Lab: Intro to Spark 1.1 on DSE 4.6

Cassandra + Spark

Lab created on: Sept 2, 2014 (last updated Dec 9, 2014)
(*please send edits and corrections to*): sameerf@databricks.com

This lab was created with collaboration from engineers at DataStax and Databricks, specifically: Piotr Kołaczkowski (DS), Holden Karau (DB), Pat McDonough (DB), Patrick Wendell (DB) and Matei Zaharia (DB).

Estimated lab completion time: **2.5 hours**

Objective:

This lab will introduce you to using Apache Spark 1.1 on DataStax Enterprise Edition 4.6.0 in the Amazon cloud. The lab assumes that the audience is a beginner to both Cassandra and Spark. So the document walks the reader through installing DSE, learning Cassandra and then learning Spark. The ultimate goal here is to introduce students to Cassandra + Spark in a devops manner: looking at config files, writing some simple CQL or Spark code, breaking things and troubleshooting issues, exploring the Spark source code, etc. Although the ideal way to use this lab is actually type + run the commands in a parallel environment, the lab can still be used for purely reading. All the output of the commands are pasted in this lab, so you can get a very clear idea of what would happen if you had actually run the command.

The following high level steps are part of this lab:

- Connect via SSH to your EC2 instance
- Create a new keyspace and table in C* and add data to it
- Start the scala based Spark shell
- Import the fresh data into a Spark RDD
- Persist an RDD to memory
- Write an RDD into Cassandra
- Launch the OpsCenter web GUI
- Launch the Spark web GUIs
- Locate the Spark logs

To run this lab, you will need one m3.large EC2 instance running in the us-west-2c (Oregon) data center.

The m3.large instance type comes with 2 vCPUs, 7.5 GB of RAM and one 32 GB SSD.

If you are running this lab in a Databricks training course, the instructor should have given you the following:

- Public DNS hostname of your specific instance
- .pem and .ppk key pairs to authenticate you to your instance

Resources to learn more about Cassandra

++ Apache Documentation ++
Cassandra homepage @ Apache:
https://cassandra.apache.org

Note there is a link at the bottom of the Apache Cassandra page to the Users mailing list, where you can get free tech support.

++ Planet Cassandra ++
Planet Cassandra is the community website for Cassandra, with lots of documentation, use cases, videos and meetup events:
http://planetcassandra.org/what-is-apache-cassandra/

Training resources can be found here:
http://planetcassandra.org/cassandra-training

++ DataStax ++
The official documentation on Cassandra from DataStax is one of the best resources to learn the Cassandra ecosystem:
http://www.datastax.com/docs

Here is specifically the Cassandra 2.1 documentation from DataStax:
http://www.datastax.com/documentation/cassandra/2.1/cassandra/gettingStartedCassandraIntro.html

Resources to learn more about Spark

++ Apache Documentation ++
Spark homepage @ Apache:
https://spark.apache.org

Here are the official Apache docs for Spark 1.1:
https://spark.apache.org/docs/1.1.0/

Link to user + dev mailing lists:
https://spark.apache.org/community.html#mailing-lists

++ Developer/API Documentation ++
Spark Core 1.1.0 Scala docs:
https://spark.apache.org/docs/1.1.0/api/scala/index.html#org.apache.spark.package

Spark Core 1.1.0 Java docs:
https://spark.apache.org/docs/1.1.0/api/java/index.html

Spark Core 1.1.0 API docs for Python:
https://spark.apache.org/docs/1.1.0/api/python/index.html

++ DSE 4.6 ++

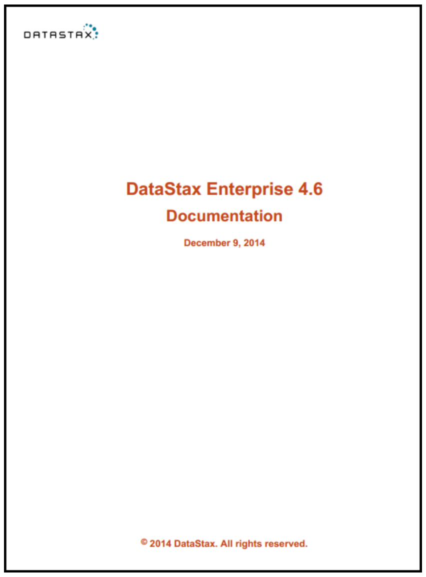
The ~400 page PDF from DataStax on DSE 4.6 includes ~60 pages of Spark documentation starting on page 83:
http://www.datastax.com/documentation/datastax_enterprise/4.6/pdf/dse46.pdf
or
http://www.datastax.com/documentation/datastax_enterprise/4.6/datastax_enterprise/spark/sparkIntro.html

++ DataStax's Cassandra <-> Spark connector ++
Source code and documentation is here:
https://github.com/datastax/spark-cassandra-connector

++ Spark Training Videos ++
From Spark Summit 2013: http://spark-summit.org/2013
From Spark Summit 2014: http://spark-summit.org/2014

2 Special Resources

A good portion of the steps in this lab come from the following 2 resources, so they are worth pointing out separately. If you want to dive deeper into Spark after completing this lab, then these are the 2 best technical resources to get started with:


DATASTAX

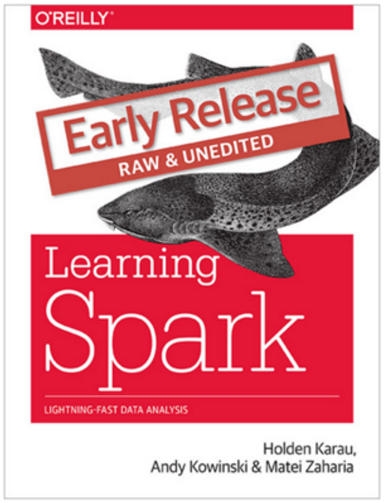
DataStax Enterprise 4.6

Documentation

December 9, 2014

© 2014 DataStax. All rights reserved.

DSE 4.6 PDF
Spark stuff starts on page 83
http://www.datastax.com/documentation/datastax_enterprise/4.6/pdf/dse46.pdf

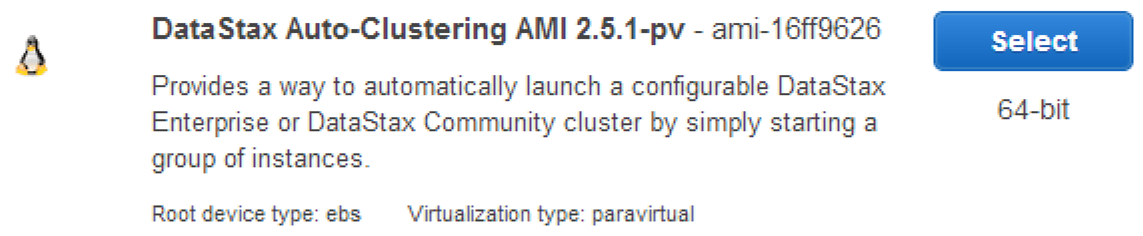

O'REILLY
Early Release
RAW & UNEDITED
Learning Spark
LIGHTNING-FAST DATA ANALYSIS
Holden Karau, Andy Kowinski & Matei Zaharia

Learning Spark
Early Release version available now.
http://shop.oreilly.com/product/0636920028512.do

Notes about the initialization of AMI (`ami-16ff9626`)

The m3.large EC2 instance you will be connecting to was launched using the steps documented here:
http://datastax.com/documentation/datastax_enterprise/4.5/datastax_enterprise/install/installAMIlaunch.html

This 64-bit AMI was launched in the us-west-2 (Oregon) datacenter to build your EC2-instance (*this AMI is not accessible in other regions*):


DataStax Auto-Clustering AMI 2.5.1-pv - ami-16ff9626

Provides a way to automatically launch a configurable DataStax Enterprise or DataStax Community cluster by simply starting a group of instances.

Root device type: ebs   Virtualization type: paravirtual

Note this AMI is a ParaVirtualization (PV) type, which traditionally performs better than Hardware Virtual Machines (HVM).

The following options were added as text to the User Data section during launch:

```
--clustername DSEcluster01 --totalnodes 1 --version enterprise --username
myname_gmail.com --password xxx --analyticsnodes 1 --searchnodes 0 --release
4.6.0-1
```

Also, the 8 GB root device storage type (/dev/sda1) was set to a 20 GB General Provisioned SSD w/ 24 baseline IOPS and 3000 burst IOPS.

As a reminder, the DataStax AMI does the following:

- Installs the latest version of DataStax Enterprise (4.5.1) with an Ubuntu 12.04 LTS 64-bit (Precise Pangolin), image (Ubuntu Cloud 20140227 release), Kernel 3.8+.
- Installs Oracle Java 7 (Java Hotspot 64-bit: Java 1.7.0_51).
- Install metrics tools such as dstat, ethtool, make, gcc, and s3cmd.
- Uses RAID0 ephemeral disks for data storage and commit logs.
- Choice of PV (Para-virtualization) or HVM (Hardware-assisted Virtual Machine) instance types.
- Launches EBS-backed instances for faster start-up, not database storage.
- Uses the private interface for intra-cluster communication.
- Starts the nodes in the specified mode (Real-time, Analytics, or Search).
- Sets the seed nodes cluster-wide.
- Installs the DataStax OpsCenter on the first node in the cluster (by default).

No one has logged into this 1-node cluster via SSH or a web UI since it launched. You will be the first to do so in the next section…

Connect to EC2 Instance

Pick your favorite SSH client and connect to the EC2 instance with the connection details below:

Port: `22`
Username: `ubuntu`
Hostname: *`<public DNS hostname instructor provided you with>`*
Key Pair: `.pem (for OS X) or .ppk (for Windows)`

I recommend using **PuTTY** on Windows and **iTerm2** (or Terminal) on OS X.

The general instructions to log in via OS X Terminal are:

Open up your terminal app of choice and type in the following…

Change the permissions of the `.pem` key file like this: **`chmod 400 my-key-pair.pem`**

SSH into the VM using this command: **`ssh -i Amazon-Private-Key.pem ubuntu@<public hostname of VM>`**

Say `Yes` to this prompt:

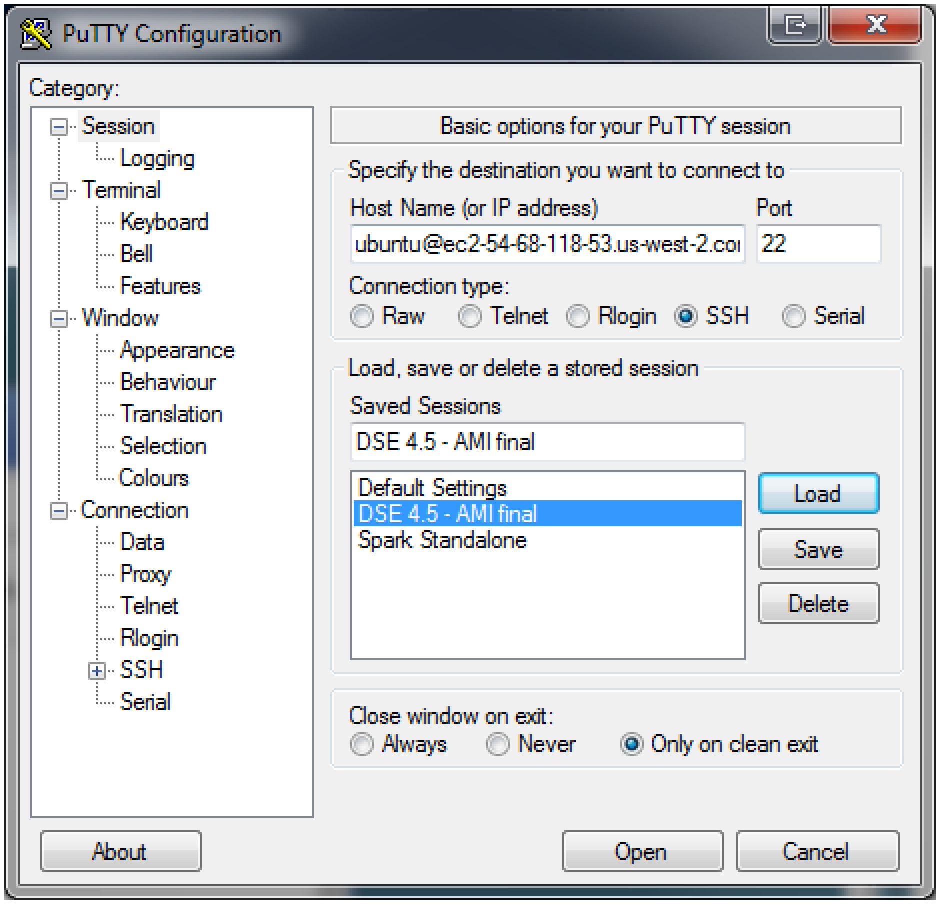
```
The authenticity of host 'ec2-198-51-100-x.compute-1.amazonaws.com
(10.254.142.33)'can't be established.RSA key fingerprint is
1f:51:ae:28:bf:89:e9:d8:1f:25:5d:37:2d:7d:b8:ca:9f:f5:f1:6f.Are you sure you
want to continue connecting (yes/no)? yes
```

Here are the official details on how to log in via Mac:
http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/

Connection Settings screenshots for PuTTY (on Windows):

Download PuTTY.exe from:
http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

Notice that appending '`ubuntu@`' to the Host Name will automatically log you into the VM:



Specify the destination you want to connect to

Host Name (or IP address): ubuntu@ec2-54-68-118-53.us-west-2.coı
Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

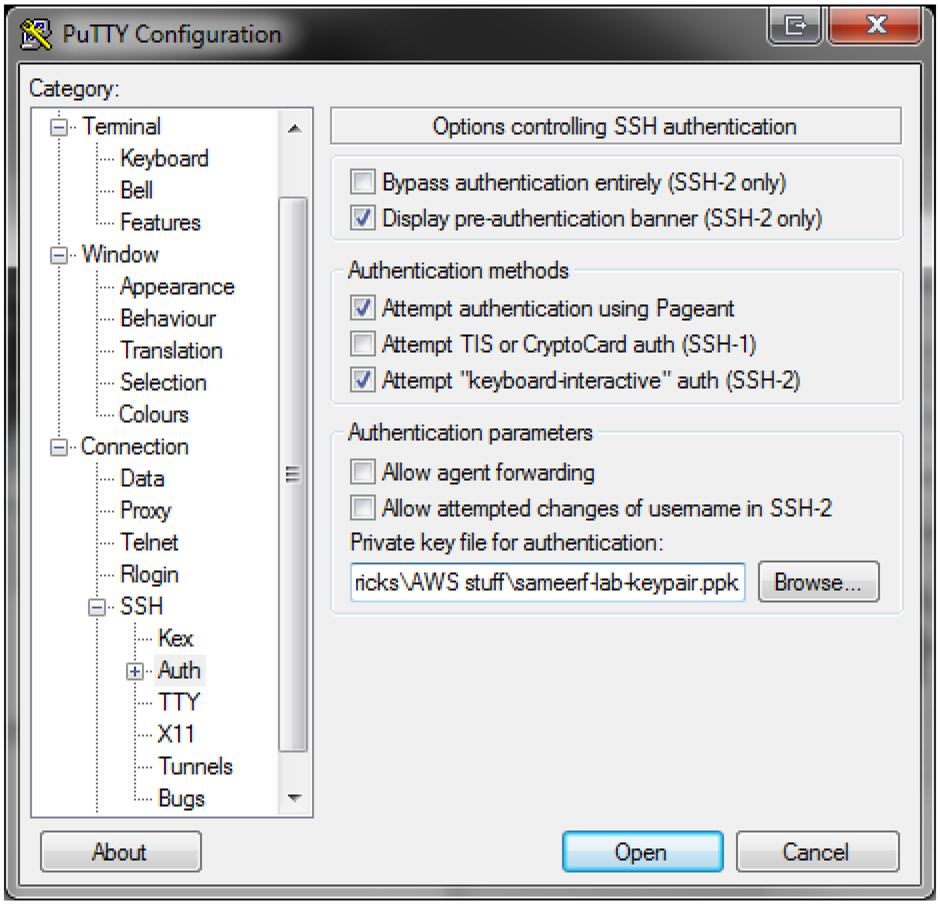
Load, save or delete a stored session

Saved Sessions: DSE 4.5 - AMI final

Default Settings
DSE 4.5 - AMI final
Spark Standalone

Close window on exit: Always Never [✓ Only on clean exit]

In the left pane, under Connection > SSH > Auth, you can provide the location to the private .ppk key on your Windows machine:



Options controlling SSH authentication

- [ ] Bypass authentication entirely (SSH-2 only)
- [✓] Display pre-authentication banner (SSH-2 only)

Authentication methods

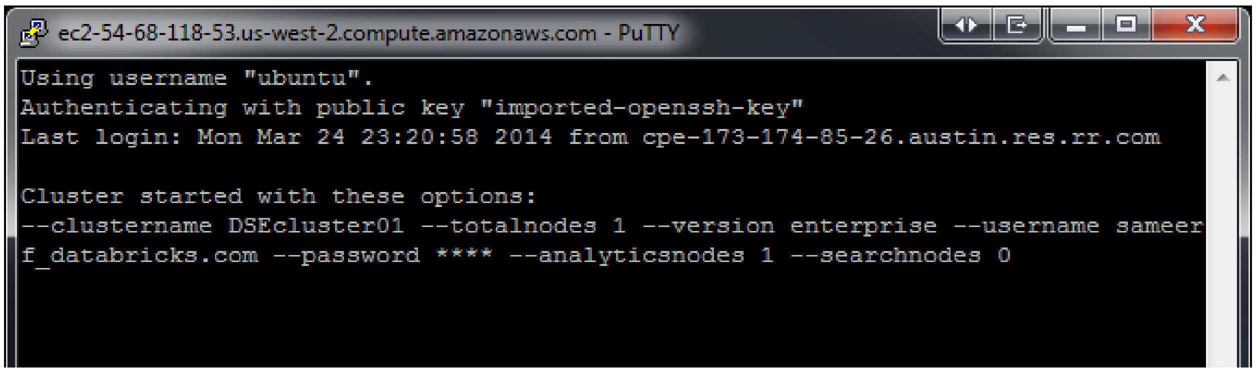
- [✓] Attempt authentication using Pageant
- [ ] Attempt TIS or CryptoCard auth (SSH-1)
- [✓] Attempt "keyboard-interactive" auth (SSH-2)

