Running Spark on a Cluster: The Basics

PART 1: START A SPARK CLUSTER AND USE THE SPARK-SHELL

In this short guide, we’ll walk through how to run modest Spark jobs on a cluster. We’ll do this in 2 parts:

* **Part 1:** (this guide) We’ll cover how to start up a Spark cluster using the Flintrock command-line tool, and we’ll run a simple word count example using the spark-shell, Spark’s interactive REPL.
* **Part 2:** (the [next guide](https://heather.miller.am/blog/launching-a-spark-cluster-part-2.html)) We’ll connect our Spark job to an S3 bucket, add a simple library dependency, and we’ll develop the job in a normal IDE, using spark-submit to submit our job and its dependencies to the cluster, without invoking the spark-shell.

We’ll begin with the simple task of spinning up a Spark cluster on Amazon EC2, and running a very simple Spark program from the spark-shell (Spark’s REPL).

To do this, we’ll need [**Flintrock**](https://github.com/nchammas/flintrock), a Python-based command-line tool designed to quickly and easily spin up Spark clusters on Amazon Web Services (AWS).

Setting Up Flintrock and Amazon Web Services

Flintrock is a command-line tool that makes it easy to start/pause/tear down Spark clusters, and to otherwise seamlessly interact with your cluster from the command line.

**INSTALLING FLINTROCK**

At the time of writing, the most up-to-date version of Flintrock is 0.9.0.

To get the latest release of Flintrock, simply install using [pip](https://pip.pypa.io/en/stable/):

$ pip3 install flintrock

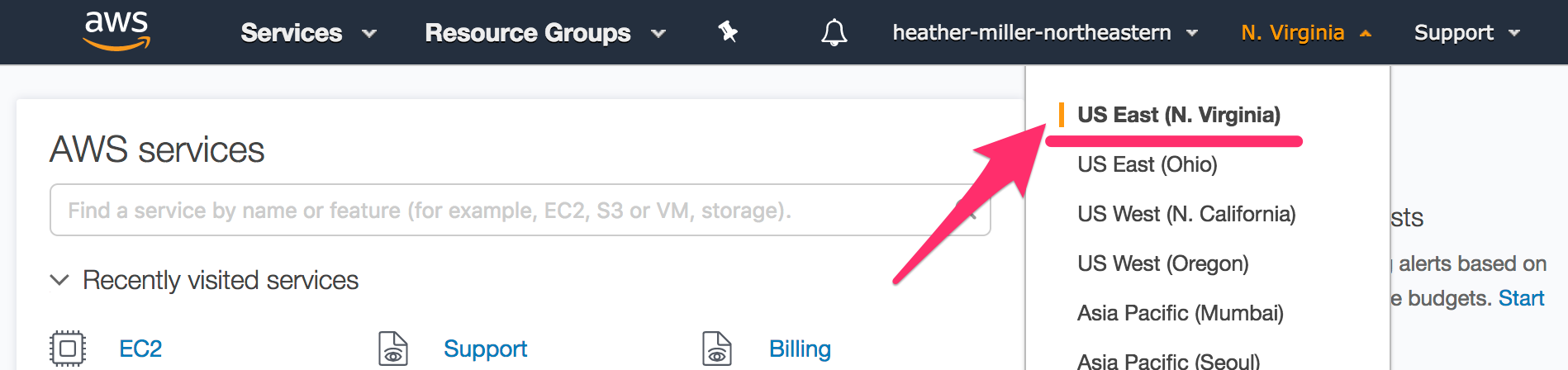
If you would like to use an option other than pip to get Flintrock, see [Flintrock’s installation instructions.](https://github.com/nchammas/flintrock#installation)

**CREATING A NEW AWS USER FOR FLINTROCK**

I assume at this point that you’ve created an [AWS account](https://aws.amazon.com/).

**Take note of the availability zone that your account is set for (top right corner of your AWS console). You will need to ensure that all of your Flintrock configuration is for the *same* availability zone**

In this guide, I am using us-east-1:



We must now create a user that we can use with Flintrock to start and manage a Spark cluster. For the sake of simplicity and reproducability, we’re going to create an almost-root user, which has the rights to do most anything except AWS account administration such as changing billing details.

*(In a production scenario, you should obviously further restrict access to individual users.)*

To create a new user, simply:

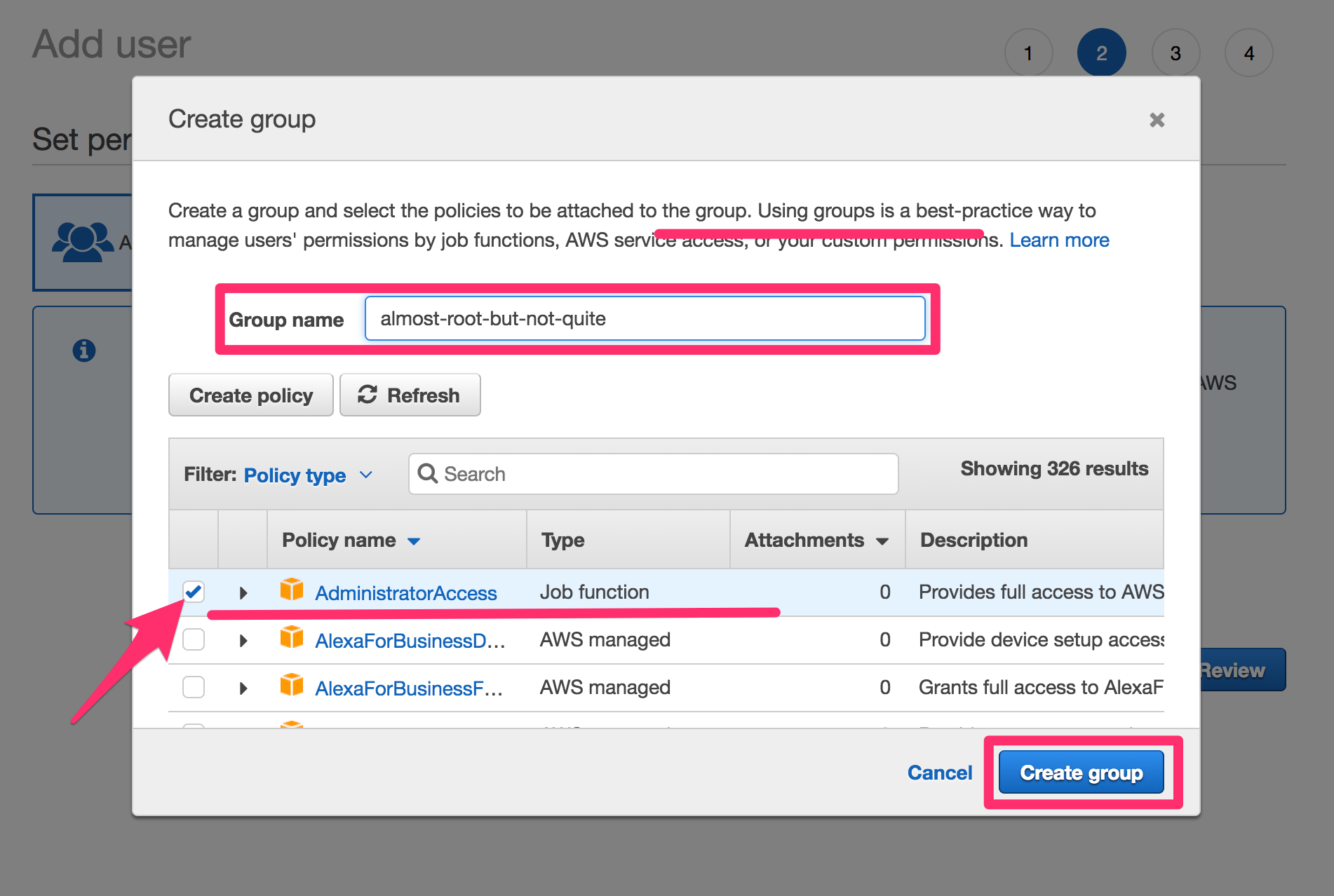
1. Go to the AWS console, and in the dropdown or “all services” list, select **IAM**(which stands for Identity and Access Management).
2. On the left panel, select **Users**.
3. Click the blue **Add user** button.

