If we have a cluster with 3 m3.xlarge worker nodes. Each node has 4 virtual cores. Let's give each executor it's own core. We could either allow 1 executor per node with 4 cores or 2 executors per node and restrict each to 2 cores. Let' try the latter. In case of 3 nodes, 2 executors each we're looking at
   `--num-executors 6 --executor-cores 2`

   Now let's sort out memory. Each node comes with 15Gb of memory. With two executor per node, we will divide 15Gb among two executor and leave some for the OS - 6Gb x 2 executors and 3Gb for OS:
   `--num-executors 6 --executor-cores 2 --executor-memory 6Gb`

In my experiments, however, I noticed that there is no need for this type of manual control. Spark and Yarn have been consistently allocating proper number of executors/cores to parallelize al 17 tasks very efficiently (when talking about running pdf converter/NLP taks against 17 PDFs)

7) Since we saved the output to /tmp/out.txt, we can now analyze execution in more details

Analysis of job execution:

```
Vadims-MBP:~ vadim$ egrep "Starting task|Finished" out10 | grep "stage 2"
16/07/20 04:14:53 INFO TaskSetManager: Starting task 0.0 in stage 2.0 (TID 4, ip-10-168-213-45.ec2.internal, partition 0,NODE_LOCAL, 2163 bytes)
16/07/20 04:14:53 INFO TaskSetManager: Starting task 1.0 in stage 2.0 (TID 5, ip-10-168-213-45.ec2.internal, partition 1,NODE_LOCAL, 2163 bytes)
16/07/20 04:14:53 INFO TaskSetManager: Starting task 2.0 in stage 2.0 (TID 6, ip-10-168-213-45.ec2.internal, partition 2,NODE_LOCAL, 2163 bytes)
16/07/20 04:14:53 INFO TaskSetManager: Starting task 3.0 in stage 2.0 (TID 7, ip-10-168-213-45.ec2.internal, partition 3,NODE_LOCAL, 2163 bytes)
16/07/20 04:14:58 INFO TaskSetManager: Starting task 4.0 in stage 2.0 (TID 8, ip-10-237-238-231.ec2.internal, partition 4,RACK_LOCAL, 2163 bytes)
16/07/20 04:14:58 INFO TaskSetManager: Starting task 5.0 in stage 2.0 (TID 9, ip-10-237-238-231.ec2.internal, partition 5,RACK_LOCAL, 2163 bytes)
16/07/20 04:14:58 INFO TaskSetManager: Starting task 6.0 in stage 2.0 (TID 10, ip-10-237-238-231.ec2.internal, partition 6,RACK_LOCAL, 2163 bytes)
16/07/20 04:14:58 INFO TaskSetManager: Starting task 7.0 in stage 2.0 (TID 11, ip-10-237-238-231.ec2.internal, partition 7,RACK_LOCAL, 2163 bytes)
16/07/20 04:14:59 INFO TaskSetManager: Starting task 8.0 in stage 2.0 (TID 12, ip-10-168-213-45.ec2.internal, partition 8,NODE_LOCAL, 2163 bytes)
16/07/20 04:14:59 INFO TaskSetManager: Finished task 0.0 in stage 2.0 (TID 4) in 6428 ms on ip-10-168-213-45.ec2.internal (1/17)
16/07/20 04:15:03 INFO TaskSetManager: Starting task 9.0 in stage 2.0 (TID 13, ip-10-168-185-210.ec2.internal, partition 9,RACK_LOCAL, 2163 bytes)
16/07/20 04:15:03 INFO TaskSetManager: Starting task 10.0 in stage 2.0 (TID 14, ip-10-168-185-210.ec2.internal, partition 10,RACK_LOCAL, 2163 bytes)
16/07/20 04:15:03 INFO TaskSetManager: Starting task 11.0 in stage 2.0 (TID 15, ip-10-168-185-210.ec2.internal, partition 11,RACK_LOCAL, 2163 bytes)
16/07/20 04:15:03 INFO TaskSetManager: Starting task 12.0 in stage 2.0 (TID 16, ip-10-168-185-210.ec2.internal, partition 12,RACK_LOCAL, 2163 bytes)
16/07/20 04:15:04 INFO TaskSetManager: Starting task 13.0 in stage 2.0 (TID 17, ip-10-168-185-210.ec2.internal, partition 13,RACK_LOCAL, 2163 bytes)
16/07/20 04:15:04 INFO TaskSetManager: Starting task 14.0 in stage 2.0 (TID 18, ip-10-168-185-210.ec2.internal, partition 14,RACK_LOCAL, 2163 bytes)
16/07/20 04:15:04 INFO TaskSetManager: Starting task 15.0 in stage 2.0 (TID 19, ip-10-168-185-210.ec2.internal, partition 15,RACK_LOCAL, 2163 bytes)
16/07/20 04:15:04 INFO TaskSetManager: Starting task 16.0 in stage 2.0 (TID 20, ip-10-168-185-210.ec2.internal, partition 16,RACK_LOCAL, 2163 bytes)
16/07/20 04:15:06 INFO TaskSetManager: Finished task 12.0 in stage 2.0 (TID 16) in 2330 ms on ip-10-168-185-210.ec2.internal (2/17)
16/07/20 04:15:06 INFO TaskSetManager: Finished task 10.0 in stage 2.0 (TID 14) in 2406 ms on ip-10-168-185-210.ec2.internal (3/17)
16/07/20 04:15:06 INFO TaskSetManager: Finished task 11.0 in stage 2.0 (TID 15) in 2454 ms on ip-10-168-185-210.ec2.internal (4/17)
16/07/20 04:15:07 INFO TaskSetManager: Finished task 15.0 in stage 2.0 (TID 19) in 2738 ms on ip-10-168-185-210.ec2.internal (5/17)
16/07/20 04:15:07 INFO TaskSetManager: Finished task 13.0 in stage 2.0 (TID 17) in 2784 ms on ip-10-168-185-210.ec2.internal (6/17)
16/07/20 04:15:07 INFO TaskSetManager: Finished task 14.0 in stage 2.0 (TID 18) in 2791 ms on ip-10-168-185-210.ec2.internal (7/17)
16/07/20 04:16:13 INFO TaskSetManager: Finished task 16.0 in stage 2.0 (TID 20) in 68402 ms on ip-10-168-185-210.ec2.internal (8/17)
16/07/20 04:16:13 INFO TaskSetManager: Finished task 1.0 in stage 2.0 (TID 5) in 80170 ms on ip-10-168-213-45.ec2.internal (9/17)
16/07/20 04:16:19 INFO TaskSetManager: Finished task 9.0 in stage 2.0 (TID 13) in 75341 ms on ip-10-168-185-210.ec2.internal (10/17)
16/07/20 04:16:54 INFO TaskSetManager: Finished task 2.0 in stage 2.0 (TID 6) in 121360 ms on ip-10-168-213-45.ec2.internal (11/17)
16/07/20 04:17:30 INFO TaskSetManager: Finished task 3.0 in stage 2.0 (TID 7) in 157517 ms on ip-10-168-213-45.ec2.internal (12/17)
16/07/20 04:17:35 INFO TaskSetManager: Finished task 8.0 in stage 2.0 (TID 12) in 155606 ms on ip-10-168-213-45.ec2.internal (13/17)
16/07/20 04:18:34 INFO TaskSetManager: Finished task 4.0 in stage 2.0 (TID 8) in 215693 ms on ip-10-237-238-231.ec2.internal (14/17)
16/07/20 04:19:52 INFO TaskSetManager: Finished task 6.0 in stage 2.0 (TID 10) in 294032 ms on ip-10-237-238-231.ec2.internal (15/17)
16/07/20 04:20:37 INFO TaskSetManager: Finished task 5.0 in stage 2.0 (TID 9) in 338784 ms on ip-10-237-238-231.ec2.internal (16/17)
16/07/20 04:22:20 INFO TaskSetManager: Finished task 7.0 in stage 2.0 (TID 11) in 441585 ms on ip-10-237-238-231.ec2.internal (17/17)
Vadims-MBP:~ vadim$
```