Authentication parameters

- [ ] Allow agent forwarding
- [ ] Allow attempted changes of username in SSH-2

Private key file for authentication: ricks\AWS stuff\sameerf-lab-keypair.ppk

This is what you'll see once you are successfully logged in:



```
Using username "ubuntu".
Authenticating with public key "imported-openssh-key"
Last login: Mon Mar 24 23:20:58 2014 from cpe-173-174-85-26.austin.res.rr.com

Cluster started with these options:
--clustername DSEcluster01 --totalnodes 1 --version enterprise --username sameer
f_databricks.com --password **** --analyticsnodes 1 --searchnodes 0
```

Upon connecting to the server, you should see the following output at the cmd prompt:

```
Using username "ubuntu".
Authenticating with public key "imported-openssh-key"
Last login: Mon Mar 24 23:20:58 2014 from cpe-173-174-85-26.austin.res.rr.com

Cluster started with these options:
--clustername DSEcluster01 --totalnodes 1 --version enterprise --username
sameer_blueplastic.com --password **** --analyticsnodes 1 --searchnodes 0


Raiding complete
Waiting for nodetool...
The cluster is now in it's finalization phase. This should only take a
moment...

Note: You can also use CTRL+C to view the logs if desired:
    AMI log: ~/datastax_ami/ami.log
    Cassandra log: /var/log/cassandra/system.log


Datacenter: Analytics
=====================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Owns (effective)  Host ID
Token                                      Rack
UN  10.0.1.28  41.07 KB   100.0%
3f40dc5e-a247-45a9-96ca-246a37301c83  -9223372036854765808
rack1

Opscenter: http://ec2-54-149-142-232.us-west-2.compute.amazonaws.com:8888/
    Please wait 60 seconds if this is the cluster's first start...


Tools:
    Run: datastax_tools
Demos:
    Run: datastax_demos
Support:
    Run: datastax_support
```

```
------------------------------------
DataStax AMI for DataStax Enterprise
and DataStax Community
AMI version 2.5
DataStax Enterprise version 4.6.0-1

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

spark-env.sh configuration file

The C* YAML file for this installation is at:
`/etc/dse/cassandra/cassandra.yaml`

Open the file using vi or nano and spend a few minutes skimming through this file. Most of the Cassandra-level settings are located here.

- On line 31 of the YAML file, you will notice that the `initial_token` line is not commented out, however line 25 for `num_tokens` is commented out. Therefore vnodes are not enabled by default in this cluster.

In your EC2 instance, the Spark configuration file is located here:
`/etc/dse/spark/spark-env.sh`

Note that if you were using the DataStax Enterprise GUI/Text Installer (`DataStaxEnterprise-4.5.x-linux-x64-installer.run`), the configuration file would actually be here:
`/usr/share/dse/resources/spark/conf/spark-env.sh`

Before continuing, take a couple of minutes to open the `spark-env.sh` file (using vim, emacs or nano) and notice that:

This spark-env.sh file contains the ports for the Master and the Web UIs:
```
export SPARK_MASTER_PORT=7077
export SPARK_MASTER_WEBUI_PORT=7080
export SPARK_WORKER_WEBUI_PORT=7081
```

It also has settings for memory consumption *(The first 2 settings are commented out, so they're being pulled from dse.yaml)***:**
```
# Set the amount of memory used by Spark Worker - if uncommented, it
overrides the setting initial_spark_worker_resources in dse.yaml.
# export SPARK_WORKER_MEMORY=2048m

# Set the number of cores used by Spark Worker - if uncommented, it overrides
the setting initial_spark_worker_resources in dse.yaml.
# export SPARK_WORKER_CORES=4

# The amount of memory used by SparkShell in the client environment.
export SPARK_REPL_MEM="256M"
```

```
# The amount of memory used by Spark Driver program
export SPARK_DRIVER_MEMORY="512M"
```

Note: SPARK_REPL_MEM is a left-over from an earlier Spark version where SPARK_DRIVER_MEM was not present. DataStax is going to remove this setting in the next version, so please only use SPARK_DRIVER_MEMORY and don't use `SPARK_REPL_MEM`.

The spark-env.sh file also points to the tmp and log directories:

```
# Directory for Spark temporary files. It will be used by Spark Master, Spark
Worker, Spark Shell and Spark applications.
export SPARK_TMP_DIR="/tmp/spark"

# Directory where RDDs will be cached
export SPARK_RDD_DIR="/var/lib/spark/rdd"

# The directory for storing master.log and worker.log files
export SPARK_LOG_DIR="/var/log/spark"
```

There are also some Spark related settings in the following file:
`/etc/dse/dse.yaml`

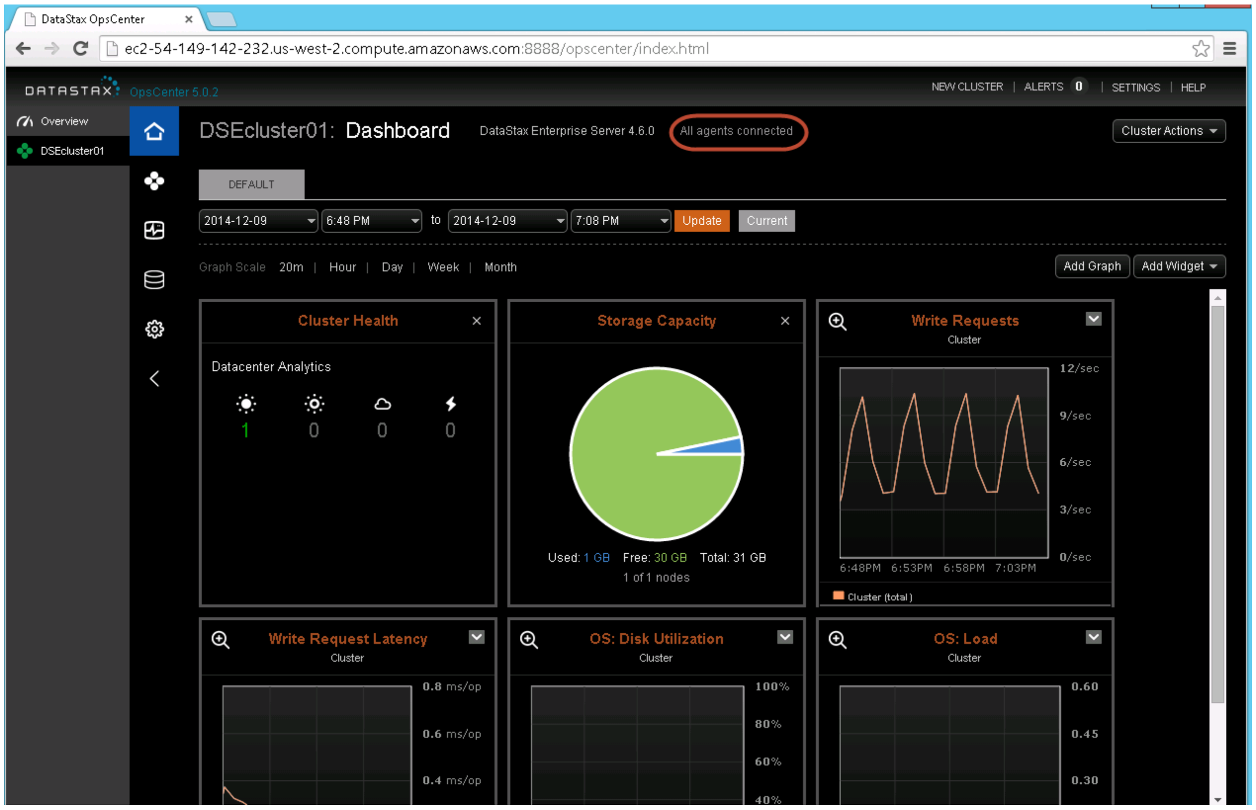
The main Spark setting in the `dse.yaml` file is:

```
# The fraction of available system resources to be used by Spark Worker
#
# This the only initial value, once it is reconfigured, the new value is
stored
# and retrieved on next run.
initial_spark_worker_resources: 0.7
```

OpsCenter GUI

Open a browser window on your local machine and ensure that OpsCenter is working. This way any firewall or networking issues between your local computer and the AWS EC2 instance can be caught now.

Open a Chrome or FireFox browser window and go to:
`http://<your EC2 instance's public DNS hostname>:8888`



In the above window, ensure that you see "All agents connected" at the top.

Take a few minutes to click around the OpsCenter 5.0 GUI to familiarize yourself with what sort of metrics are exposed.

Run Basic Spark Commands + Explore Spark UIs

First, let's download a small text file that we'll use in Spark later.

```
ubuntu@ip-10-0-18-129:~$ cd ~
ubuntu@ip-10-0-18-129:~$ wget http://blueplastic.com/databricks/toy_file.txt
--2014-09-02 19:07:42--  http://blueplastic.com/databricks/toy_file.txt
Resolving blueplastic.com (blueplastic.com)... 74.220.207.68
Connecting to blueplastic.com (blueplastic.com)|74.220.207.68|:80...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 121 [text/plain]
Saving to: `toy_file.txt'

100%[===================================================>] 121
--.-K/s   in 0s

2014-09-02 19:07:42 (12.6 MB/s) - `toy_file.txt' saved [121/121]
```

Take a look at what's inside the file:

```
ubuntu@ip-10-0-18-129:~$ cat toy_file.txt
This is the first line.
Look, the second line has a WHALE on it.
So, does the third: WHALE
But the 4th line does not :-(
```

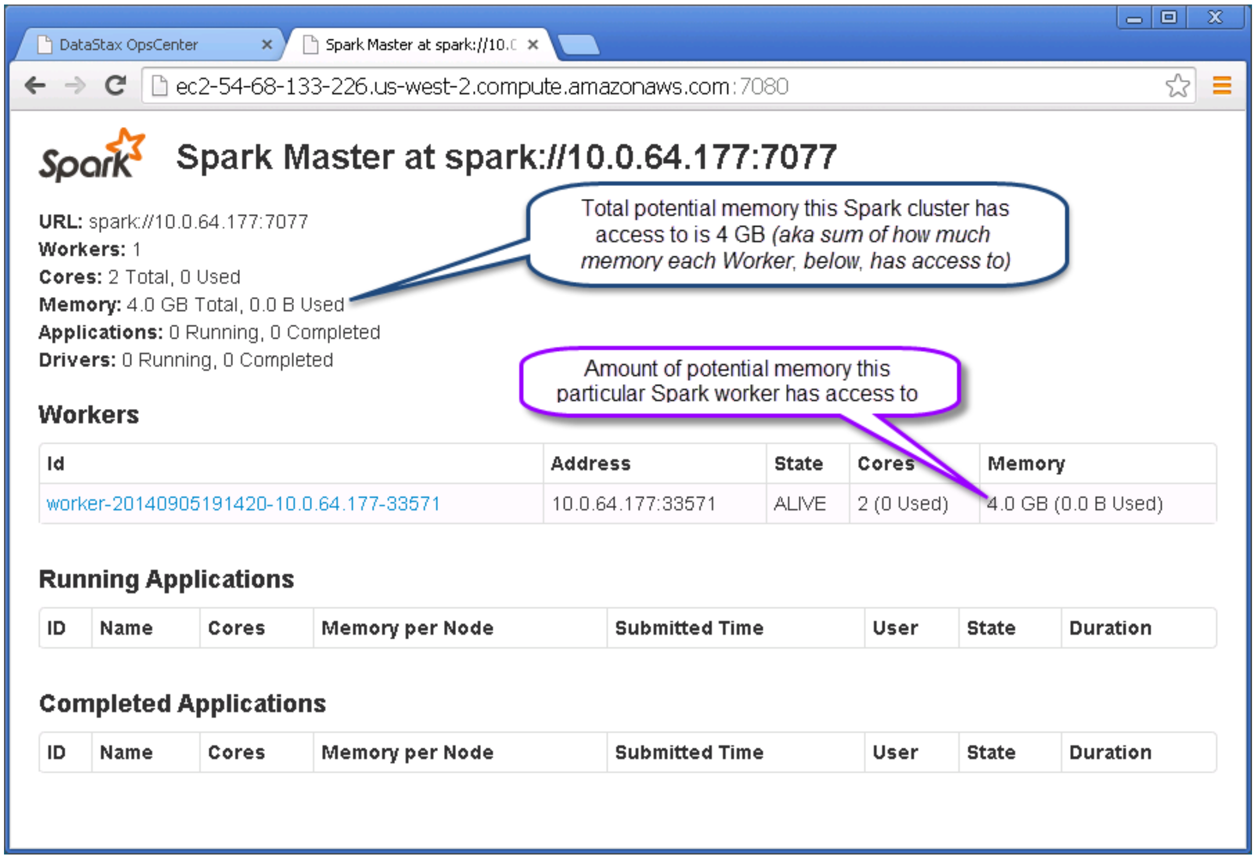
On to important things. Locate where the Spark Master server is running:

```
ubuntu@ip-10-0-18-129:~$ dsetool sparkmaster
spark://10.0.18.129:7077
```

Spark Master is running on the local node's private IP address at port 7077. *The private IP will be different for your unique instance.*

However, the Spark Master web GUI is located on port 7080 by default. See if you can reach it now by opening a new tab in your local browser and then go to:

`http://<your EC2 instance's public DNS hostname>:7080`


DataStax OpsCenter
Spark Master at spark://10.0
ec2-54-68-133-226.us-west-2.compute.amazonaws.com:7080
Total potential memory this Spark cluster has access to is 4 GB (aka sum of how much memory each Worker, below, has access to)
Amount of potential memory this particular Spark worker has access to

Spark Master at spark://10.0.64.177:7077

URL: spark://10.0.64.177:7077
Workers: 1
Cores: 2 Total, 0 Used
Memory: 4.0 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed

Workers

Id	Address	State	Cores	Memory
worker-20140905191420-10.0.64.177-33571	10.0.64.177:33571	ALIVE	2 (0 Used)	4.0 GB (0.0 B Used)

Running Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

In the Spark Master GUI, towards the top left, you can see that the **Spark cluster has access to up to 4 GB of memory**, and there are 0 Applications and 0 Drivers running. This field can be a bit misleading. This is not the amount of memory that the Spark Master JVM has access to! Rather the 4 GB value at the top is the sum of each Spark Worker's potential available memory (in executors).

Below that, in the middle of the page, under the Workers area, you can see one Worker is running in the cluster. Again, the GUI can be a bit misleading here also. The **4 GB of memory that you see at the far right of the screen under workers**, is NOT the amount of memory that this Spark worker JVM is using or has access to for its own JVM usage! Rather, it is the amount of potential memory that this Worker can allocate to any future Executors it has to launch. Right now we have 0 Applications (and therefore 0 Executors) running, so this worker shows "0.0 B Used" under its Memory details. However, consider the fact that the Worker JVM itself is actually running, so obviously the Worker must be using some RAM.

In the Spark configuration file (`/etc/dse/spark/spark-env.sh`) which you looked at earlier, there is a setting for:

```
# Set the amount of memory used by Spark Worker - if uncommented, it
overrides the setting initial_spark_worker_resources in dse.yaml.
# export SPARK_WORKER_MEMORY=2048m
```

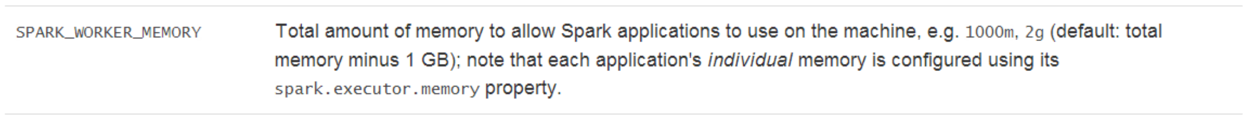
This setting is commented out, but if was uncommented, it would dictate that each Spark Worker in this cluster can allocate up to a 2048 MB of RAM limit total to its child Executors. In other words, it is the total amount of memory for running all Spark applications on a machine. From DSE 4.6 PDF: "*The value must be less than or equal to the amount of free memory on the machine after running Cassandra and all the OS-level processes. If you have 2 Worker nodes in your Spark cluster, that would mean that each Worker node can donate up to 2 GB of RAM, for a total of 4 GB available to the Spark Master to schedule applications on.*"

But actually… since the above setting is commented out, it is actually figuring out the Spark Worker Memory setting from the initial_spark_worker_resources in dse.yaml. This worker_resource setting is 0.7 in dse.yaml.

Here are the formulas used to figure out the worker memory and worker cores from this setting:

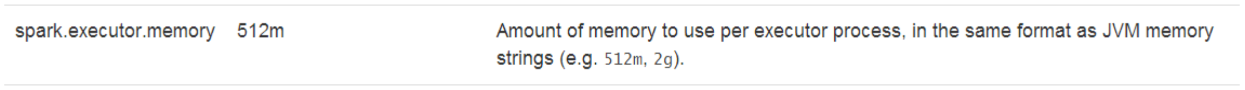
```
Spark worker memory = initial_spark_worker_resources * (total system memory -
memory assigned to C*)

Spark worker cores = initial_spark_worker_resources * total system cores
```

You can also find a description of the Spark Worker Memory setting (along with other environment variables explained here (scroll down a page or two):
https://spark.apache.org/docs/1.1.0/spark-standalone.html



SPARK_WORKER_MEMORY	Total amount of memory to allow Spark applications to use on the machine, e.g. `1000m`, `2g` (default: total memory minus 1 GB); note that each application's *individual* memory is configured using its `spark.executor.memory` property.

The above screenshot also refers to a property named '`spark.executor.memory`' in the description. You can find more details about Spark application-level settings like '`spark.executor.memory`' here:
https://spark.apache.org/docs/1.1.0/configuration.html



spark.executor.memory	512m	Amount of memory to use per executor process, in the same format as JVM memory strings (e.g. `512m`, `2g`).

The **`spark-env.sh`** file mentions:

```
# The default amount of memory used by a single executor (also, for the
executor used by Spark REPL).
# It can be modified in particular application by command line parameters.
512M is the default value.
# export SPARK_EXECUTOR_MEMORY="512M"
```

So, in the DSE 4.6 distribution of Spark, the default of 512 MB for each Executor has remained at 512 MB.

Finally, the `spark-env.sh` file also mentions these 2 settings:

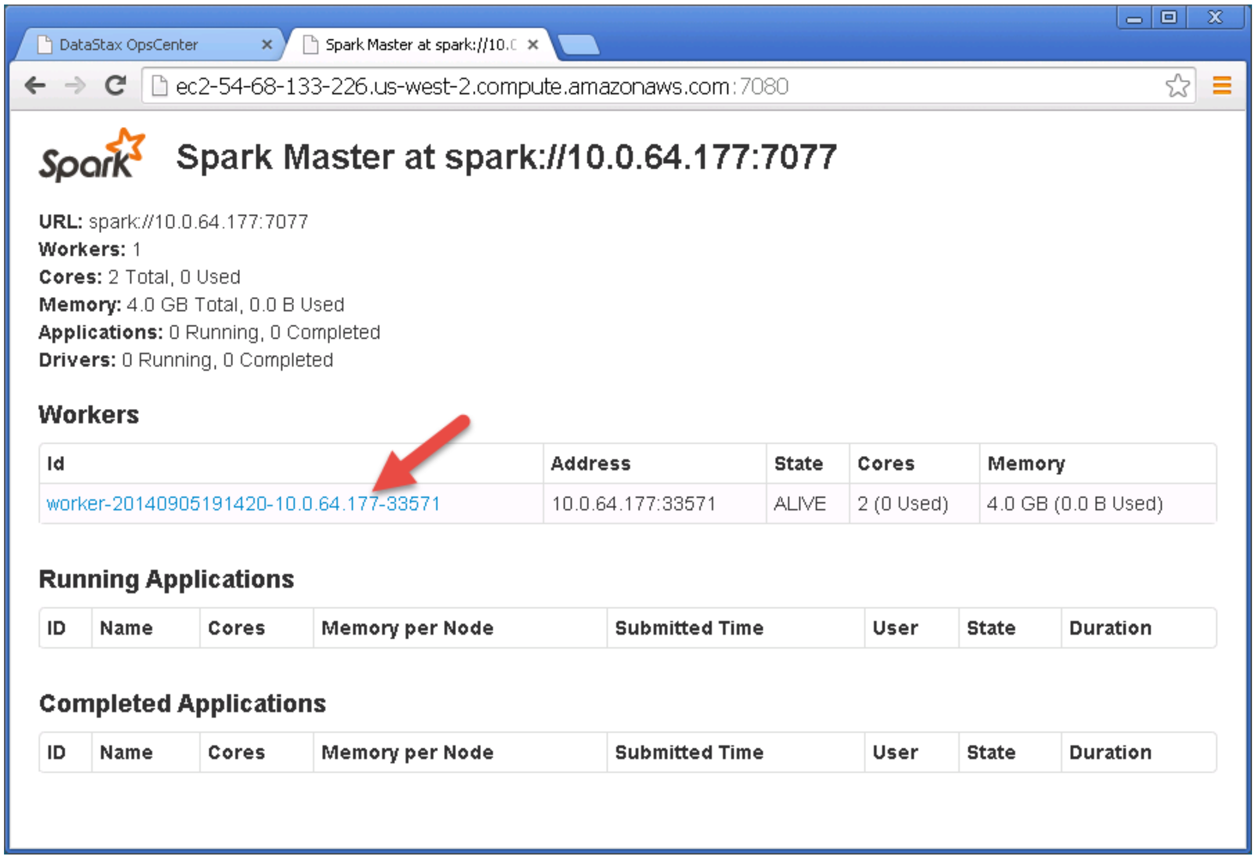
```
export SPARK_REPL_MEM="256M"
export SPARK_DRIVER_MEMORY="512M"
```

The `SPARK_REPL_MEM` environment variable is used by Spark in DataStax Enterprise, but not by open source Spark. The value of this variable controls the amount of memory assigned to the Spark scala shell. Configure a small amount of memory for the shell because the Spark shell hands off heavy processing to the workers, which ultimately passes the work down to the Executors.

The Spark shell is sometimes called a REPL for ‘read-eval-print-loop’. The Spark shell is essentially a modified version of the normal scala shell/REPL. We will be launching the Spark shell momentarily...

Note that in a future version of DSE, the export SPARK_REPL_MEM setting will be removed entirely and only the SPARK_DRIVER_MEMORY setting will remain. This DRIVER_MEMORY setting will affect both drivers and REPL. Essentially, a REPL is just a driver, so it makes sense to have one setting for both.

Try clicking on the link for the ALIVE worker (The red arrow in the screenshot below).


DataStax OpsCenter
Spark Master at spark://10.0
ec2-54-68-133-226.us-west-2.compute.amazonaws.com:7080

Spark Master at spark://10.0.64.177:7077

URL: spark://10.0.64.177:7077
Workers: 1
Cores: 2 Total, 0 Used
Memory: 4.0 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed

Workers

Id	Address	State	Cores	Memory
worker-20140905191420-10.0.64.177-33571	10.0.64.177:33571	ALIVE	2 (0 Used)	4.0 GB (0.0 B Used)

Running Applications

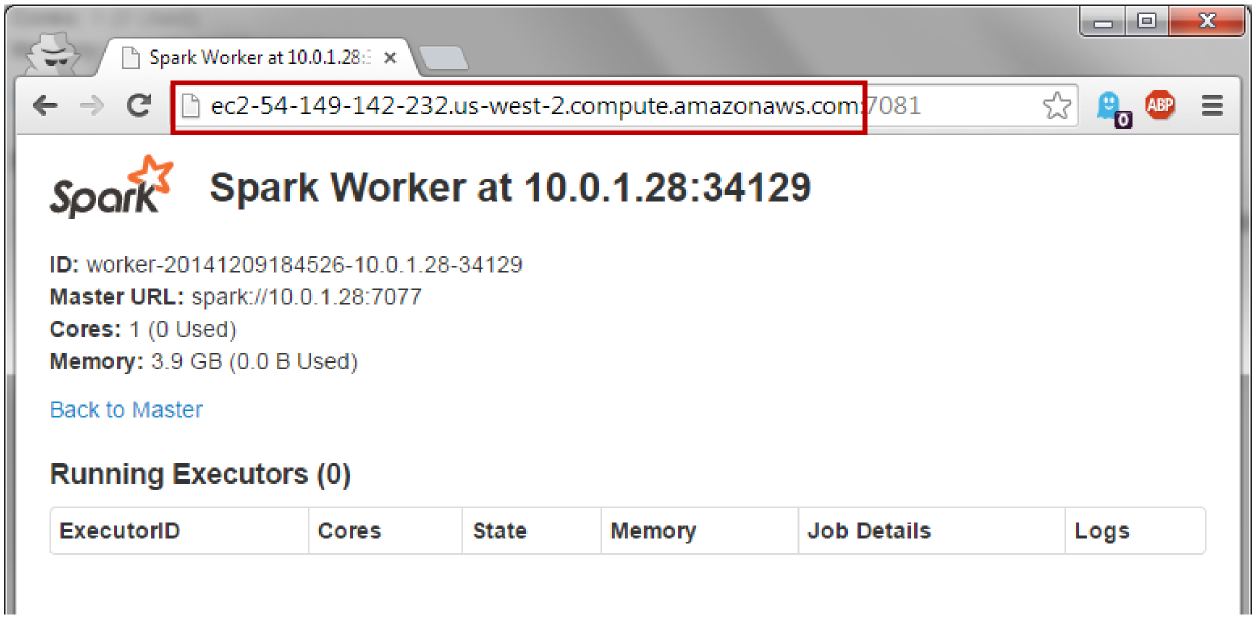
ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Warning: By default the Worker page will not load as the URL will have the private IP of the EC2 instance. Replace the private IP with the public DNS hostname and the page should load.

So, to see the Spark Worker UI, go to:
`http://<your EC2 instance’s public DNS hostname>:7081`


Spark Worker at 10.0.1.28:3
ec2-54-149-142-232.us-west-2.compute.amazonaws.com:7081

Spark Worker at 10.0.1.28:34129

ID: worker-20141209184526-10.0.1.28-34129
Master URL: spark://10.0.1.28:7077
Cores: 1 (0 Used)
Memory: 3.9 GB (0.0 B Used)

Back to Master

Running Executors (0)

ExecutorID	Cores	State	Memory	Job Details	Logs

You can see in the screen above that there are 0 Executors launched in this cluster… which makes sense since we have not run any applications yet.

Moving back to the cmd line, the "dsetool ring" command will show a (JT) under the Workload column next to the node where the Master is running.

```
ubuntu@ip-10-0-18-129:~$  dsetool ring
Note: Ownership information does not include topology, please specify a
keyspace.
Address          DC           Rack         Workload         Status   State
Load             Owns                 Token
10.0.18.129      Analytics    rack1        Analytics(JT)    Up       Normal
77.36 KB         100.00%              -9223372036854765808
```

Note, the specific 1-node cluster you are using does not have Hadoop installed on it at all. So, in the future, it may be better to change (JT) to (SM) for Spark Master. If a node was just running a Spark Worker, you would have seen a (TT) next to it… which would be good to change to (SW) for Spark Worker.

Run JPS to see what JVMs are currently running on this machine:

```
ubuntu@ip-10-0-64-177:~$ sudo jps
12509 jar
14703 DseSparkMaster
19734 Jps
7777 DseDaemon
14759 DseSparkWorker
```

We see the same Spark Master and Spark Worker JVMs launched with their respective pids that we explored in the GUI above.

Connect to the scala Spark shell:

```
ubuntu@ip-10-0-18-129:~$ dse spark
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.1.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_51)
Type in expressions to have them evaluated.
Type :help for more information.
Creating SparkContext...
Initializing SparkContext with MASTER: spark://10.0.1.28:7077
Created spark context..
Spark context available as sc.
HiveSQLContext available as hc.
CassandraSQLContext available as csc.
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

There is a `:sh` command in the Spark shell that lets you submit linux cmd line commands:

```
scala> :sh sudo jps
res4: scala.tools.nsc.interpreter.ProcessResult = `sudo jps` (7 lines, exit
0)
```

The res4 output that you see stands for ‘result #4’.

Now, print the output of result 2:

```
scala> res4.show
12509 jar
30843 SparkSubmit
22616 Jps
14703 DseSparkMaster
22541 CoarseGrainedExecutorBackend
7777 DseDaemon
14759 DseSparkWorker
```

Now that we’ve launched the Spark shell, more JVMs have instantiated to support the Shell, namely the `SparkSubmit` and `CoarseGrainedExecutorBackend`. The `SparkSubmit` is the driver for our ‘Spark shell” application and the `CoarseGrainedExecutorBackend` is the sole Executor running to support our application.

Let’s get some Java level details about `SparkReplMain`. We’ll need to execute an external process (jps) from the scala shell and use the pipe operator for this command, so first import the scala `process` package:

```
scala> import scala.sys.process._
import scala.sys.process._

scala> "jps -v" #| "grep SparkSubmit"
res6: scala.sys.process.ProcessBuilder =  ( [jps, -v] #| [grep, SparkSubmit]
)

scala> res6.run
res6: scala.sys.process.Process =
scala.sys.process.ProcessImpl$PipedProcesses@2f4e4a17

scala> 3354 SparkSubmit -Dspark.cassandra.connection.host=10.0.1.28
-Dspark.kryoserializer.buffer.mb=10
-Dspark.executorEnv.SPARK_CASSANDRA_AUTH_TOKEN=
-Djava.system.class.loader=com.datastax.bdp.loader.DseClientClassLoader
-XX:MaxPermSize=256M
```

```
-Djava.library.path=::/usr/share/dse/hadoop/native/Linux-amd64-64/lib
-Dspark.cassandra.connection.factory=com.datastax.bdp.spark.DseCassandraConne
ctionFactory
-Dspark.cassandra.auth.conf.factory=com.datastax.bdp.spark.DseAuthConfFactory
-Dcassandra.config.loader=com.datastax.bdp.config.DseConfigurationLoader
-Djava.io.tmpdir=/tmp/spark/ubuntu -Xms512M -Xmx512M
-javaagent:/usr/share/dse/cassandra/lib/jamm-0.2.5.jar
<hit enter to return to scala shell>
```

In the output above, you can see that the Spark shell/REPL/Driver uses 512 MB of RAM. The flag Xmx specifies the maximum memory allocation pool for a Java Virtual Machine (JVM)'s, while Xms specifies the initial memory allocation pool.

This means that the REPL JVM will be started with Xms amount of memory and will be able to use a maximum of Xmx amount of memory.

The import technique used above also demonstrates how to import scala packages into the Spark shell. You may read more about the `process` package here (or explore other scala packages):
http://www.scala-lang.org/api/current/index.html#scala.sys.process.package

The Spark Worker itself (`DseSparkWorker`) is a pretty light-weight JVM, which doesn't have heavy memory requirements. Let's verify this:

```
scala> "jps -v" #| "grep DseSparkWorker"
res8: scala.sys.process.ProcessBuilder =  ( [jps, -v] #| [grep,
DseSparkWorker] )

scala> res6.run
res9: scala.sys.process.Process =
scala.sys.process.ProcessImpl$PipedProcesses@1aecf762
```

Hmm, no output. What happened? Why aren't we seeing the details of the `DseSparkWorker` JVM? The problem is that the Spark Worker JVM was started under the 'cassandra' user's context (you will see this demonstrated via a Spark UI 4 or 5 pages down). However the `SparkSubmit` JVM (our REPL/driver) is running under the 'ubuntu' user. So there is a permissions mismatch.

Just sudo the command:

```
scala> "sudo jps -v" #| "grep DseSparkWorker"
```

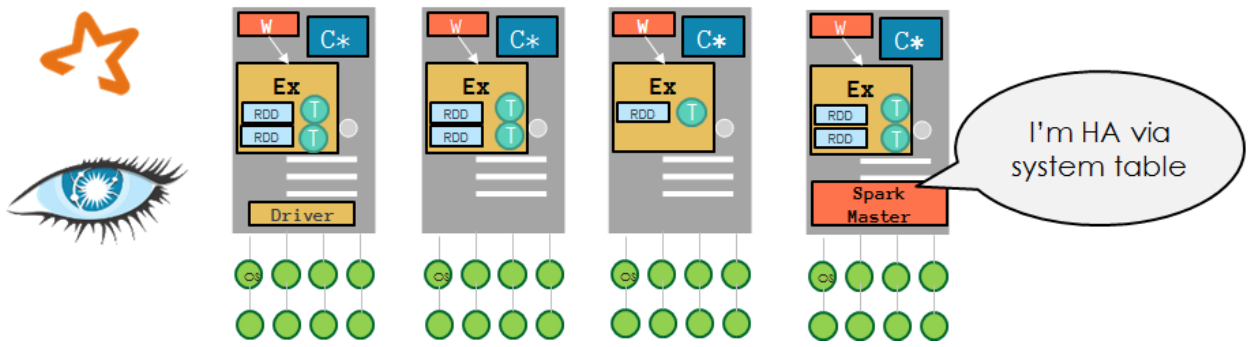
```
res10: scala.sys.process.ProcessBuilder =  ( [sudo, jps, -v] #| [grep,
DseSparkWorker] )

scala> res10.run
res11: scala.sys.process.Process =
scala.sys.process.ProcessImpl$PipedProcesses@345f4637

scala> 17922 DseSparkWorker -Dspark.cassandra.connection.host=10.0.1.28
-Dspark.kryoserializer.buffer.mb=10
-Dspark.cassandra.connection.host=10.0.1.28
-Dspark.kryoserializer.buffer.mb=10 -Dspark.akka.logLifecycleEvents=false
-Dspark.deploy.recoveryMode=CASSANDRA
-Djava.system.class.loader=com.datastax.bdp.loader.DseClientClassLoader
-Djava.library.path=::/usr/share/dse/hadoop/native/Linux-amd64-64/lib:/usr/sh
are/dse/hadoop/native/Linux-amd64-64/lib
-Dspark.cassandra.connection.factory=com.datastax.bdp.spark.DseCassandraConne
ctionFactory
-Dspark.cassandra.auth.conf.factory=com.datastax.bdp.spark.DseAuthConfFactory
-XX:MaxPermSize=128M -XX:MaxHeapFreeRatio=50 -XX:MinHeapFreeRatio=20
-Dspark.local.dir=/tmp/spark/worker
-Dlog4j.configuration=file:///etc/dse/spark/log4j-server.properties
-Dspark.log.file=/var/log/spark/worker.log -Xms16M -Xmx256M
-javaagent:/usr/share/dse/cassandra/lib/jamm-0.2.5.jar
<hit enter to return to scala shell>
```

Notice that the min (16 MB) and max (256MB) memory limits of the Worker JVM are much smaller since the Worker doesn't do the actual computational work (the Executor does).