On the page that appears (add user step 1):

1. Provide a username (I chose sparkuser)
2. Check the box next to **both** access types; “Programmatic access” as well as “AWS Management Console access”
3. Click the **Next: Permissions** button.

On the Permissions page (add user step 2):

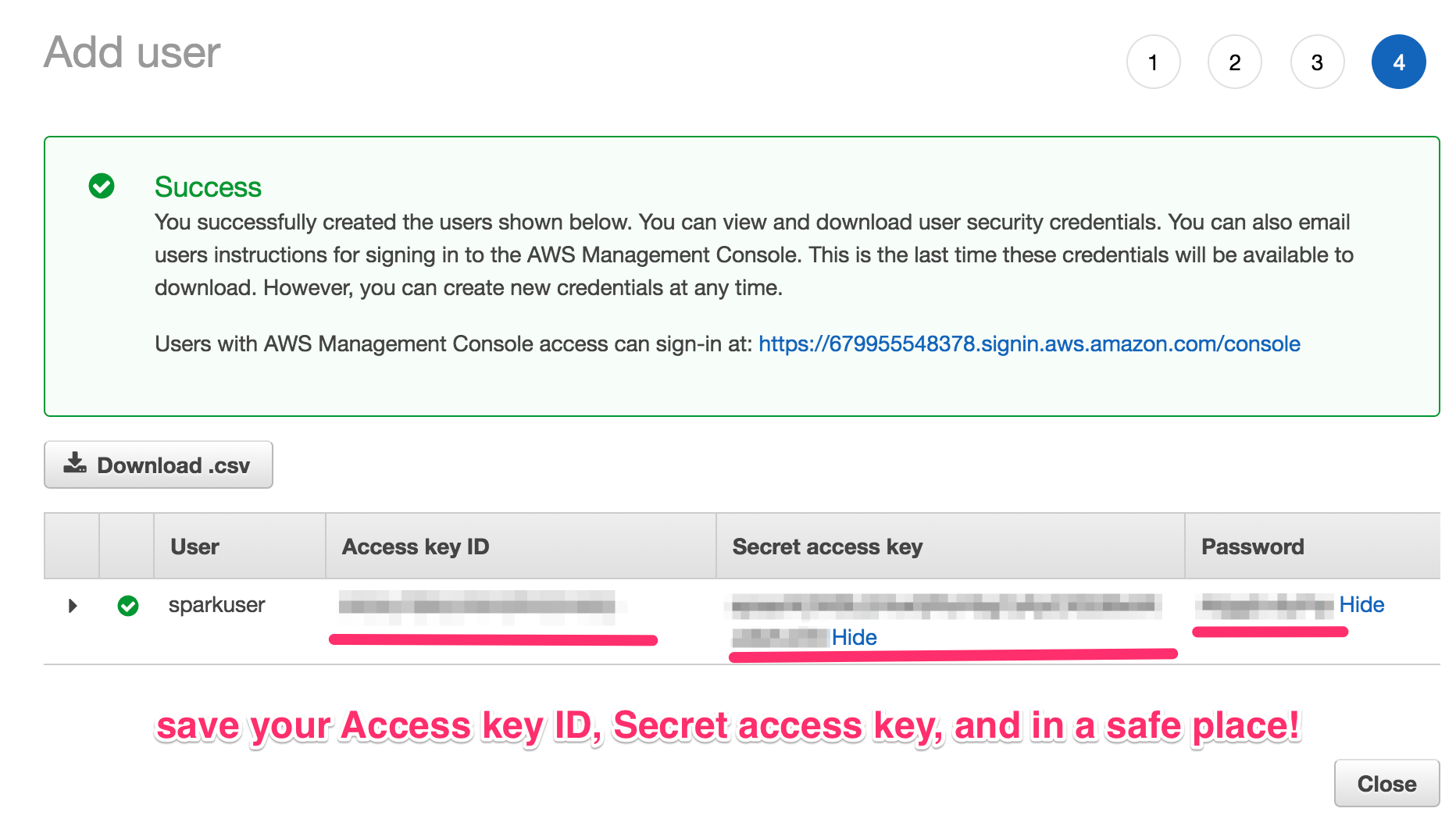
Select “Add user to group” and click “Create group”. Create a group with the following policy selected:



Add your user to this newly created permissions group, and click the **Next: Review**button.

Verify that the user details and permissions summary look correct, and click **Create user**.

**Important next step: save the key ID, secret access key, and password associated with your newly-created user!** Either download as a CSV file, or copy/paste the Access Key ID, Secret Access Key, and Password to a safe place.



In order for Flintrock to use your newly-created user’s Access Key ID and Secret Access Key, you’ll need to set the following environment variables so Flintrock can use this user later.

$ export AWS\_ACCESS\_KEY\_ID=xxxxxxxxxxxxxxxxxxxx

$ export AWS\_SECRET\_ACCESS\_KEY=yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy

**GENERATING EC2 KEY PAIRS**

In order for you to remotely access your EC2 instances, you will need to generate a key pair in order to authenticate.