- we're grepping through the output file for "Starting task" and "Finished"

- we're restricting to stage 2, because stage one was simple "collect" to figure out how many partitions we needed, but stage 2 is the actual pdf conversion/NLP

- we can see that all tasks were started in parallel, there was no wait for first batch of tasks to complete before starting second batch of tasks. For example, if we only had 1 executor and 2 threads, we would see first two tasks starting, 15 waiting in the queue. As first two tasks completed, next 2 tasks would start, etc. Here, we're clearly observing massive parallel processing of all 17 tasks

- we can also see which worker nodes are involved in processing, and it appears that all three are more or less utilized

- since some PDFs are bigger and some smaller, Yarn has no knowledge of that and it's possible for lots of smaller tasks to end up on one node, while the other two nodes will have to work harder on bigger PDFs. This skew will be less obvious in more massive tasks (larger amounts of PDFs)

- we can observer start/stop time of each task, and Finish line has also number of milliseconds it took for the task to complete

- it's clear that task 7 was responsible for length of the entire job, meaning, it was started pretty much right away and finished last - must have been working on the largest PDF file out of all 17

Would adding one more m3.xlarge into the mix increase performance of this job by 1/3rd ? Probably not, because there were no tasks waiting for a slot to run - all were already running in parallel. We could spread all 17 tasks amount 4 worker nodes as opposed to just 3 nodes, and that would probably make processing of each task go a little faster (simply because each node would be splitting compute power between lesser number of tasks), but that wouldn't produce significant performance increase.

A different story would be if we had 100 tasks to run on this cluster - they would most definitely NOT get scheduled all at once and as a result, there would be a queue of tasks. Some would get only scheduled after previous tasks got completed. Adding an extra node in that case would definitely have a significant impact.

This shows shows how executors were being allocated during this job's run:

```
Vadims-MBP:~ vadim$ grep -i "New executor" out10
16/07/20 04:14:28 INFO ExecutorAllocationManager: Requesting 1 new executor because tasks are backlogged (new desired total will be 1)
16/07/20 04:14:34 INFO ExecutorAllocationManager: New executor 1 has registered (new total is 1)
16/07/20 04:14:54 INFO ExecutorAllocationManager: Requesting 2 new executors because tasks are backlogged (new desired total will be 2)
16/07/20 04:14:55 INFO ExecutorAllocationManager: Requesting 1 new executor because tasks are backlogged (new desired total will be 3)
16/07/20 04:14:56 INFO ExecutorAllocationManager: Requesting 2 new executors because tasks are backlogged (new desired total will be 5)
16/07/20 04:14:58 INFO ExecutorAllocationManager: New executor 2 has registered (new total is 2)
16/07/20 04:15:03 INFO ExecutorAllocationManager: New executor 4 has registered (new total is 3)
16/07/20 04:15:04 INFO ExecutorAllocationManager: New executor 5 has registered (new total is 4)
16/07/20 04:15:05 INFO ExecutorAllocationManager: New executor 3 has registered (new total is 5)
```