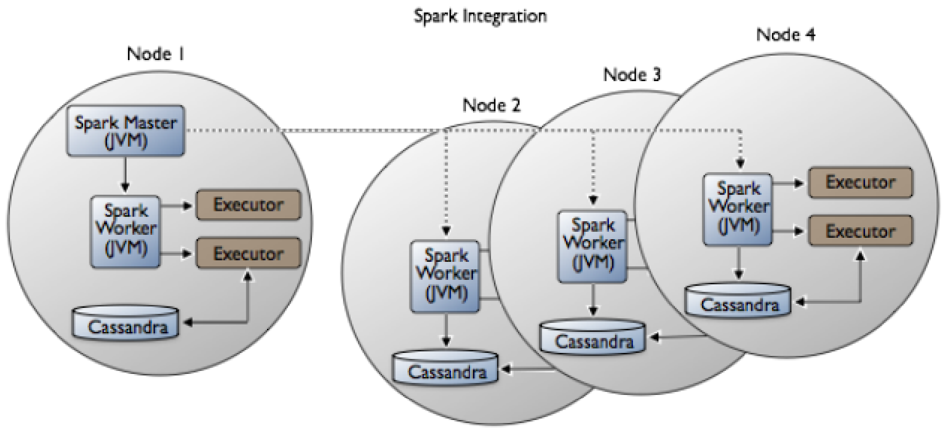
At a high level, every Spark application consists of a *driver program* that launches various parallel operations on your behalf into a cluster. The driver program contains your application's `main` function and defines distributed datasets on the cluster, then applies operations (transformation or action) to them.

Currently, the Spark Shell (aka Spark REPL) is the driver program itself and you can just type in operations you want to run.

Driver programs access Spark through a `SparkContext` object, which represents a connection to a computing cluster. In the shell, a `SparkContext` object is automatically created for you, as the variable called `sc`.



Here is another architecture diagram (as copied from the DSE 4.6 PDF):



Try printing out `sc` to see its type:

```
scala> sc
res12: shark.SharkContext = shark.SharkContext@562fbe44
```

Actually, notice that this is not a `SparkContext`, but rather a `SharkContext`. A `SharkContext` extends `SparkContext` and lets you create RDDs from SQL queries.

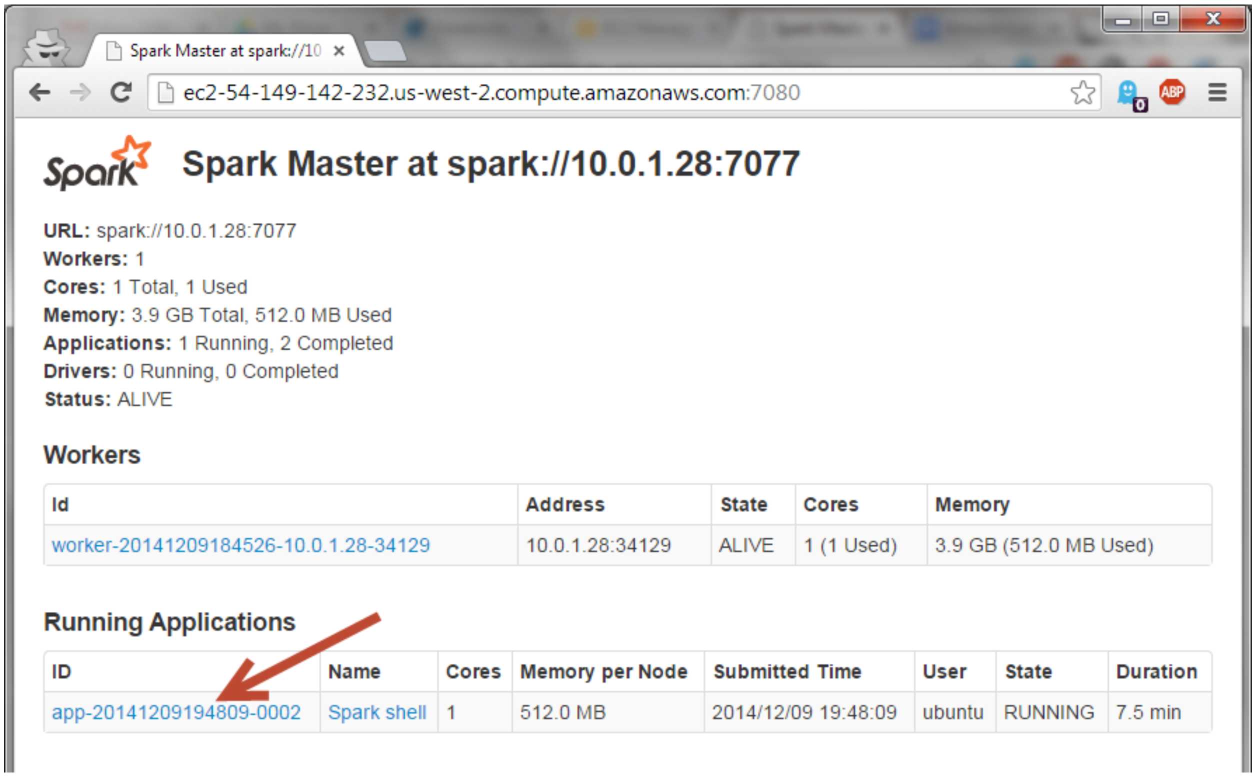
A context object is the main entry point for Spark functionality. A Context represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.

`As you learn the Spark API, you can try the :help command to double-check syntax and command specifications:`

```
scala> :help
All commands can be abbreviated, e.g. :he instead of :help.
Those marked with a * have more detailed help, e.g. :help imports.

:help [command]                                                  print this
summary or command-specific help
:history [num]                                                   show the
history (optional num is commands to show)
:h? <string>                                                     search the
history
:imports [name name ...]                                         show import
history, identifying sources of names
:implicits [-v]                                                  show the
implicits in scope
:javap <path|class>                                              disassemble
a file or class name
:load <path>                                                     load and
interpret a Scala file
:paste                                                           enter paste
mode: all input up to ctrl-D compiled together
:quit                                                            exit the
repl
:sh <command line>                                               run a shell
command (result is implicitly => List[String])
:silent
disable/enable automatic printing of results
:type [-v] <expr>                                                display the
type of an expression without evaluating it
:warnings                                                        show the
suppressed warnings from the most recent line which had any
:showSchema [all]|<keyspace_name>|<keyspace_name> <table_name>   show
Cassandra schema
```

Now that you've launched the Spark shell, you have actually instantiated a Spark Application called "Spark shell".

Go back to the Spark Master UI and notice the Running Application:
`http://<your EC2 instance’s public DNS hostname>:7080`


Spark Master at spark://10.0.1.28:7077

ec2-54-149-142-232.us-west-2.compute.amazonaws.com:7080

URL: spark://10.0.1.28:7077
Workers: 1
Cores: 1 Total, 1 Used
Memory: 3.9 GB Total, 512.0 MB Used
Applications: 1 Running, 2 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

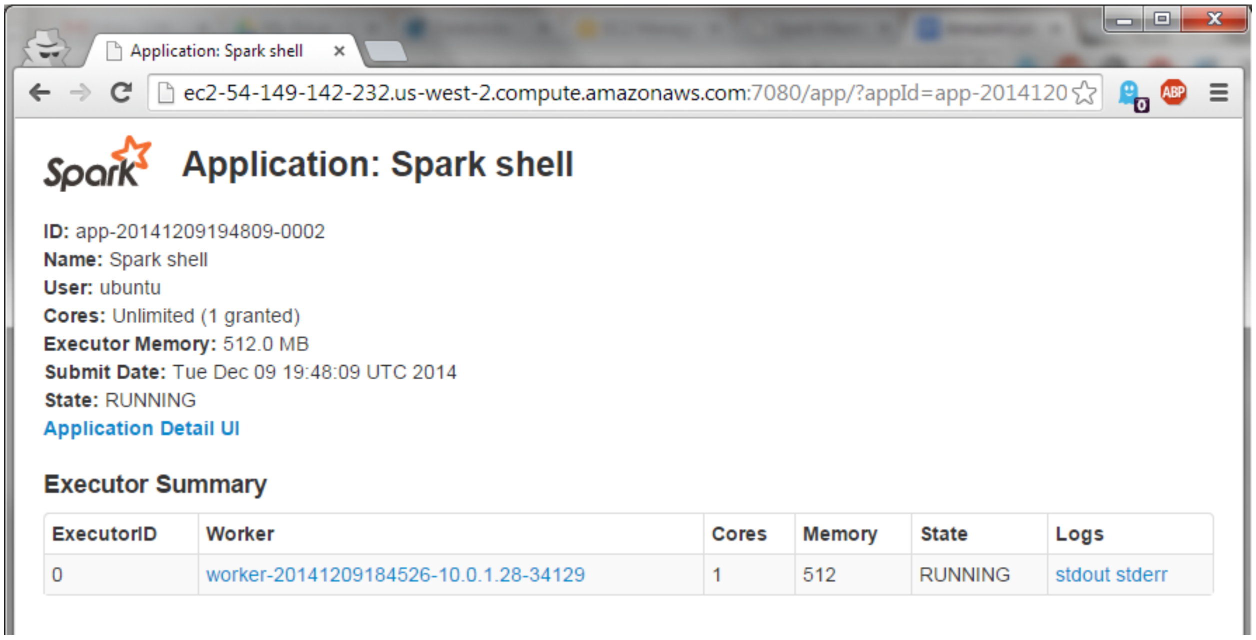
Workers

Id	Address	State	Cores	Memory
worker-20141209184526-10.0.1.28-34129	10.0.1.28:34129	ALIVE	1 (1 Used)	3.9 GB (512.0 MB Used)

Running Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20141209194809-0002	Spark shell	1	512.0 MB	2014/12/09 19:48:09	ubuntu	RUNNING	7.5 min

Click on the ID (red arrow) of the running application to see details about the application’s executors.



Application: Spark shell

ec2-54-149-142-232.us-west-2.compute.amazonaws.com:7080/app/?appId=app-2014120

Application: Spark shell

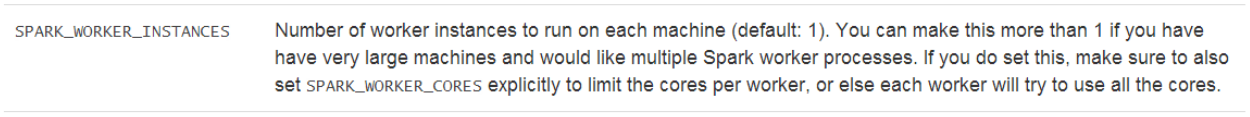
ID: app-20141209194809-0002
Name: Spark shell
User: ubuntu
Cores: Unlimited (1 granted)
Executor Memory: 512.0 MB
Submit Date: Tue Dec 09 19:48:09 UTC 2014
State: RUNNING
Application Detail UI

Executor Summary

ExecutorID	Worker	Cores	Memory	State	Logs
0	worker-20141209184526-10.0.1.28-34129	1	512	RUNNING	stdout stderr

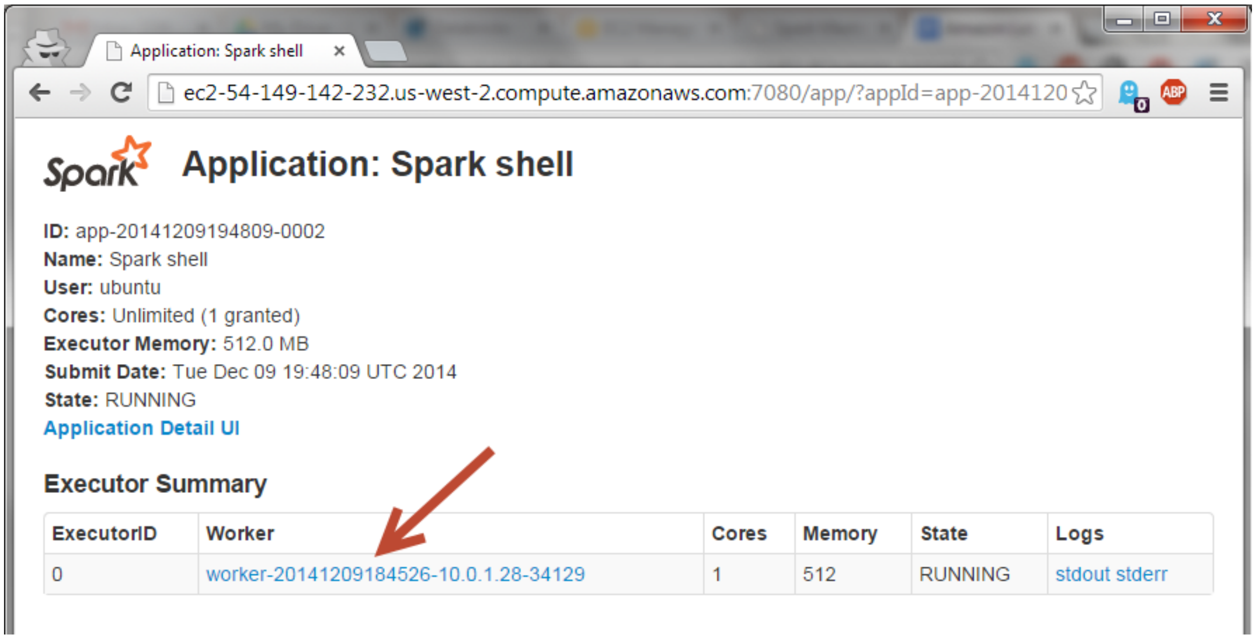
You are now at the '**Spark shell' Application** details page. Here you should see that the 'Spark shell' application has 1 executor at ExecutorID 0. It is using 1 core and 512 MB of memory. You can also see the worker that launched that executor. The worker itself doesn't actually do the work of the application. Instead, the worker's job is to launch an Executor, within which tasks will run. The tasks are the actual threads doing the work for the application. In a real production cluster, on this page, you would probably see many Executors running, with each Executor tied to one parent Worker. In a future release of Spark (perhaps 1.3 or 1.4), it will be possible for one Worker to launch multiple Executors on the machine it manages within the same Spark application, but this is not currently possible. Currently, one Worker will launch one Executor on its local machine for an application. If you have really large machines (lots of CPU, RAM), you may want to allow a machine to have 'n' Workers and 'n' Executors, where n > 1. This can be done by changing the `SPARK_WORKER_INSTANCES` environment variable, which defaults to 1… so only one Worker can run on each machine (with 1 Executor).

To read more about how to run multiple Workers + Executors on one machine, visit the following URL and search for "`SPARK_WORKER_INSTANCES`":
https://spark.apache.org/docs/1.1.0/spark-standalone.html#cluster-launch-scripts
(scroll down a bit to "Environment Variable")



`SPARK_WORKER_INSTANCES`	Number of worker instances to run on each machine (default: 1). You can make this more than 1 if you have have very large machines and would like multiple Spark worker processes. If you do set this, make sure to also set `SPARK_WORKER_CORES` explicitly to limit the cores per worker, or else each worker will try to use all the cores.

Click on the Worker to continue:


Application: Spark shell
ec2-54-149-142-232.us-west-2.compute.amazonaws.com:7080/app/?appId=app-2014120

Application: Spark shell

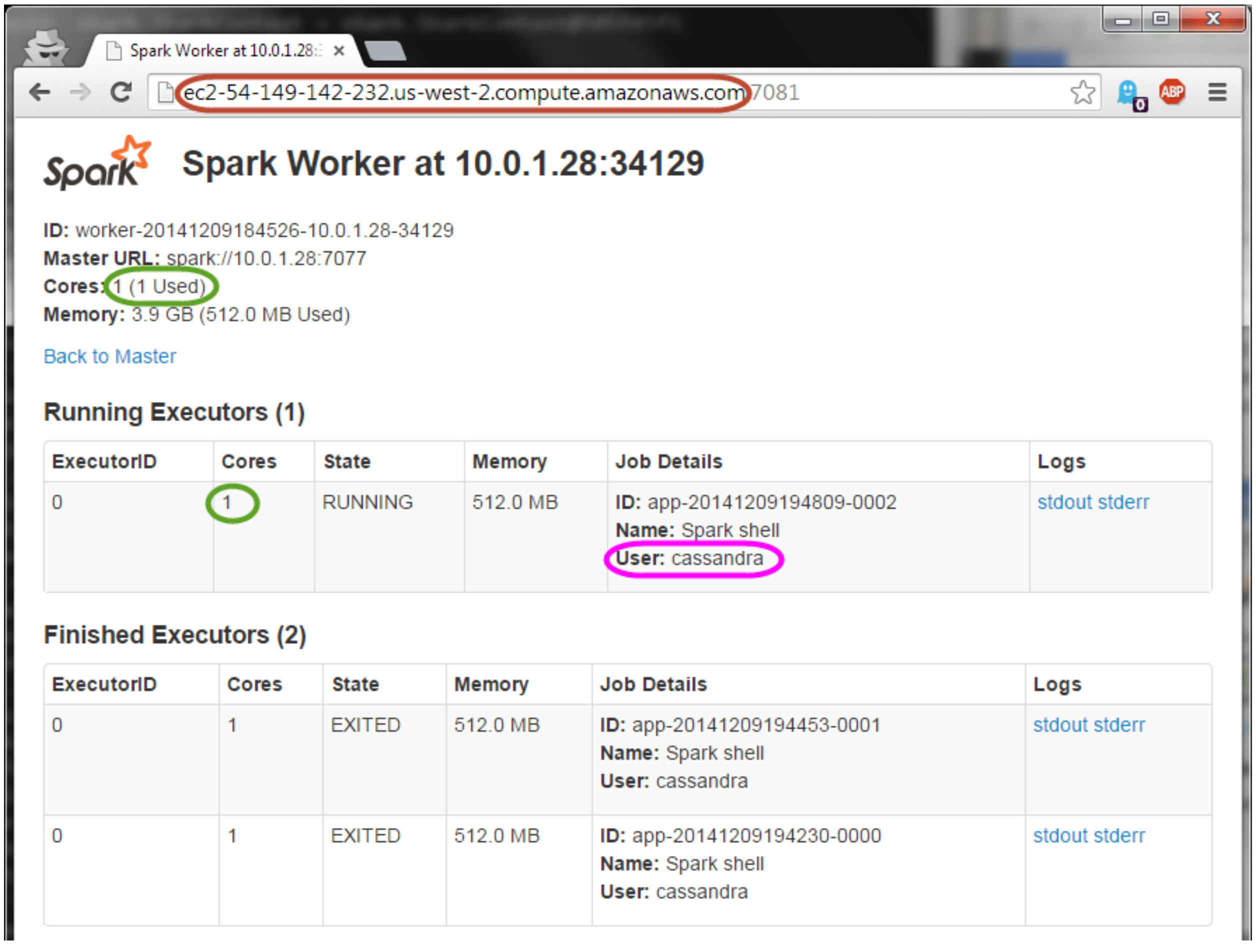
ID: app-20141209194809-0002
Name: Spark shell
User: ubuntu
Cores: Unlimited (1 granted)
Executor Memory: 512.0 MB
Submit Date: Tue Dec 09 19:48:09 UTC 2014
State: RUNNING
Application Detail UI

Executor Summary

ExecutorID	Worker	Cores	Memory	State	Logs
0	worker-20141209184526-10.0.1.28-34129	1	512	RUNNING	stdout stderr

Warning: Remember to swap out the private IP in the URL with the public DNS hostname of your EC2 instance!