To generate a key pair for you to use to authenticate, simply:

1. Go to the AWS console, and in the dropdown or “all services” list, select **EC2**.
2. On the left panel, under “Network & Security,” select **Key Pairs**.
3. Click the blue **Create Key Pair** button.
4. Name your key pair spark\_cluster, and click **Create**.
5. This will start an automatic download of a file called spark\_cluster.pem.
6. Change the permissions of the .pem file so that only the owner can read it:

$ chmod 400 spark\_cluster.pem

**CONFIGURING FLINTROCK WITH AWS**

Now, all we have to do is provide our desired cluster configuration to Flintrock before we can start up our Spark cluster.

To do this, at the command line, just do:

flintrock configure

This will open up your system’s configured text editor with a default YAML configuration file, called config.yaml.

For my job, I have set the following options:

services:

spark:

version: 2.2.0

hdfs:

version: 2.7.5

provider: ec2

providers:

ec2:

key-name: spark\_cluster

identity-file: /path/to/spark\_cluster.pem

instance-type: m4.large

region: us-east-1

ami: ami-97785bed *# Amazon Linux, us-east-1*

user: ec2-user

tenancy: default *# default | dedicated*

ebs-optimized: no *# yes | no*

instance-initiated-shutdown-behavior: terminate *# terminate | stop*

launch:

num-slaves: 1

install-hdfs: True

install-spark: True

debug: false

To explain this selection of options:

* key-name and identity-file are simply the .pem file that we generated above.
* instance-type is the EC2 instance type you’d like to run. The full pricing list for all instance types per region is available [here](https://aws.amazon.com/ec2/pricing/).
* region is the AWS region that we’ve chosen in our account. In my case, I’m using us-east-1.
* ami is the Amazon Machine Image (AMI) that we want to use in our cluster. For my region, us-east-1, I’ve simply chosen the latest Linux AMI for my region, which happens to be ami-97785bed at the time of writing.
* user is the username of the non-root user that’s created on each VM
* num-slaves is the number of non-master Spark nodes in the cluster. So in selecting 1 slave, that means I’ll have 2 nodes in total; one master, and one worker/slave.
* install-hdfs should be set to true if you want to access data in S3.

Starting a Spark Cluster with Flintrock

We now have everything we need setup to spin up a Spark cluster!

To start our cluster, we just have to choose a name for our cluster, e.g., my-spark-cluster and do:

$ flintrock launch my-spark-cluster

It should take ~2 minutes to complete. When finished, you should see the following:

$ flintrock launch my-spark-cluster

Launching 2 instances...

[107.21.87.63] SSH online.

[107.21.87.63] Configuring ephemeral storage...

[107.21.87.63] Installing Java 1.8...

[34.234.87.162] SSH online.

[34.234.87.162] Configuring ephemeral storage...

[34.234.87.162] Installing Java 1.8...

[34.234.87.162] Installing HDFS...

[107.21.87.63] Installing HDFS...

[107.21.87.63] Installing Spark...

[34.234.87.162] Installing Spark...

[107.21.87.63] Configuring HDFS master...

[107.21.87.63] Configuring Spark master...

HDFS online.

Spark online.

launch finished in 0:02:53.

Cluster master: ec2-107-21-87-63.compute-1.amazonaws.com

Login with: flintrock login my-spark-cluster

*Note: if you get an error saying that Flintrock could not find your AWS credentials, remember to set your AWS\_ACCESS\_KEY\_ID and AWS\_SECRET\_ACCESS\_KEYenvironment variables as specified above.*

Now we can login to the master node of our cluster! To do that, just do:

$ flintrock login my-spark-cluster

And you should see something to the effect of:

\_\_| \_\_|\_ )

\_| ( / Amazon Linux AMI

\_\_\_|\\_\_\_|\_\_\_|

https://aws.amazon.com/amazon-linux-ami/2017.09-release-notes/

[ec2-user@ip]$

Running a Simple Spark Job From the spark-shell

On the command line of the master node of our cluster (i.e., in the EC2 shell shown above), to launch the spark-shell, just do:

$ spark-shell

After some start-up output, you should eventually see:

Spark context Web UI available at http://ec2-xxxxxxxxxx.compute-1.amazonaws.com:4040

Spark context available as 'sc' (master = local[\*], app id = local-1522613551031).

Spark session available as 'spark'.

Welcome to

\_\_\_\_ \_\_

/ \_\_/\_\_ \_\_\_ \_\_\_\_\_/ /\_\_

\_\ \/ \_ \/ \_ `/ \_\_/ '\_/

/\_\_\_/ .\_\_/\\_,\_/\_/ /\_/\\_\ version 2.2.0

/\_/

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0\_161)

Type in expressions to have them evaluated.

Type :help for more information.

scala>

To run a simple word count example on our Spark cluster, just enter:

val text = List("Hadoop MapReduce, a disk-based big data processing engine, is being replaced by a new generation of memory-based processing frameworks, the most popular of which is Spark.", "Spark supports Scala, Java, Python, and R.")

val rdd = sc.parallelize(text)

val counts = rdd.flatMap(line => line.split(" "))

.map(word => (word, 1))

.reduceByKey(\_ + \_)

counts.collect()

The result should look like:

scala> counts.collect()

res0: Array[(String, Int)] = Array((is,2), (Java,,1), (big,1), (data,1), (new,1), (generation,1), (engine,,1), (Scala,,1), (MapReduce,,1), (Python,,1), (R.,1), (supports,1), (replaced,1), (Spark,1), (frameworks,,1), (memory-based,1), (most,1), (a,2), (popular,1), (disk-based,1), (processing,2), (Spark.,1), (which,1), (of,2), (by,1), (and,1), (being,1), (the,1), (Hadoop,1))

And voilà! You’ve just run a simple Spark job on your own Spark cluster!

To shut down your cluster, just run:

$ flintrock destroy my-spark-cluster

# Running Spark on a Cluster: The Basics

PART 2: DEPENDENCIES, S3, AND DEPLOYING VIA SPARK-SUBMIT

In this short guide, we’ll walk through how to run modest Spark jobs on a cluster. We’ll do this in 2 parts:

* **Part 1:** (the [previous guide](https://heather.miller.am/blog/launching-a-spark-cluster-part-1.html)) We’ll cover how to start up a Spark cluster using the Flintrock command-line tool, and we’ll run a simple word count example using the spark-shell, Spark’s interactive REPL.
* **Part 2:** (this guide) We’ll connect our Spark job to an S3 bucket, add a simple library dependency, and we’ll develop the job in a normal IDE, using spark-submit to submit our job and its dependencies to the cluster, without invoking the spark-shell.