Finally, you have arrived at the deepest layer: the worker page, which shows that 1 executor is running. If there were multiple workers, you would have to click on each worker to then see its child executors. The username that the executor is running as is 'cassandra'. This executor is the actual JVM that is the workhorse of the cluster. So on all the slave nodes, this cassandra user and its executor JVM be be doing the actual reads/writes to Cassandra (by invoking the local coordinator node to coordinate a read or write). This worker has access to allocate up to **1 core** to Executors and indeed the one Executor that is running is only using **1 core**. The Executor is also using 512 MB of memory out of a potential 4 GB (technicall 3.9 GB in this image) that the Worker can hand out. Technically, this one Executor could have launched with 4 GB of memory. But right now, as things stand, 3 more Spark applications can still be launched and this one Worker can launch 3 more Exectuors with 512 MB of memory each.


Spark Worker at 10.0.1.28:
ec2-54-149-142-232.us-west-2.compute.amazonaws.com:7081

Spark Worker at 10.0.1.28:34129

ID: worker-20141209184526-10.0.1.28-34129
Master URL: spark://10.0.1.28:7077
Cores: 1 (1 Used)
Memory: 3.9 GB (512.0 MB Used)

Back to Master

Running Executors (1)

ExecutorID	Cores	State	Memory	Job Details	Logs
0	1	RUNNING	512.0 MB	**ID:** app-20141209194809-0002 **Name:** Spark shell **User:** cassandra	stdout stderr

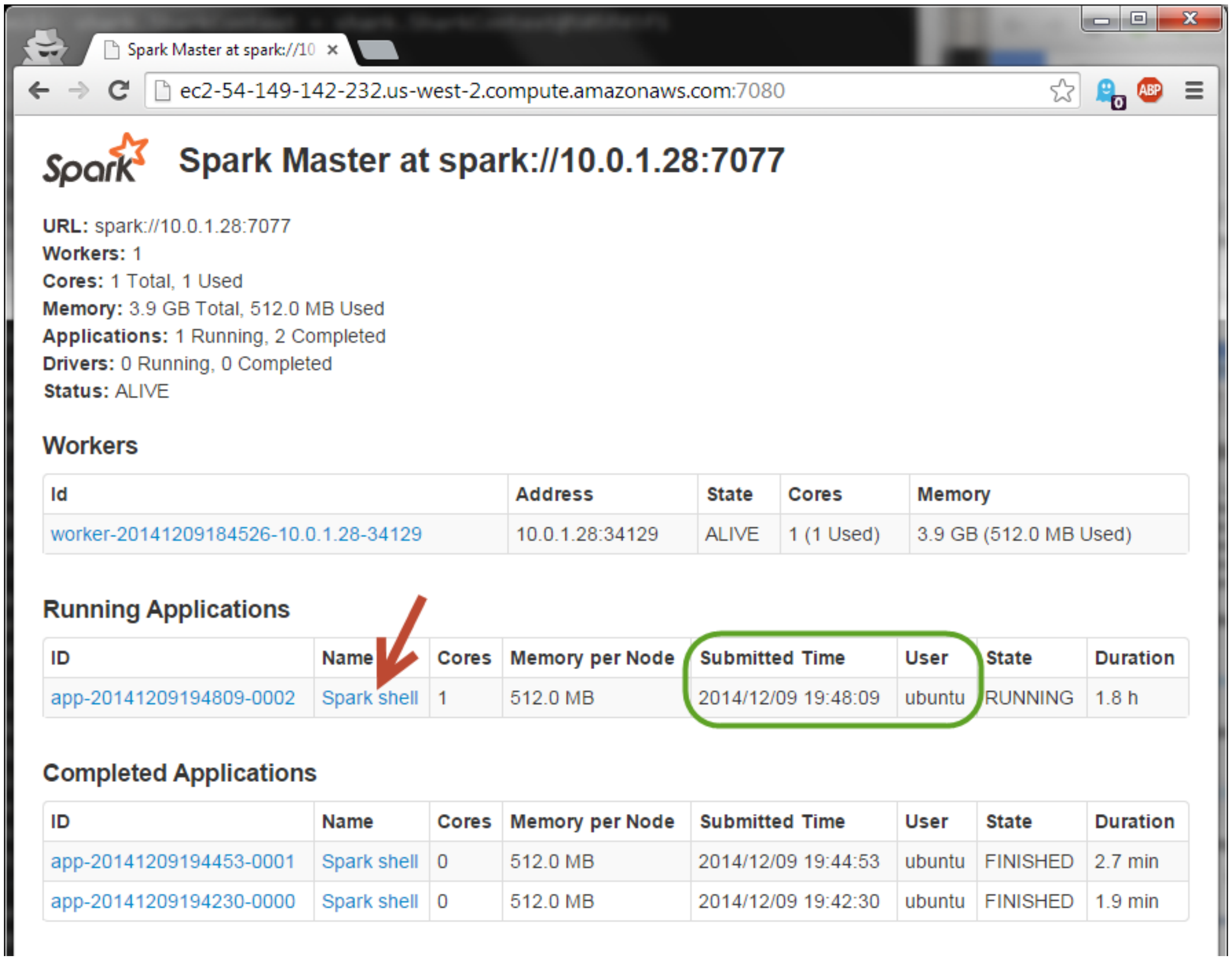
Finished Executors (2)

ExecutorID	Cores	State	Memory	Job Details	Logs
0	1	EXITED	512.0 MB	**ID:** app-20141209194453-0001 **Name:** Spark shell **User:** cassandra	stdout stderr
0	1	EXITED	512.0 MB	**ID:** app-20141209194230-0000 **Name:** Spark shell **User:** cassandra	stdout stderr

One last GUI related thing to show you. Before we continue, launch 'Spark shell' application's Stages page. To get to the Spark Stages page, first go back to the Spark Master UI:
`http://<your EC2 instance's public DNS hostname>:7080`

Notice under Running Applications, in the 3rd column from the right, the User is 'ubuntu', which means that the 'ubuntu' user submitted this application at the given Submitted TIme.

Then click on the link under the second column for the Name "Spark shell".


Spark Master at spark://10
ec2-54-149-142-232.us-west-2.compute.amazonaws.com:7080

Spark Master at spark://10.0.1.28:7077

URL: spark://10.0.1.28:7077
Workers: 1
Cores: 1 Total, 1 Used
Memory: 3.9 GB Total, 512.0 MB Used
Applications: 1 Running, 2 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Id	Address	State	Cores	Memory
worker-20141209184526-10.0.1.28-34129	10.0.1.28:34129	ALIVE	1 (1 Used)	3.9 GB (512.0 MB Used)

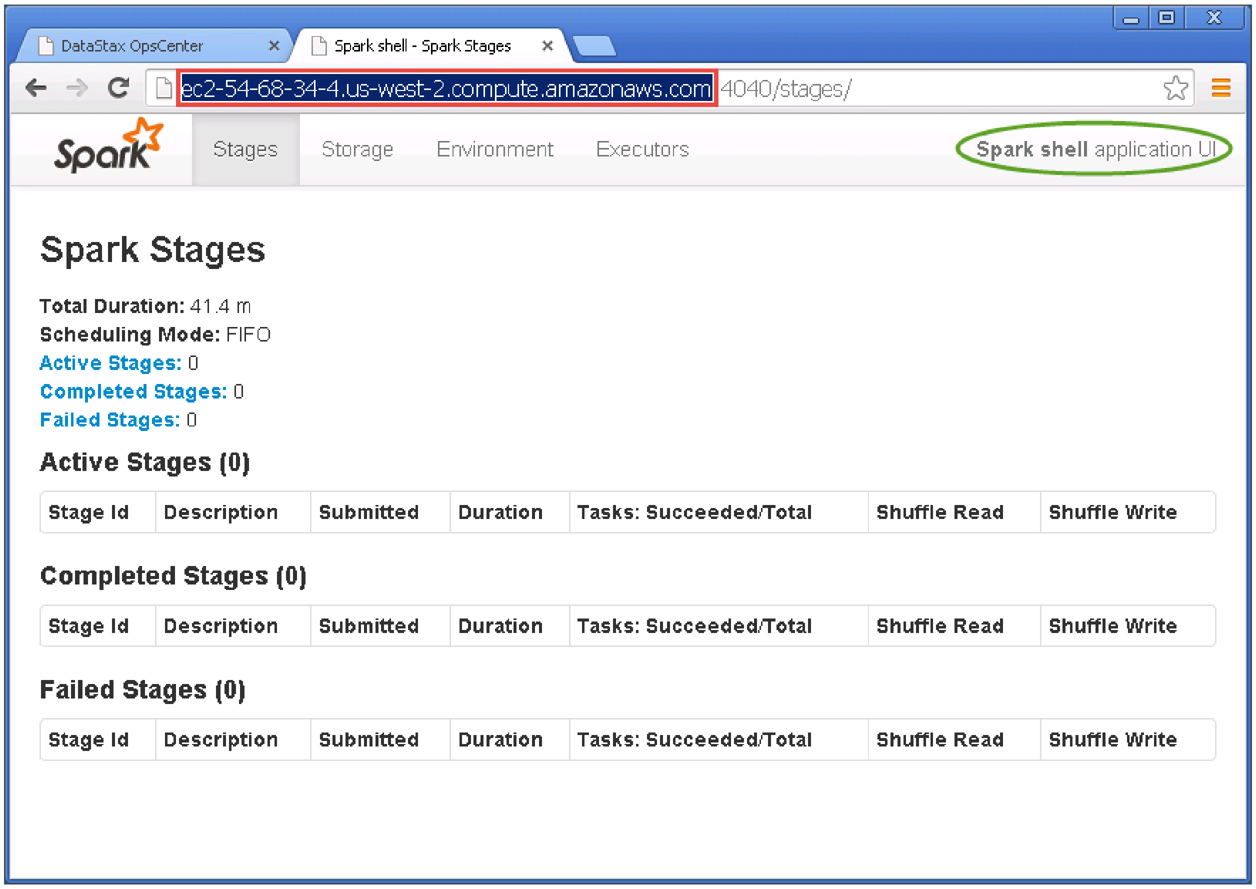
Running Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20141209194809-0002	Spark shell	1	512.0 MB	2014/12/09 19:48:09	ubuntu	RUNNING	1.8 h

Completed Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20141209194453-0001	Spark shell	0	512.0 MB	2014/12/09 19:44:53	ubuntu	FINISHED	2.7 min
app-20141209194230-0000	Spark shell	0	512.0 MB	2014/12/09 19:42:30	ubuntu	FINISHED	1.9 min

Warning: Once again swap out the internal hostname in the URL with the public DNS hostname of your EC2 instance.


DataStax OpsCenter
Spark shell - Spark Stages
ec2-54-68-34-4.us-west-2.compute.amazonaws.com:4040/stages/
Spark
Stages
Storage
Environment
Executors
Spark shell application UI

Spark Stages

Total Duration: 41.4 m
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 0
Failed Stages: 0

Active Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write

Completed Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write

Failed Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write

In the top right corner, notice that you are looking at the Spark Stages specifically for the "Spark shell" application.

Now that the Spark Stages page is loaded, keep it open as we will be refreshing it soon… as soon as we kick off a stage of computation.

Move back to the cmd line with the scala Spark Shell running. Let's create a simple app that reads `toy_file.txt` into memory (and then eventually we'll do some transformations to the data in memory). First we'll read the file and create an RDD called lines:

```
scala> val inputRDD = sc.textFile("/home/ubuntu/toy_file.txt")
inputRDD: org.apache.spark.rdd.RDD[String] = MappedRDD[1] at textFile at
<console>:25
```

Note that the command above didn't actually do anything. It didn't even read the file. Although you can define new RDDs on the fly, Spark only computes them in a lazy fashion, the first time they are used in an action. If the above file did not exist, you would not have gotten an error message.

Also, later in this lab you'll see how to read data from and write data to Cassandra tables. It's good to first play with Spark without the C* complexities… but we'll eventually graduate to reading/writing from/to C*!

Let's run the `count()` action against the lines RDD to see how many lines it is composed of:

```
scala> inputRDD.count()
org.apache.hadoop.mapred.InvalidInputException: Input path does not exist:
cfs://10.0.18.129/home/ubuntu/toy_file.txt
        at
org.apache.hadoop.mapred.FileInputFormat.listStatus(FileInputFormat.java:197)
        at
org.apache.hadoop.mapred.FileInputFormat.getSplits(FileInputFormat.java:208)
        at org.apache.spark.rdd.HadoopRDD.getPartitions(HadoopRDD.scala:140)
        at
org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:207)
<rest of error truncated>
```

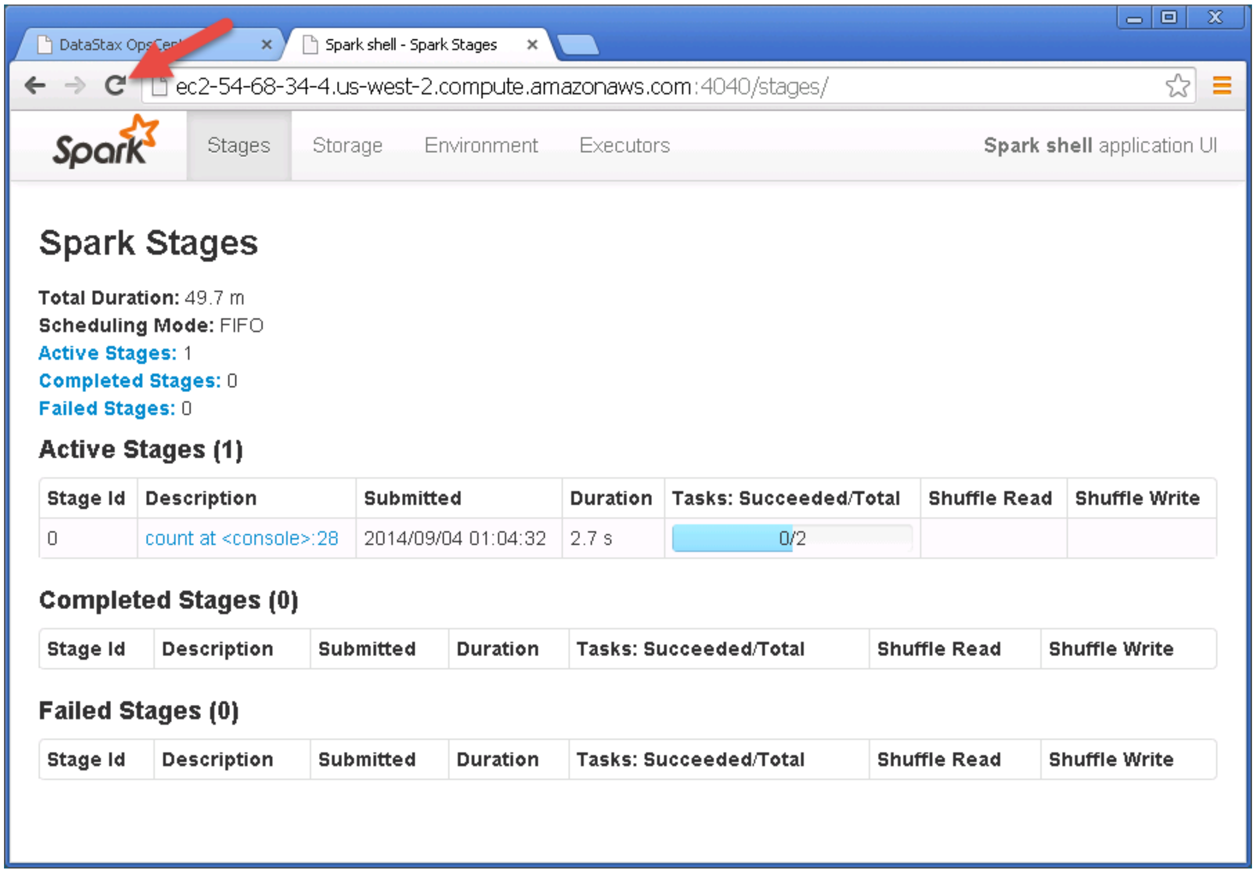
Hmm, we're getting an error. Input path does not exist. Notice that Spark is trying to read the data from: `cfs://10.0.18.129/home/ubuntu/toy_file.txt`

We actually want the data read from the local linux file system (ext4). In order to do this, we have to hard-define the full path like so, and then count the # of lines:

```
scala> val inputRDD = sc.textFile("file:///home/ubuntu/toy_file.txt")
inputRDD: org.apache.spark.rdd.RDD[String] = MappedRDD[5] at textFile at
<console>:25
```

```
scala> inputRDD.count()
res4: Long = 4
```

If you quickly switch back to the 'Spark Stages' web UI and refresh the page, you may catch the `.count()` action while it's actively executing:


DataStax OpsCenter
Spark shell - Spark Stages
ec2-54-68-34-4.us-west-2.compute.amazonaws.com:4040/stages/
Spark
Stages
Storage
Environment
Executors
Spark shell application UI

Spark Stages

Total Duration: 49.7 m
Scheduling Mode: FIFO
Active Stages: 1
Completed Stages: 0
Failed Stages: 0

Active Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write
0	count at <console>:28	2014/09/04 01:04:32	2.7 s	0/2		

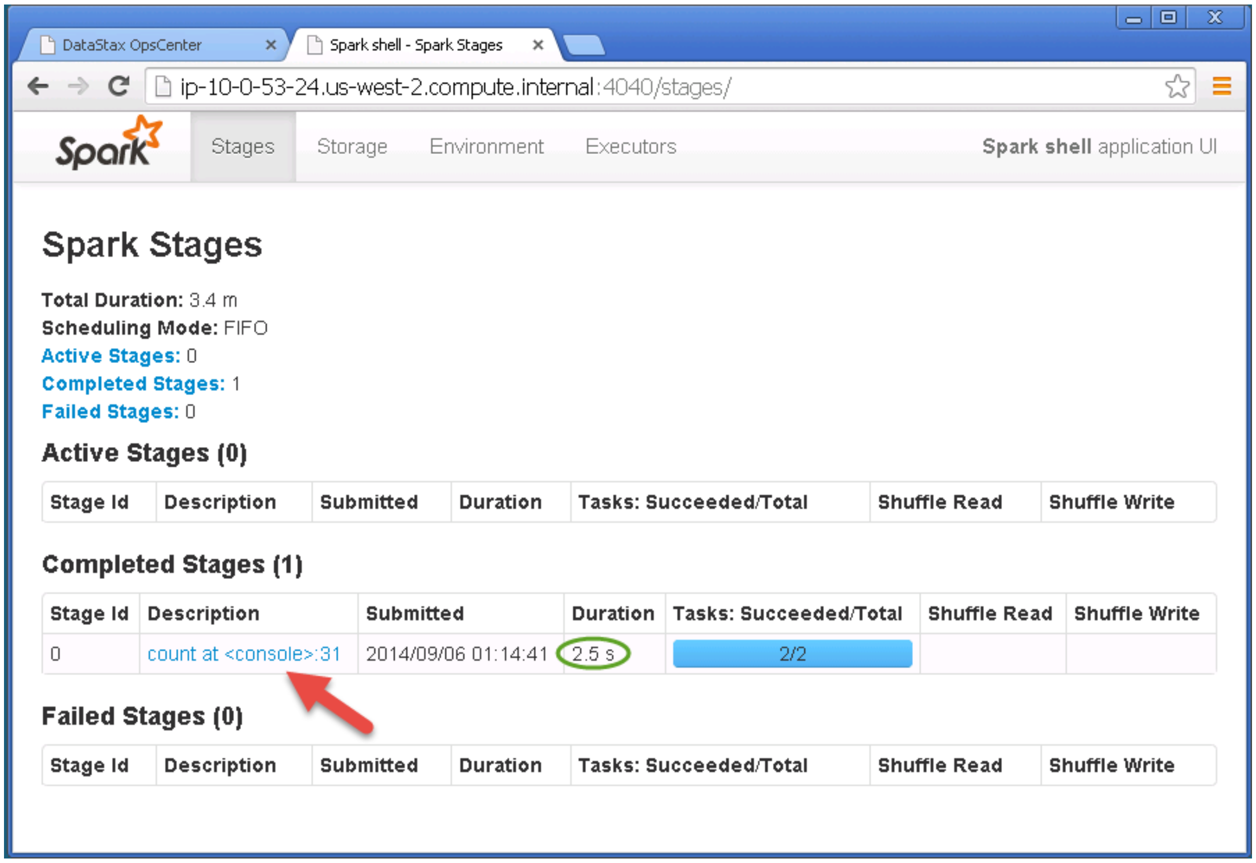
Completed Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write

Failed Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write

But more likely, you'll catch it after the Stage has already completed:


DataStax OpsCenter
Spark shell - Spark Stages
ip-10-0-53-24.us-west-2.compute.internal:4040/stages/
Stages
Storage
Environment
Executors
Spark shell application UI

Spark Stages

Total Duration: 3.4 m
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 1
Failed Stages: 0

Active Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write

Completed Stages (1)

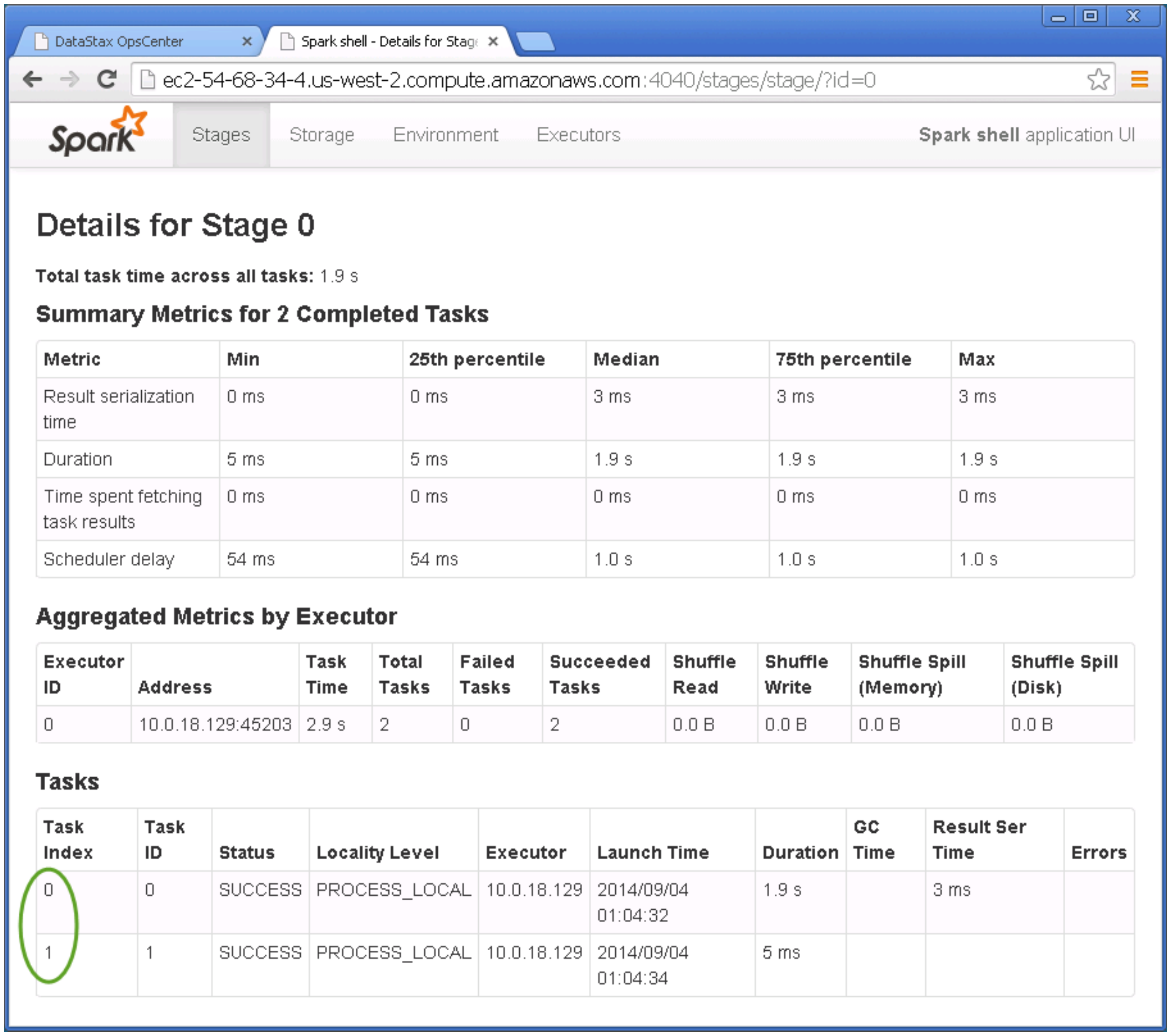
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write
0	count at <console>:31	2014/09/06 01:14:41	2.5 s	2/2		

Failed Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write

Notice here that the duration of the `.count()` stage was 2.5 seconds. Then click on the description "count at <console>:31" *(the actual # like 31 may be different in your environment)*

The Details page shows details for Stage 0 (the `.count()` action) of the 'Spark shell' application:


DataStax OpsCenter
Spark shell - Details for Stage
ec2-54-68-34-4.us-west-2.compute.amazonaws.com:4040/stages/stage/?id=0
Stages
Storage
Environment
Executors
Spark shell application UI

Details for Stage 0

Total task time across all tasks: 1.9 s

Summary Metrics for 2 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Result serialization time	0 ms	0 ms	3 ms	3 ms	3 ms
Duration	5 ms	5 ms	1.9 s	1.9 s	1.9 s
Time spent fetching task results	0 ms	0 ms	0 ms	0 ms	0 ms
Scheduler delay	54 ms	54 ms	1.0 s	1.0 s	1.0 s

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read	Shuffle Write	Shuffle Spill (Memory)	Shuffle Spill (Disk)
0	10.0.18.129:45203	2.9 s	2	0	2	0.0 B	0.0 B	0.0 B	0.0 B

Tasks

Task Index	Task ID	Status	Locality Level	Executor	Launch Time	Duration	GC Time	Result Ser Time	Errors
0	0	SUCCESS	PROCESS_LOCAL	10.0.18.129	2014/09/04 01:04:32	1.9 s		3 ms	
1	1	SUCCESS	PROCESS_LOCAL	10.0.18.129	2014/09/04 01:04:34	5 ms			

Notice the **2 tasks** at the bottom of the screenshot above. Why were there 2 tasks run in this stage? Is one of them to read the file and the other to do the line count? **No!**

There is actually a default parameter in Spark's `textFile()` function in `SparkContext.scala`, that sets `minPartitions` to 2. This parameter sets how many input splits to create. Here we are creating two input splits, but the entire `toy_file.text` is being read by just one input split and therefore being counted by just the first task. The 2nd task does nothing, therefore it finishes in ~5 ms… whereas the 1st task takes 1.9 secs to complete.
Take a look at the Side Note below for a deeper explanation of `textFile()`, or skip it if you don't care.

- - - - - - - - - - SIDE NOTE - - - - - - - - - -

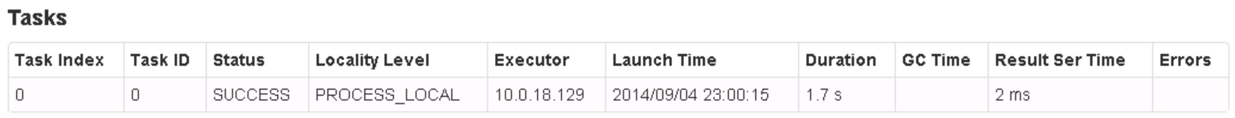
Warning: *Do **not** run the commands in this section! This is for educational purposes only. If you run these commands, it will throw off your lab environment from this document, which may make things confusing and inconsistent for you after this side note.*

View the scala source code for `textFile()` here:
https://github.com/apache/spark/blob/2784822e4c63083a647cc2d6c7089065ef3b947d/core/src/main/scala/org/apache/spark/SparkContext.scala#L451

Let's say you want to hard set the `minPartitions` for `textFile()` to 1, so that actions like `.count()` run against the file only need 1 task. You can pass in an extra argument to the `textFile()` function like this:

```
scala> val inputRDD = sc.textFile("file:///home/ubuntu/toy_file.txt", 1)
inputRDD: org.apache.spark.rdd.RDD[String] = MappedRDD[1] at textFile at
<console>:25

scala> inputRDD.count()
res2: Long = 4
```

Now `count()` will result in just one task:



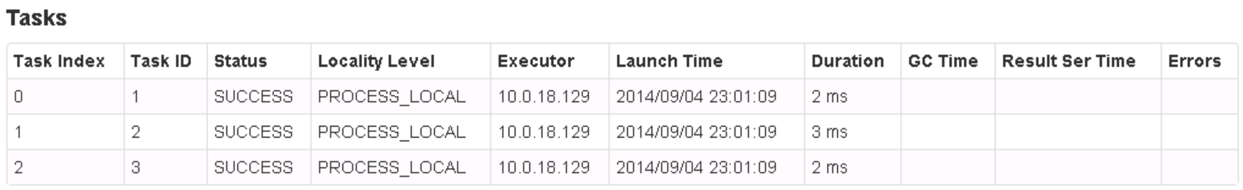
Tasks

Task Index	Task ID	Status	Locality Level	Executor	Launch Time	Duration	GC Time	Result Ser Time	Errors
0	0	SUCCESS	PROCESS_LOCAL	10.0.18.129	2014/09/04 23:00:15	1.7 s		2 ms	

Or if you pass in 3 for `minPartitions`:

```
scala> val inputRDD3 = sc.textFile("file:///home/ubuntu/toy_file.txt", 3)
inputRDD3: org.apache.spark.rdd.RDD[String] = MappedRDD[3] at textFile at
<console>:25

scala> inputRDD3.count()
res3: Long = 4
```

Now `count()` will result in three tasks:



Tasks

Task Index	Task ID	Status	Locality Level	Executor	Launch Time	Duration	GC Time	Result Ser Time	Errors
0	1	SUCCESS	PROCESS_LOCAL	10.0.18.129	2014/09/04 23:01:09	2 ms			
1	2	SUCCESS	PROCESS_LOCAL	10.0.18.129	2014/09/04 23:01:09	3 ms			
2	3	SUCCESS	PROCESS_LOCAL	10.0.18.129	2014/09/04 23:01:09	2 ms			

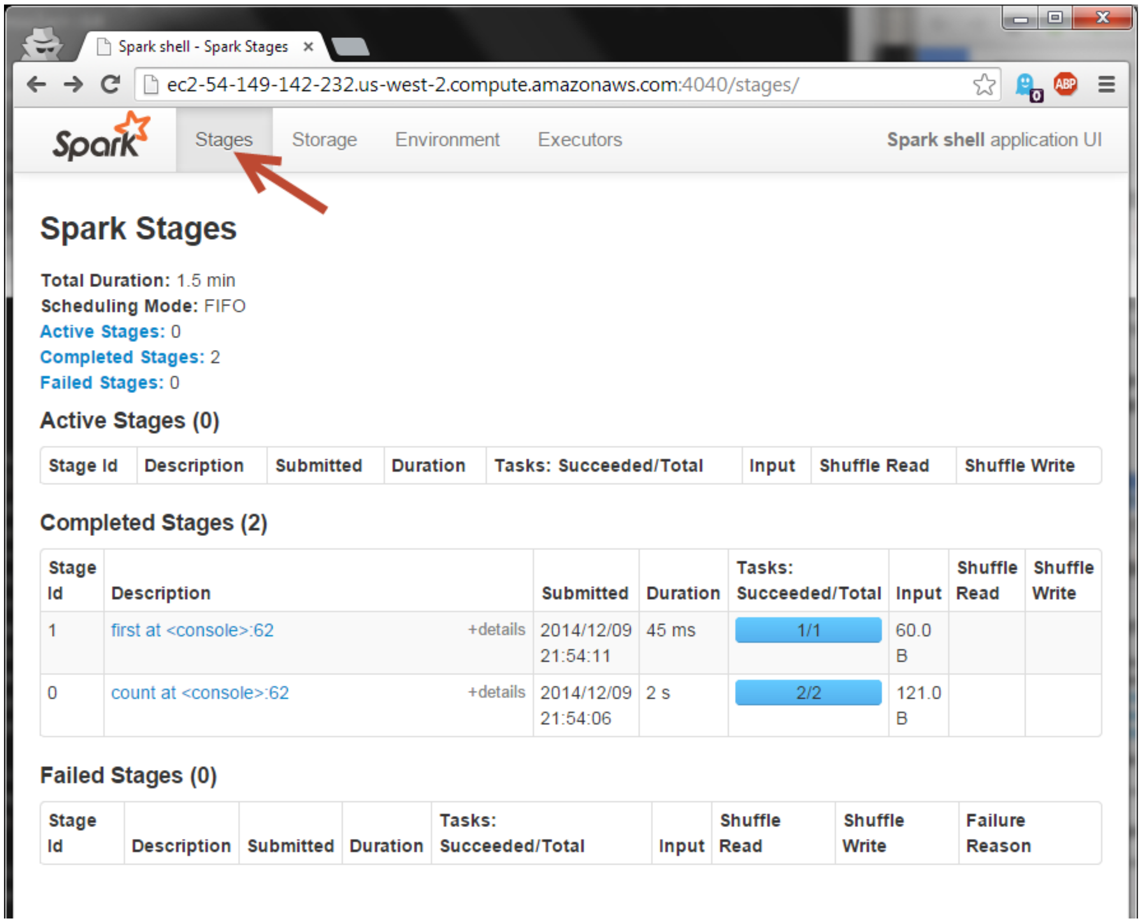
- - - - - - - - - - - /END SIDE NOTE - - - - - - - - - - -

Moving on… switch back to the scala Spark shell and try to retrieve just the first line in this RDD:

```
scala> inputRDD.first()
res5: String = This is the first line.
```

In the above `first()` action, Spark only scans the file until it finds the first matching line; it doesn't read the whole file.

If you go back to the Stages web page (by clicking on 'Stages' in the UI), you will see that the `.first()` action launched a new Stage.