In this guide, we assume you’re developing a Spark application in your regular development environment and that when ready, you’d simply like to submit that job to be run on a Spark cluster running on EC2.

To do this, we need a variety of tools:

1. **Spark pre-built for Apache Hadoop**. Spark comes with a script called spark-submit which we will be using to submit our job to the cluster.
2. **Scala and sbt**.
3. **Flintrock**. A command-line tool for launching Spark clusters. See [part 1](https://heather.miller.am/blog/launching-a-spark-cluster-part-1.html) for this.
4. **A running cluster started with Flintrock**. See [part 1](https://heather.miller.am/blog/launching-a-spark-cluster-part-1.html) for this.

Given these tools, our general workflow is as follows:

1. Develop locally.
2. When ready to deploy, compile, package up jars to send to cluster.
3. Copy jars to master and worker nodes using Flintrock.
4. Use spark-submit script to start job.
5. Check Spark web UI to see output (stdout/stderr).

## Downloading Spark

Head over to <https://spark.apache.org/downloads.html>. (At the time of writing, Spark 2.2.0 will offer the smoothest set-up experience.)

Simply select the 2.2.0 release of Spark, ensure that the “pre-built for Apache Hadoop 2.7 and later” option is selected, and simply download [Spark 2.2.0, pre-built for Apache Hadoop 2.7 and later](https://www.apache.org/dyn/closer.lua/spark/spark-2.2.0/spark-2.2.0-bin-hadoop2.7.tgz).

Reasons to use Flintrock over AWS’s Elastic Map Reduce (EMR); with Flintrock, it’s easier to pause a running cluster or do a number of Spark-specific cluster tasks directly from the command-line.

Once downloaded, unzip, and cd into the spark-2.2.0-bin-hadoop2.7 directory. To ensure everything works as expected, simply do:

$ ./bin/spark-submit --help

And you should see the following:

$ ./bin/spark-submit --help

Usage: spark-submit [options] <app jar | python file> [app arguments]

Usage: spark-submit --kill [submission ID] --master [spark://...]

Usage: spark-submit --status [submission ID] --master [spark://...]

Usage: spark-submit run-example [options] example-class [example args]

Options:

--master MASTER\_URL spark://host:port, mesos://host:port, yarn, or local.

--deploy-mode DEPLOY\_MODE Whether to launch the driver program locally ("client") or

on one of the worker machines inside the cluster ("cluster")

(Default: client).

--class CLASS\_NAME Your applications main class (for Java / Scala apps).

--name NAME A name of your application.

--jars JARS Comma-separated list of local jars to include on the driver

and executor classpaths.

## Simple Spark Project

We assume you will be developing in [Scala](http://scala-lang.org/) with [sbt](https://www.scala-sbt.org/). To keep things simple and as reproducible as possiible, I will be working with the following simple Spark project.

The project consists of only three files; build.sbt, build.properties, and our actual Spark application, DemoDeploy.scala. It’s laid out as follows:

├── build.sbt

├── project

│   └── build.properties

└── src

└── main

└── scala

└── DemoDeploy.scala

Our build.sbt is as simple as possible. For now, we include only Spark as a dependency:

scalaVersion := "2.11.9"

name := "simple-spark-deploy"

version := "0.1"

val sparkVersion = "2.2.0"

libraryDependencies ++= Seq (

"org.apache.spark" %% "spark-core" % "2.2.0"

)

Our build.properties simply indicates which version of sbt we’ll be using. For this project, I’ve been using 0.13.16:

sbt.version=0.13.16

And finally, our simple Spark program is as follows:

package demodeploy

import org.apache.spark.SparkConf

import org.apache.spark.SparkContext

import org.apache.spark.SparkContext.\_

import org.apache.spark.rdd.RDD

object DemoDeploy {

def main(args: Array[String]) {

val conf = new SparkConf().setAppName("DemoDeploy")

val sc = new SparkContext(conf)

println("====DEMO DEPLOY====")

val text = List("Hadoop MapReduce, a disk-based big data processing engine, is being replaced by a new generation of memory-based processing frameworks, the most popular of which is Spark.", "Spark supports Scala, Java, Python, and R.")

val rdd = sc.parallelize(text)

val counts = rdd.flatMap(line => line.split(" "))

.map(word => (word, 1))

.reduceByKey(\_ + \_)

.collect()

counts foreach println

sc.stop()

}

}

Here, we do word count on an RDD[String], we collect the result as an array of pairs at the master node, and we print each element.

## Deploying Our Simple Spark Project

To deploy this job on our running cluster [from part 1](https://heather.miller.am/blog/launching-a-spark-cluster-part-1.html), we’ve first got to **package it up up as a jar to send to the master/worker nodes in our cluster.**

### Packaging Our Job

Most of the time, it’s sufficient to use the package task that comes built-in with sbt to assmble a jar that you can ship to your cluster nodes.

To do that, just cd into your project’s root directory, and run sbt package.

$ sbt package

[info] Loading global plugins from /Users/hmiller/.sbt/0.13/plugins

[info] Loading project definition from /Users/hmiller/Dropbox/git-shared/simple-spark-deploy/project

...

[info] Compiling 1 Scala source to /Users/hmiller/Dropbox/git-shared/simple-spark-deploy/target/scala-2.11/classes...

[info] Packaging /Users/hmiller/Dropbox/git-shared/simple-spark-deploy/target/scala-2.11/simple-spark-deploy\_2.11-0.1.jar ...

[info] Done packaging.

[success] Total time: 12 s, completed Apr 2, 2018 9:44:30 AM

Take note of the name and location of the packaged jar,/Users/hmiller/Dropbox/git-shared/simple-spark-deploy/target/scala-2.11/simple-spark-deploy\_2.11-0.1.jar in this case.

### Moving Our jar to the Cluster

In order for our Spark cluster to run our newly-packaged application, we simply have to copy it over to the master and worker nodes in the Spark cluster (named my-spark-cluster) we started in [part 1](https://heather.miller.am/blog/launching-a-spark-cluster-part-1.html).

To do that, we use Flintrock’s copy-file command, which takes 3 arguments.

$ flintrock copy-file [cluster-name] [path-to-local-jar] [path-to-remote-destination-of-jar]

Since the user we created on our Spark cluster was called ec2-user, the remote directory we must copy our packaged jar to is simply /home/ec2-user.