Spark shell - Spark Stages
ec2-54-149-142-232.us-west-2.compute.amazonaws.com:4040/stages/
Spark
Stages
Storage
Environment
Executors
Spark shell application UI

Spark Stages

Total Duration: 1.5 min
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 2
Failed Stages: 0

Active Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
1	first at <console>:62 +details	2014/12/09 21:54:11	45 ms	1/1	60.0 B		
0	count at <console>:62 +details	2014/12/09 21:54:06	2 s	2/2	121.0 B		

Failed Stages (0)

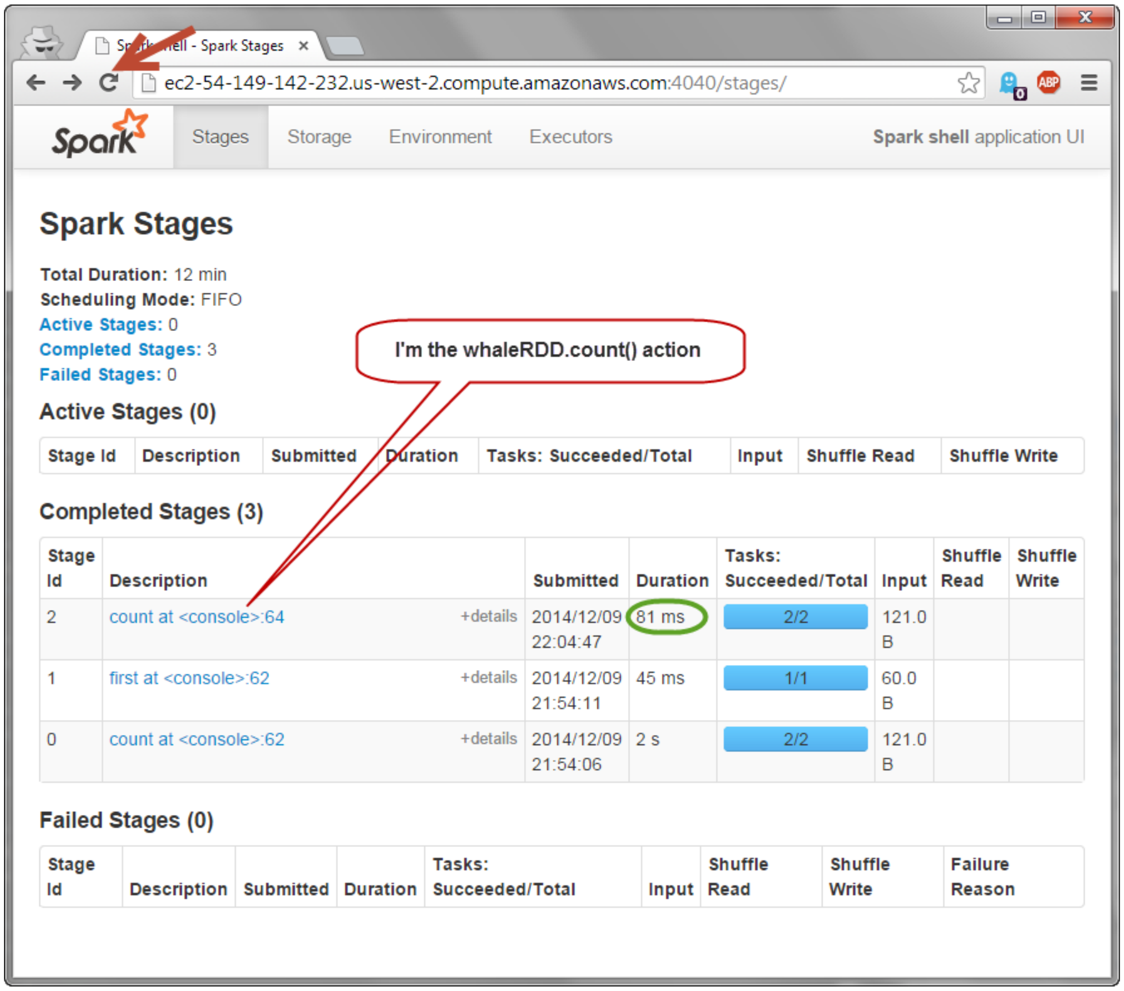
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write	Failure Reason

We can use the `filter` transformation to yank out just the 2 lines that contain the word "WHALE" in `toy_file.txt`:

```
scala> val whaleRDD = inputRDD.filter(line => line.contains("WHALE"))
whaleRDD: org.apache.spark.rdd.RDD[String] = FilteredRDD[4] at filter at
<console>:27
```

```
scala> whaleRDD.count()
res6: Long = 2
```

Another refresh of the Spark Stages page will show you Stage id #2, which corresponds to the `whaleRDD.count()` action. Also, notice that this stage id #2 completed in just 53 ms. The `toy_file.txt` was probably cached in the OS's memory buffers, so it re-read very quickly. From a Spark perspective, this 2nd stage had to re-read the file all over again since we have not persisted any RDDs to memory. We will explore this idea later in the lab after we read data from a C* table.


Spark Shell - Spark Stages
ec2-54-149-142-232.us-west-2.compute.amazonaws.com:4040/stages/
Spark
Stages
Storage
Environment
Executors
Spark shell application UI
I'm the whaleRDD.count() action

Spark Stages

Total Duration: 12 min
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 3
Failed Stages: 0

Active Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write

Completed Stages (3)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
2	count at <console>:64 +details	2014/12/09 22:04:47	81 ms	2/2	121.0 B		
1	first at <console>:62 +details	2014/12/09 21:54:11	45 ms	1/1	60.0 B		
0	count at <console>:62 +details	2014/12/09 21:54:06	2 s	2/2	121.0 B		

Failed Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write	Failure Reason

Verify that the filter transformation we ran to create the `whaleRDD` actually has only the WHALE lines by printing the first line in the RDD:

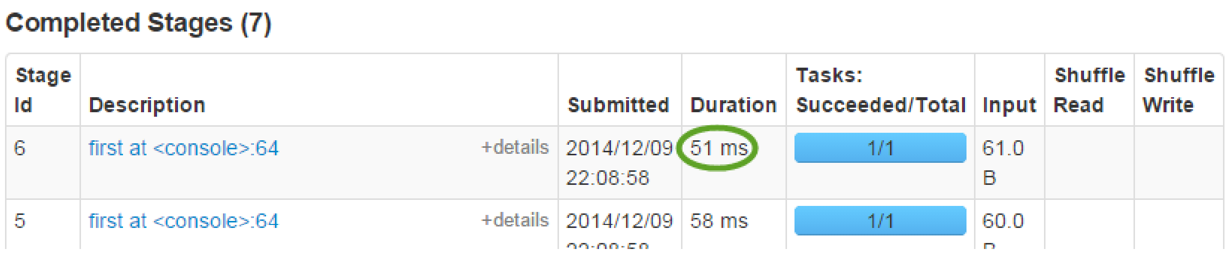
```
scala> whaleRDD.first()
res7: String = Look, the second line has a WHALE on it.
```

It is important to know that the filter transformation does not mutate the existing `inputRDD`. Instead, it returns a pointer to an entirely new RDD. `inputRDD` can still be re-used later in the program to search for other words. Actually, let's do that now.

Use inputRDD again to search for lines with the word "does" in them:

```
scala> val doesRDD = inputRDD.filter(line => line.contains("does"))
doesRDD: org.apache.spark.rdd.RDD[String] = FilteredRDD[8] at filter at
<console>:27

scala> doesRDD.count()
res12: Long = 2

scala> doesRDD.first()
res13: String = So, does the third: WHALE
```

If you refresh the 'Spark Stages' web page, you'll see that the `doesRDD.first()` action took approximately 51 ms to run:



Completed Stages (7)

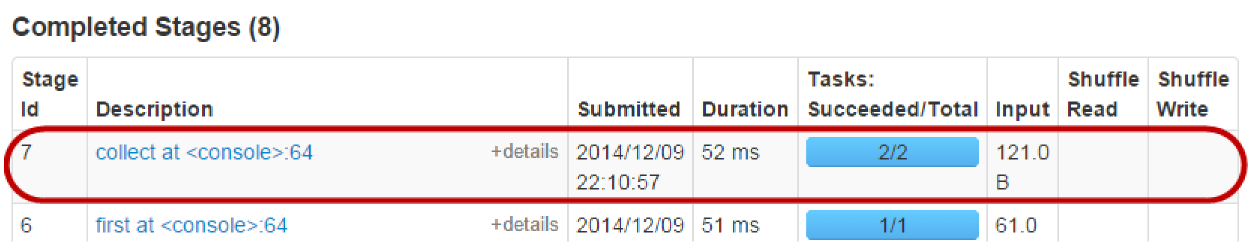
Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
6	first at <console>:64	+details	2014/12/09 22:08:58	51 ms	1/1	61.0 B		
5	first at <console>:64	+details	2014/12/09	58 ms	1/1	60.0		

What if we wanted to see both of the lines in the `doesRDD` that has the word 'does'? Use the `collect()` action:

```
scala> doesRDD.collect()
res14: Array[String] = Array(So, does the third: WHALE, But the 4th line does
not :-()
```

All RDDs have a `collect()` function to retrieve the entire RDD. This is useful if your program filters RDDs down to a very small size and you'd like to deal with it locally. Be careful though, as your entire dataset must fit in memory on a single machine to use `collect()`. So, `collect()` should not be used on large datasets. This function will send all the data from the RDD to the driver, so the driver must be able to handle the amount of data sent to it on one machine.

Another Stage launches to compute the action `doesRDD.collect()`:



Completed Stages (8)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
7	collect at <console>:64	+details	2014/12/09 22:10:57	52 ms	2/2	121.0 B		
6	first at <console>:64	+details	2014/12/09	51 ms	1/1	61.0		

And here are the 2 lines in the `whaleRDD`:

```
scala> whaleRDD.collect()
res18: Array[String] = Array(Look, the second line has a WHALE on it., So,
does the third: WHALE)
```

Notice that the 3rd line in the file appears in both filtered RDDs. Now, let's use another transformation, `union()`, to print out the number of lines that contained either "WHALE" or "does". So, the result of the union() should be 4 lines. The `union()` function will return an RDD consisting of data from both sources. Keep in mind that unlike the mathematical `union()`, if there are duplicates in the input RDDs, the result of SPark's `union()` function will contain duplicates (which we can fix if desired with `distinct()`).

```
scala> val bothRDD = whaleRDD.union(doesRDD)
bothRDD: org.apache.spark.rdd.RDD[String] = UnionRDD[9] at union at
<console>:31

scala> bothRDD.count()
res20: Long = 4

scala> bothRDD.collect()
res19: Array[String] = Array(Look, the second line has a WHALE on it., So,
does the third: WHALE, So, does the third: WHALE, But the 4th line does not
:-()
```

We can run a further transformation called `distinct()` on `bothRDD` to get rid of that duplicate third line:

```
scala> val distinctbothRDD = bothRDD.distinct()
distinctbothRDD: org.apache.spark.rdd.RDD[String] = MappedRDD[13] at distinct
at <console>:33
```

And run the `collect()` action to see if it worked:

```
scala> distinctbothRDD.collect()
res5: Array[String] = Array(Look, the second line has a WHALE on it., So,
does the third: WHALE, But the 4th line does not :-()
```

Note that Spark 1.0 introduces an Intersection transformation, which let's you keep just the intersection of the data between the two RDDs, `whaleRDD` and `doesRDD` (so just the 3rd line with `So, does the third: WHALE`)

Finally, let's write the data in `bothRDD` back to the linux file system:

```
scala> bothRDD.saveAsTextFile("file:///home/ubuntu/bothRDD")
14/12/09 22:13:18 WARN TaskSetManager: Lost task 0.0 in stage 13.0 (TID 30,
10.0.1.28): java.io.IOException: Mkdirs failed to create
file:/home/ubuntu/bothRDD/_temporary/_attempt_201412092213_0013_m_000000_30

org.apache.hadoop.mapred.FileOutputCommitter.getWorkPath(FileOutputCommitter.java:257
)

org.apache.hadoop.mapred.FileOutputFormat.getTaskOutputPath(FileOutputFormat.java:244
)

org.apache.hadoop.mapred.TextOutputFormat.getRecordWriter(TextOutputFormat.java:116)
        org.apache.spark.SparkHadoopWriter.open(SparkHadoopWriter.scala:89)
```

Why did this error happen? It's a permissions issue. Remember that the Executor is running under the Cassandra user, but here we tried to write out the RDD to the /home/ubuntu user's directory.

Notice that since the stack trace starts at `org.apache.hadoop.mapred.FileOutputCommitter`, Spark is actually using Hadoop code to write out the results.

The error comes from this code:

https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/FileOutputCommitter.html

One thing we can try to troubleshoot the above error is check whether the `/home/ubuntu/bothRDD/` folder has actually been created and whether it has sufficient permissions.

First person to resolve the above file write issue wins a Databricks USB stick and a Spark sticker!

If you're running behind in the lab, just ignore the above write error and continue (you don't need to have written out the RDD for the rest of the lab). However, we will do a little bit of troubleshooting around this issue below to get you started…

Here again is the `:sh` command in the Spark shell that lets you submit linux command line commands:

```
scala> :sh /bin/ls -alh /home/ubuntu
res24: scala.tools.nsc.interpreter.ProcessResult = `/bin/ls -alh
/home/ubuntu` (16 lines, exit 0)
```

Now, print the output of result 24 (the result # might be different in your shell!):

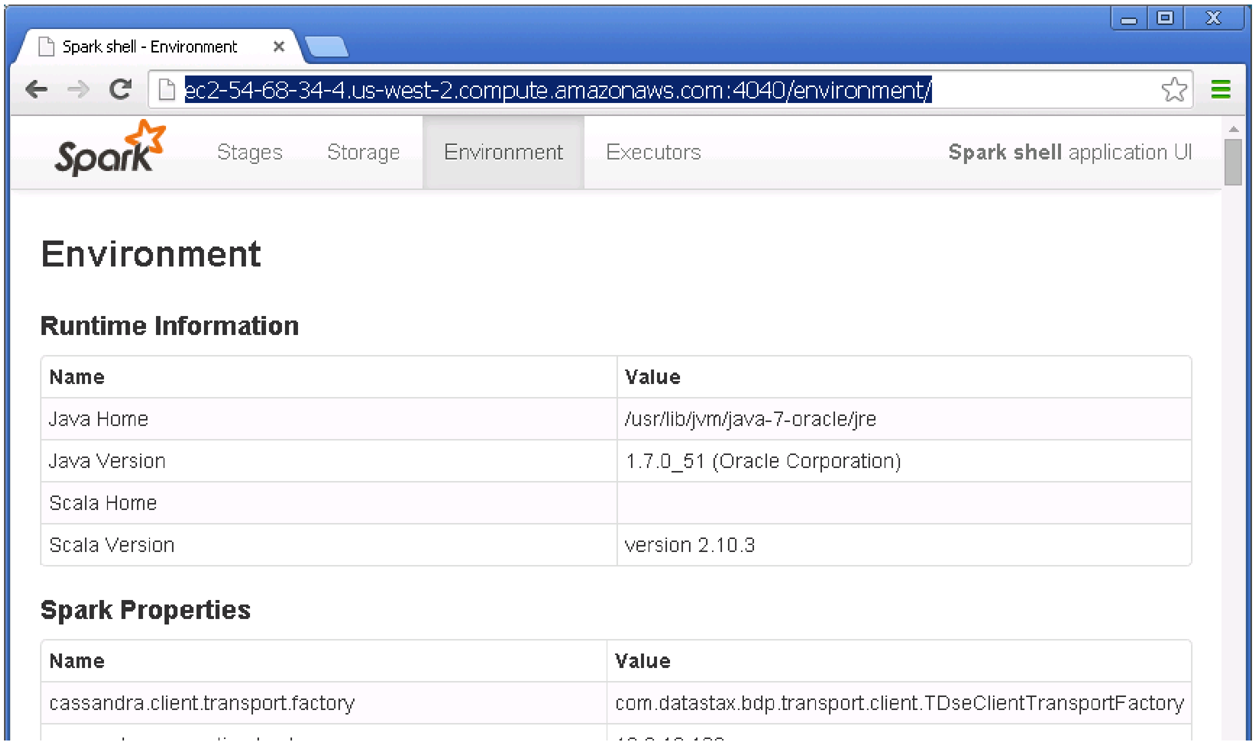
```
scala> res24.show
total 56K
drwxr-xr-x 8 ubuntu ubuntu 4.0K Sep  2 21:52 .
drwxr-xr-x 3 root   root   4.0K Feb 27  2014 ..
-rw-r--r-- 1 ubuntu ubuntu  220 Apr  3  2012 .bash_logout
-rw-r--r-- 1 ubuntu ubuntu 3.5K Apr  3  2012 .bashrc
drwxrwxr-x 3 ubuntu ubuntu 4.0K Sep  2 21:52 bothRDD
drwx------ 2 ubuntu ubuntu 4.0K Mar 24 23:08 .cache
drwxrwxr-x 2 ubuntu ubuntu 4.0K Sep  2 17:51 .cassandra
drwxr-xr-x 6 ubuntu ubuntu 4.0K Sep  2 17:51 datastax_ami
-rw-rw-r-- 1 ubuntu ubuntu   47 Mar 24 23:21 .gitconfig
drwxr-xr-x 3 ubuntu ubuntu 4.0K Mar 24 23:30 .m2
-rw-r--r-- 1 ubuntu ubuntu  809 Mar 24 23:32 .profile
-rw-rw-r-- 1 ubuntu ubuntu 2.5K Sep  2 22:19 .scala_history
drwx------ 2 ubuntu ubuntu 4.0K Sep  2 17:48 .ssh
-rw-r--r-- 1 ubuntu ubuntu    0 Mar 24 23:14 .sudo_as_admin_successful
-rw-rw-r-- 1 ubuntu ubuntu  121 Sep  2 19:07 toy_file.txt
```

In the highlighted line above, you can see that the `bothRDD` directory has full read, write and execute permissions for the 'ubuntu' user.

But, how do we check what user the Spark application is running as? Well, we saw earlier in the lab via the Spark UIs that the Spark REPL (scala shell or driver) is running as the 'ubuntu' user and the Executor was running under 'cassandra'. But there's another way to check who the Executor is running as...

Let's use the Spark web UI to find out.