So in this case, to copy our newly-packaged jar to the master and worker nodes in my-spark-cluster, we just run:

flintrock copy-file my-spark-cluster \

/Users/hmiller/Dropbox/git-shared/simple-spark-deploy/target/scala-2.11/simple-spark-deploy\_2.11-0.1.jar \

/home/ec2-user/

If you’d like to login to your master node and double-check that your jar is where it should be, or for any other sort of debugging on the master node, you can use Flintrock for this too. Simply do:

$ flintrock login my-spark-cluster

This sshes you into your my-spark-cluster’s master node, in your user’s home directory. E.g.,:

[ec2-user@ip-172-31-26-137 ~]$ pwd

/home/ec2-user

### Submitting with **spark-submit**

Finally, to actually run our job on our cluster, we must use the spark-submit script that comes with Spark.

We’ll need a few pieces of information to do the most minimal submit possible. Those include:

* **the entry point for your Spark application**, i.e., the class where your main method resides.
* **the URL of the master node of your cluster**. Note: we can use Flintrock to get this!
* **the name and path of your application jar** that resides on your cluster nodes

The entry point of our Spark application is simply demodeploy.DemoDeploy, since DemoDeploy.scala contains the main method of our application.

To obtain the URL of the master node of our cluster, we use Flintrock. Flintrock’s describe command provides basic information such as URLs about the cluster(s) you have running.

$ flintrock describe

Found 1 cluster in region us-east-1.

---

my-spark-cluster:

state: running

node-count: 2

master: ec2-107-21-87-63.compute-1.amazonaws.com

slaves:

- ec2-34-234-87-162.compute-1.amazonaws.com

Thus, in this case, our master node’s URL is ec2-107-21-87-63.compute-1.amazonaws.com. Since we’re running our Spark cluster in standalone mode, the URL we must pass to spark-submit will look like: spark://ec2-107-21-87-63.compute-1.amazonaws.com:7077. Here, spark:// indicates that we’re running in Spark standalone mode, and 7077 is the default port that the master node is configured to use in standalone mode.

Finally, recall that we copied our application jar to the /home/ec2-user/ folder on our master and worker nodes, and that the jar is called simple-spark-deploy\_2.11-0.1.jar.

We can submit our job to Spark using spark-submit as follows:

$ ./bin/spark-submit \

--class demodeploy.DemoDeploy \

--master spark://ec2-107-21-87-63.compute-1.amazonaws.com:7077 \

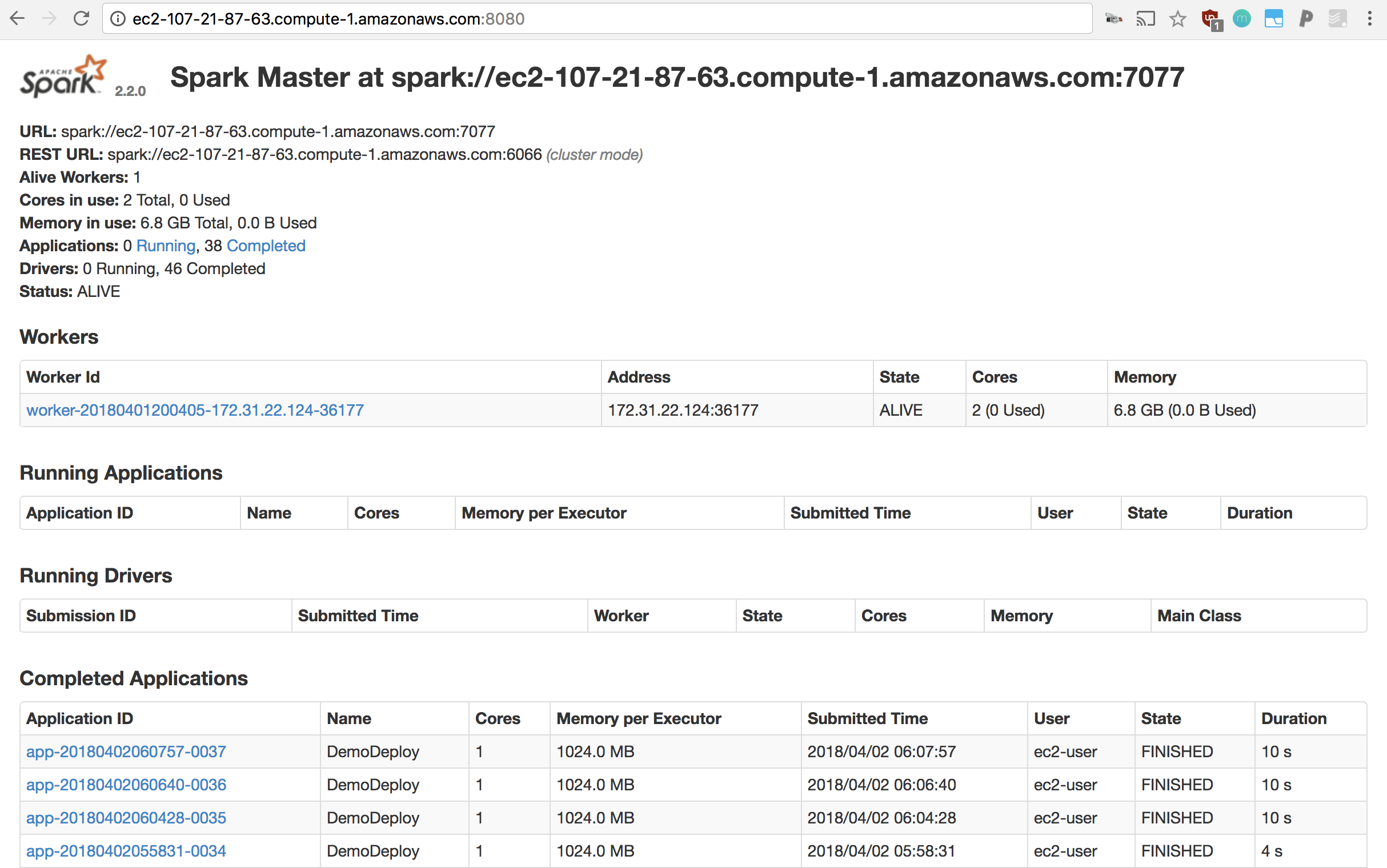
--deploy-mode cluster \

/home/ec2-user/simple-spark-deploy\_2.11-0.1.jar

Note that the you can think of the --deploy-mode cluster option as essentially a directive to Spark that you’re submitting your job from your laptop to a faraway cluster. That is, you’re sending the job to another machine that should act as the master, and that the output from the job should be printed to that machine’s stdout/stderr.

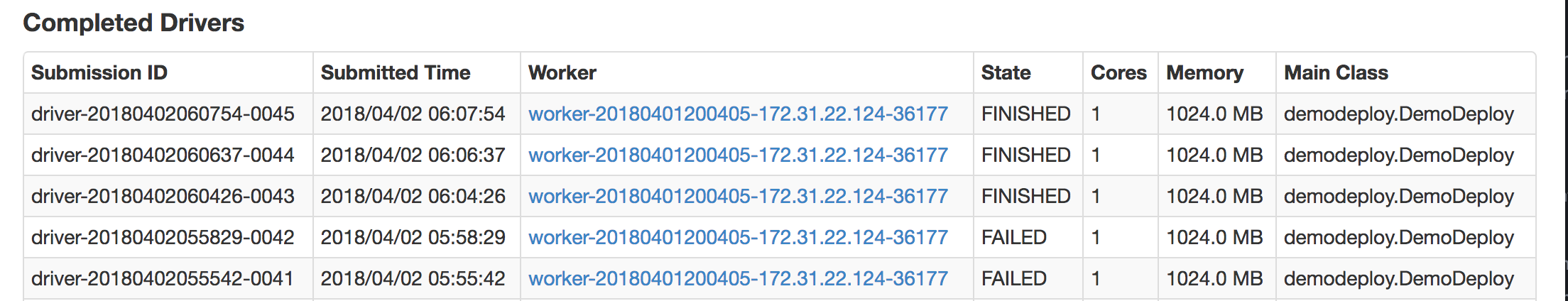
### Seeing the Results

To see the result of your job, simply point your browser to your master’s URL with port 8080 appended to the URL. For example, http://ec2-107-21-87-63.compute-1.amazonaws.com:8080



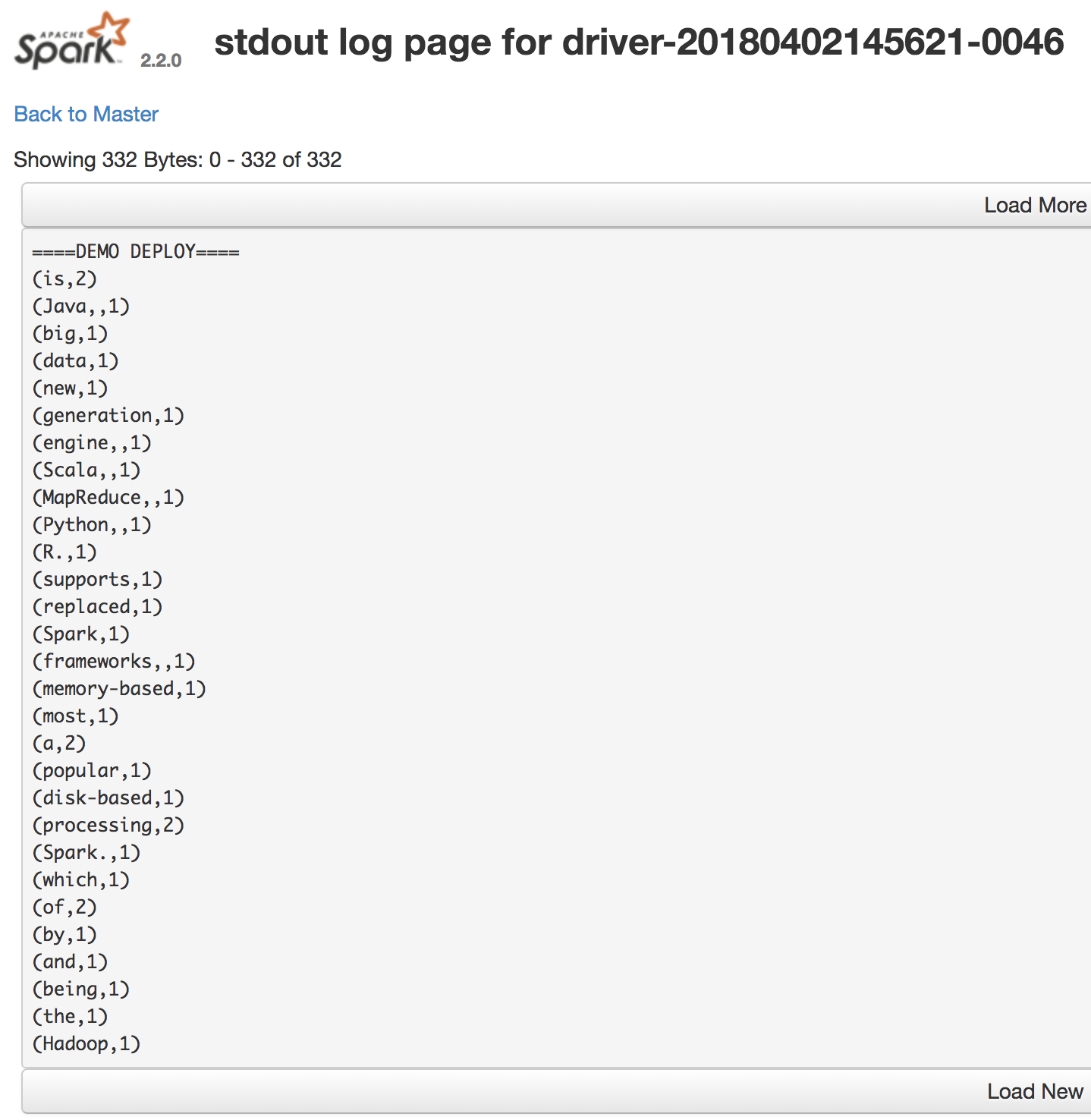
If your job is long-running, refreshing this page should allow you to see it under “Running Applications”.

Else, completed jobs can be inspected under “Completed Drivers”.



There, you can see the list of jobs that have finished, failed, errored, etc, and you can click each one to inspect its stdout and stderr (**scroll down to see the stdout/stderr for each driver!, under “Finished Drivers”**)

For example:



## Adding Dependencies

Most of the time, adding dependencies and ensuring that they’re available on each cluster node is possible by simply using sbt’s package task.

As an example, let’s add and use two libraries to our project; breeze, an NLP/linear algebra library, and spire, a mathematical library.

We must simply update our build.sbt file’s dependencies as follows:

libraryDependencies ++= Seq (

"org.apache.spark" %% "spark-core" % "2.2.0",

"org.scalanlp" %% "breeze" % "0.13.2", *// nlp/linear algebra library, example dependency*

"org.typelevel" %% "spire" % "0.14.1" *// mathematical library, example dependency*

)

And in DemoDeploy.scala, we must simply add appropriate imports and use the imported methods. For example:

package demodeploy

import breeze.linalg.\_

import spire.implicits.\_

import spire.math.\_

import org.apache.spark.SparkConf

import org.apache.spark.SparkContext

import org.apache.spark.SparkContext.\_

import org.apache.spark.rdd.RDD

object DemoDeploy {

def main(args: Array[String]) {

val conf = new SparkConf().setAppName("DemoDeploy")

val sc = new SparkContext(conf)

println("====DEMO DEPLOY====")

println("OUTPUT OF BREEZE LIBRARY:")

println(DenseVector.zeros[Double](5))

println("OUTPUT OF SPIRE:")

println(Complex(3.0, 5.0).sin)

val text = List("Hadoop MapReduce, a disk-based big data processing engine, is being replaced by a new generation of memory-based processing frameworks, the most popular of which is Spark.", "Spark supports Scala, Java, Python, and R.")

val rdd = sc.parallelize(text)

val counts = rdd.flatMap(line => line.split(" "))

.map(word => (word, 1))

.reduceByKey(\_ + \_)

.collect()

counts foreach println

sc.stop()

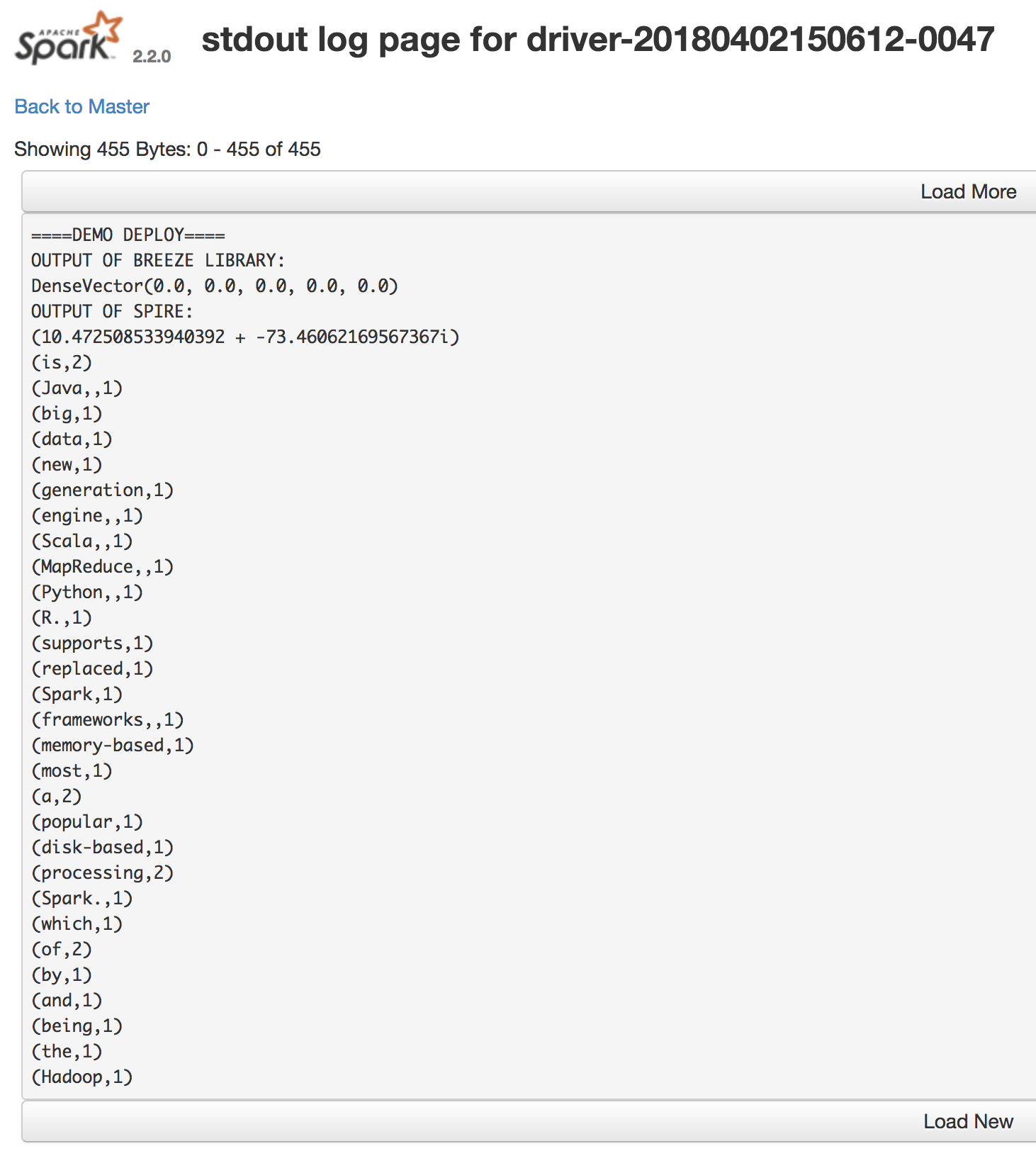
}

}

To run this on our Spark cluster, we simply repeat the steps from above, namely:

1. run sbt package to package up a jar to send to the cluster.
2. copy the packaged jar to the cluster nodes using Flintrock.
3. run spark-submit as configured above.

In the Spark UI, you should be able to see the following output for that driver’s stdout:



## Connecting to S3

### Getting Data on S3

In order to get data onto S3, you may either use the AWS command-line tools, or S3’s UI in the AWS console. We’ll be using the command-line interface.

To start, first grab awscli, which you can get using Python’s pip:

$ pip install awscli

(This assumes that you have your AWS environment variables properly set, as explained in [*part 1*](https://heather.miller.am/blog/launching-a-spark-cluster-part-1.html#creating-a-new-aws-user-for-flintrock))

Once installed, create a bucket as follows (we will call ours simple-spark-deploy-bucket):

$ aws s3 mb s3://simple-spark-deploy-bucket

We can upload a file ([rock-and-roll-wiki.txt](https://heather.miller.am/resources/files/rock-and-roll-wiki.txt)) to our bucket as follows:

aws s3 cp ./rock-and-roll-wiki.txt s3://simple-spark-deploy-bucket

### Using That Data From Your Spark Job

Now, let’s say we’d like to use this text file in our S3 bucket as our input in the word count example we’ve been working with.

To do so, we must simply add the following line to DemoDeploy.scala:

val rdd = sc.textFile("s3a://simple-spark-deploy-bucket/rock-and-roll-wiki.txt")

Notice that we are using s3a rather than s3/s3n. This is because s3a is known to have better performance, and to be the successor to s3n.

Our full DemoDeploy.scala should then look like:

package demodeploy

import breeze.linalg.\_

import spire.implicits.\_

import spire.math.\_

import org.apache.spark.SparkConf

import org.apache.spark.SparkContext

import org.apache.spark.SparkContext.\_

import org.apache.spark.rdd.RDD

object DemoDeploy {

def main(args: Array[String]) {

val conf = new SparkConf().setAppName("DemoDeploy")

val sc = new SparkContext(conf)

println("====DEMO DEPLOY====")

println("OUTPUT OF BREEZE LIBRARY:")

println(DenseVector.zeros[Double](5))

println("OUTPUT OF SPIRE:")

println(Complex(3.0, 5.0).sin)

val rdd = sc.textFile("s3a://simple-spark-deploy-bucket/rock-and-roll-wiki.txt")

val counts = rdd.flatMap(line => line.split(" "))

.map(word => (word, 1))

.reduceByKey(\_ + \_)

.collect()

counts foreach println

sc.stop()

}

}

**IMPORTANT!:** Before we can go any further, we need to ensure that a handful of specific dependencies for S3 are available on our Spark cluster. As of the time of writing, this is a workaround, which can be solved by downloading a recent Hadoop 2.7.x distribution and a specific, older version of an AWS JAR (1.7.4) that is typically not available in the EC2 Maven Repository.

To do this, simply download the following two jars somewhere on your local machine:

* [hadoop-aws-2.7.2.jar](http://central.maven.org/maven2/org/apache/hadoop/hadoop-aws/2.7.2/hadoop-aws-2.7.2.jar)
* [aws-java-sdk-1.7.4.jar](http://central.maven.org/maven2/com/amazonaws/aws-java-sdk/1.7.4/aws-java-sdk-1.7.4.jar)

And then, using Flintrock, **copy them to the /home/ec2-user/spark/jars directory.**This will ensure that these dependencies are available to Spark:

$ flintrock copy-file my-spark-cluster \

/path/to/aws-java-sdk-1.7.4.jar \

/home/ec2-user/spark/jars/

$ flintrock copy-file my-spark-cluster \

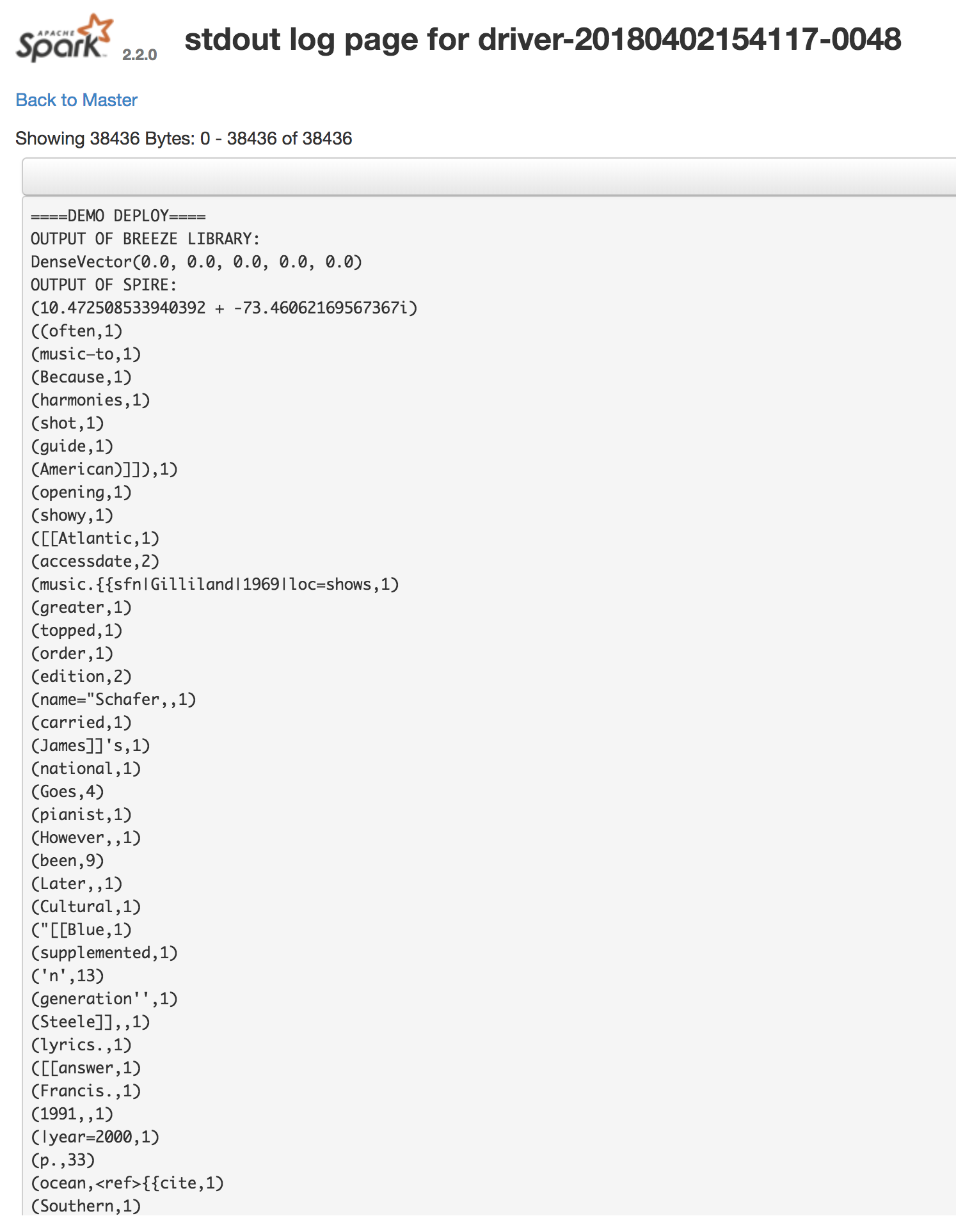
/path/to/hadoop-aws-2.7.2.jar \

/home/ec2-user/spark/jars/

To run this on our Spark cluster, we simply repeat the steps from above, namely:

1. run sbt package to package up a jar to send to the cluster.
2. copy the packaged jar to the cluster nodes using Flintrock.
3. run spark-submit as configured above.

In the Spark UI, you should be able to see the following output for that driver’s stdout:



Et voilà! Now you know how to submit Spark jobs to a running Spark cluster using Flintrock, how to add dependencies to your Spark job, and how to connect your Spark job to an S3 bucket