Open a Chrome or FireFox browser window and go to:
`http://<your EC2 instance's public DNS hostname>:4040/environment/`


Spark shell - Environment
ec2-54-68-34-4.us-west-2.compute.amazonaws.com:4040/environment/
Stages
Storage
Environment
Executors
Spark shell application UI

Environment

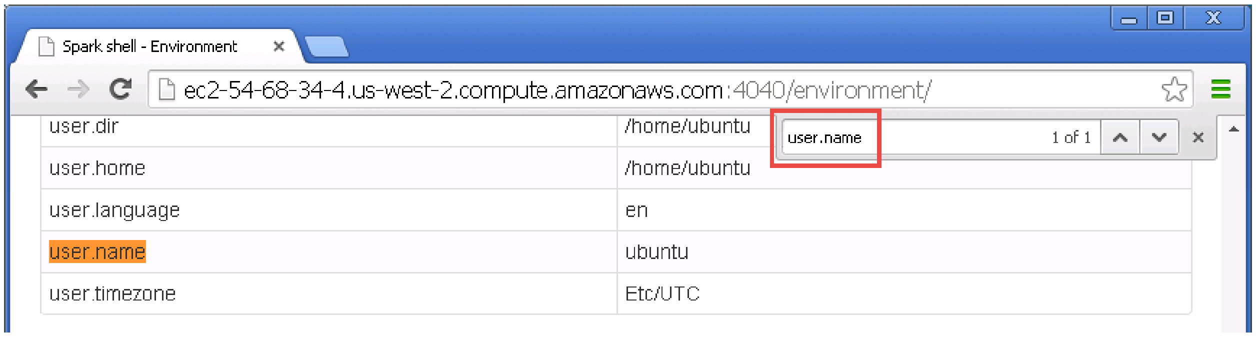
Runtime Information

Name	Value
Java Home	/usr/lib/jvm/java-7-oracle/jre
Java Version	1.7.0_51 (Oracle Corporation)
Scala Home	
Scala Version	version 2.10.3

Spark Properties

Name	Value
cassandra.client.transport.factory	com.datastax.bdp.transport.client.TDseClientTransportFactory

Hit CTRL + F in the browser to search for 'user.name' on the page:


Spark shell - Environment
ec2-54-68-34-4.us-west-2.compute.amazonaws.com:4040/environment/
user.name
1 of 1

user.dir	/home/ubuntu
user.home	/home/ubuntu
user.language	en
user.name	ubuntu
user.timezone	Etc/UTC

You can see that the Spark shell (aka Spark application) was launched by the user 'ubuntu'. However, from earlier, we know that the actual Executor started under the user 'cassandra', so it is the Cassandra user who needs to have write access to `/home/ubuntu/bothRDD` for our text file write to be successful.

Go ahead and familiarize yourself with the Spark environment variables and then see if you can fix the file write issue!

May I suggest just writing the file to a different location where you know the 'cassandra' user definitely has write permissions… like so:

```
scala> bothRDD.saveAsTextFile("file:///tmp/bothRDD")
```

```
14/12/09 22:17:13 WARN TaskSetManager: Lost task 0.0 in stage 14.0 (TID 37,
10.0.1.28): java.io.IOException: Mkdirs failed to create
file:/tmp/bothRDD/_temporary/_attempt_201412092217_0014_m_000000_37

org.apache.hadoop.mapred.FileOutputCommitter.getWorkPath(FileOutputCommitter.
java:257)

org.apache.hadoop.mapred.FileOutputFormat.getTaskOutputPath(FileOutputFormat.
java:244)
```

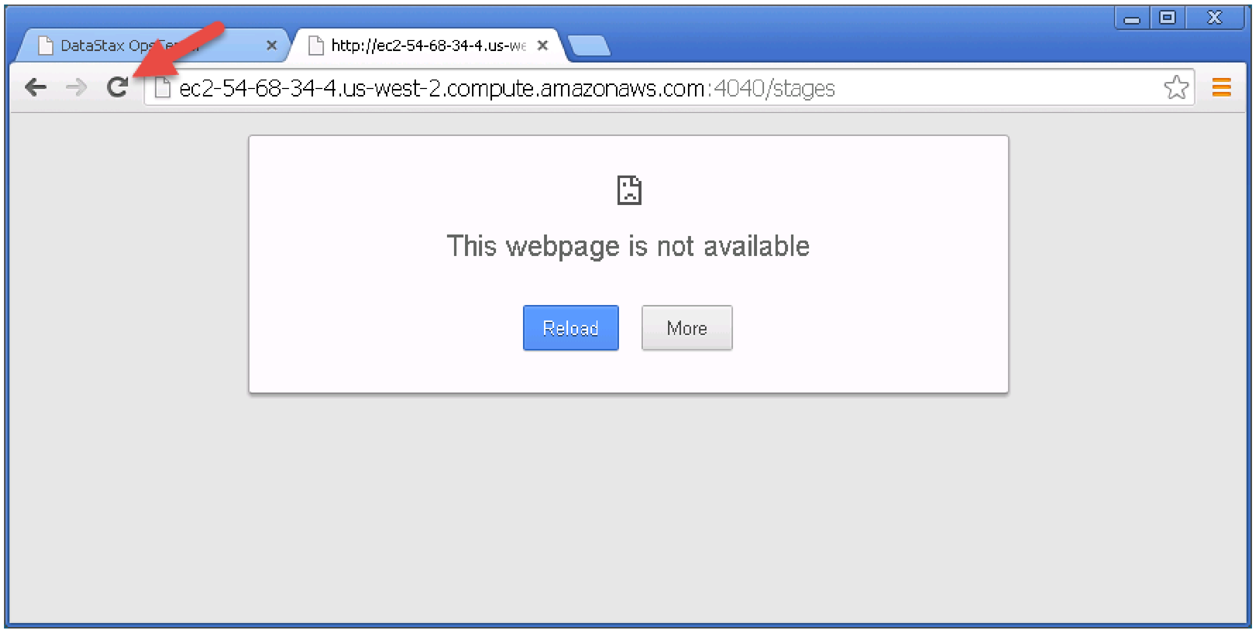
But that also fails. What do you think is happening?

Let the instructor know if you figure it out, or just skip the troubleshooting and continue onward...

Stop the spark application and exit the shell:

```
scala> :q
Stopping spark context.
```

Now the Spark Stages page will stop loading since we've terminated the 'Spark shell' application:

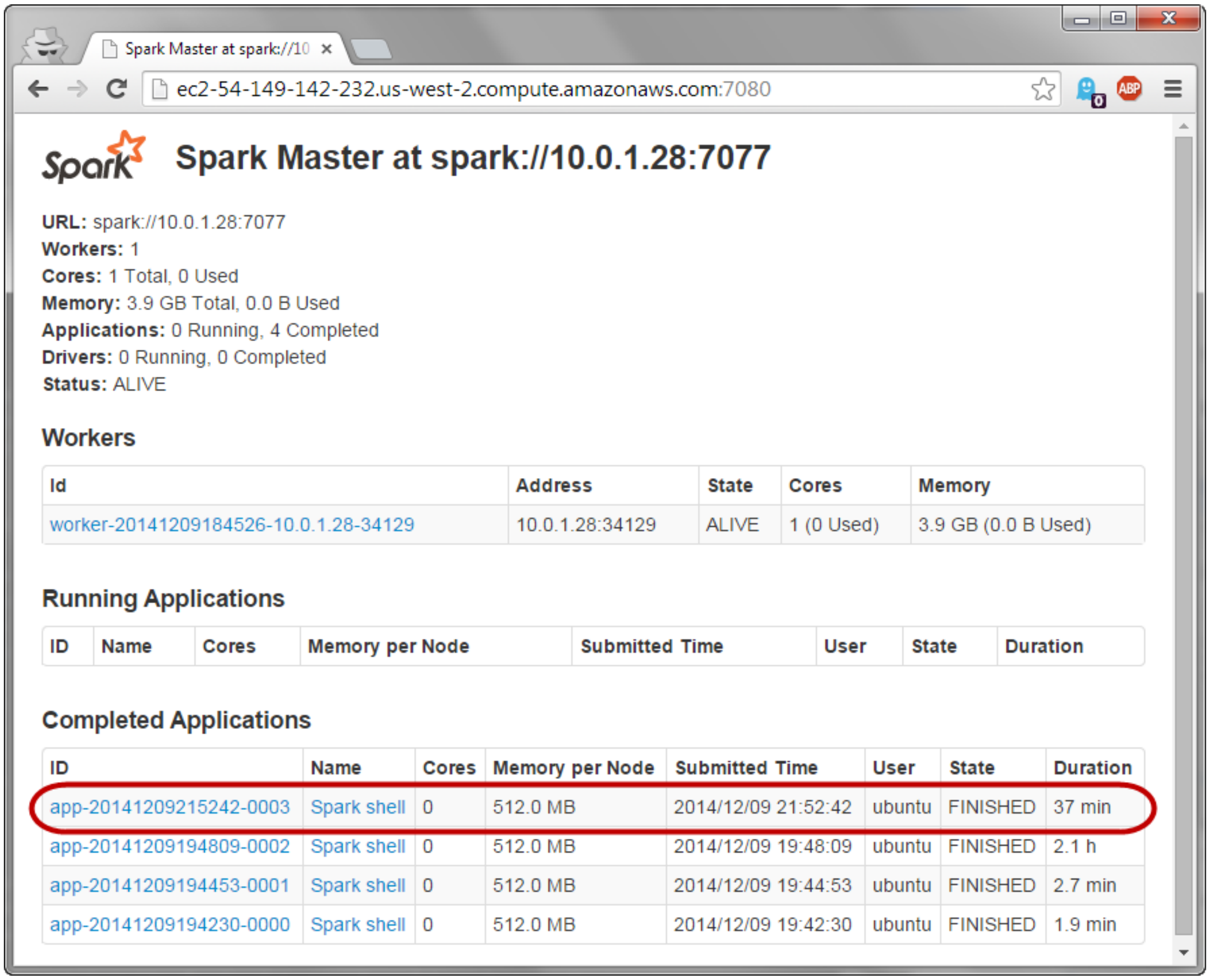

DataStax OpsCenter
http://ec2-54-68-34-4.us-we
ec2-54-68-34-4.us-west-2.compute.amazonaws.com:4040/stages
This webpage is not available
Reload
More

Note that starting with Spark 1.0, there is a History Server exposed by default on port 18080 which can show event history + logging for past applications. This port can be configured using the setting: `spark.history.ui.port`. However, it doesn't appear to be configured in DSE 4.6 yet.

Finally the Spark Master page will show the application as Finished:

Open a Chrome or FireFox browser window and go to:

`http://<your EC2 instance's public DNS hostname>:7080/`


Spark Master at spark://10
ec2-54-149-142-232.us-west-2.compute.amazonaws.com:7080

Spark Master at spark://10.0.1.28:7077

URL: spark://10.0.1.28:7077
Workers: 1
Cores: 1 Total, 0 Used
Memory: 3.9 GB Total, 0.0 B Used
Applications: 0 Running, 4 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Id	Address	State	Cores	Memory
worker-20141209184526-10.0.1.28-34129	10.0.1.28:34129	ALIVE	1 (0 Used)	3.9 GB (0.0 B Used)

Running Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

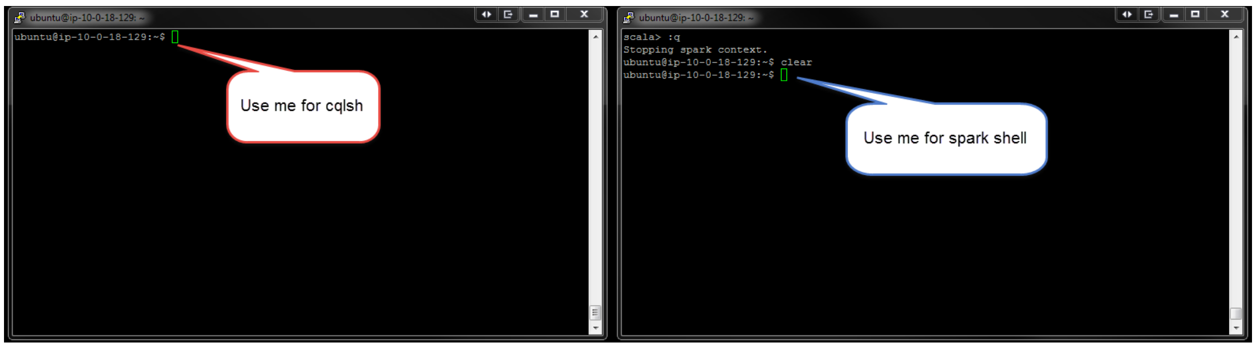
ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20141209215242-0003	Spark shell	0	512.0 MB	2014/12/09 21:52:42	ubuntu	FINISHED	37 min
app-20141209194809-0002	Spark shell	0	512.0 MB	2014/12/09 19:48:09	ubuntu	FINISHED	2.1 h
app-20141209194453-0001	Spark shell	0	512.0 MB	2014/12/09 19:44:53	ubuntu	FINISHED	2.7 min
app-20141209194230-0000	Spark shell	0	512.0 MB	2014/12/09 19:42:30	ubuntu	FINISHED	1.9 min

This is a good place to break for lunch.

Create a table in CQL shell and import it into a Spark RDD

For this exercise, we'll attempt a full life cycle implementation with Cassandra & Spark. Let's create a simple Cassandra table/column family, populate it with data and then import that data into a Spark RDD. Then we will perform some transformations on the data and write the results back into a new Cassandra table/CF.

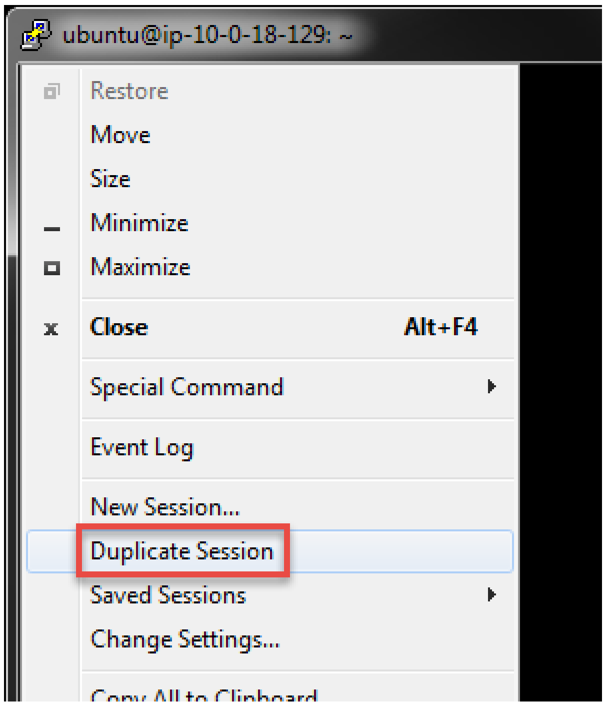
Before proceeding, it'll help to have 2 SSH client windows open for the remainder of the lab exercises. This way, you can use one window for cqlsh commands and another window for spark shell commands.



```
ubuntu@ip-10-0-18-129:~$
```

```
scala> :q
Stopping spark context.
ubuntu@ip-10-0-18-129:~$ clear
ubuntu@ip-10-0-18-129:~$
```

If you're using PuTTY on Windows, you can clone your window by choosing the "Duplicate Session" option:



From the designated C* window, launch the cql shell:

```
ubuntu@ip-10-0-18-129:~$ cqlsh localhost 9160
Connected to DSEcluster01 at localhost:9160.
```

```
[cqlsh 4.1.1 | Cassandra 2.0.11.83 | DSE 4.6.0 | CQL spec 3.1.1 | Thrift
protocol 19.39.0]
Use HELP for help.
cqlsh>
```

Next, create a KEYSPACE named ‘`tinykeyspace`’ and a table/column family within it named ‘`keyvaluetable`’. Finally, populate the table with 2 rows:

```
cqlsh> CREATE KEYSPACE tinykeyspace WITH REPLICATION = {'class' :
'NetworkTopologyStrategy', 'Analytics' : 1};

cqlsh> CREATE TABLE tinykeyspace.keyvaluetable (word text PRIMARY KEY, count
int);

cqlsh> INSERT INTO tinykeyspace.keyvaluetable (word, "count") VALUES ('foo',
20);

cqlsh> INSERT INTO tinykeyspace.keyvaluetable (word, "count") VALUES ('bar',
40);
```

Now, let’s move to the 2nd SSH window for Spark. Before we launch the Spark shell, let’s turn up the logging level to INFO from the default setting of WARN.

Use VIM or your favorite text editor to edit the log4j.properties file:

```
ubuntu@ip-10-0-53-24:~$ sudo vim /etc/dse/spark/log4j.properties
```

Simply change the 2nd line in the from the WARN -> INFO like so:

```
# Set everything to be logged to the console
log4j.rootCategory=INFO, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
<rest of file truncated>
```

Save + quit out of the file.

Open a fresh Spark shell and import those 2 rows (notice all the new INFO messages that can help with troubleshooting issues in the future):

```
ubuntu@ip-10-0-53-24:~$ dse spark
```

```
14/12/09 22:45:42 INFO SecurityManager: Changing view acls to: ubuntu,
14/12/09 22:45:42 INFO SecurityManager: Changing modify acls to: ubuntu,
14/12/09 22:45:42 INFO SecurityManager: SecurityManager: authentication
disabled; ui acls disabled; users with view permissions: Set(ubuntu, ); users
with modify permissions: Set(ubuntu, )
14/12/09 22:45:42 INFO HttpServer: Starting HTTP Server
14/12/09 22:45:42 INFO Utils: Successfully started service 'HTTP class
server' on port 36763.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.1.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_51)
Type in expressions to have them evaluated.
Type :help for more information.
Creating SparkContext...
Initializing SparkContext with MASTER: spark://10.0.1.28:7077
14/12/09 22:45:50 INFO SecurityManager: Changing view acls to: ubuntu,
14/12/09 22:45:50 INFO SecurityManager: Changing modify acls to: ubuntu,
14/12/09 22:45:50 INFO SecurityManager: SecurityManager: authentication
disabled; ui acls disabled; users with view permissions: Set(ubuntu, ); users
with modify permissions: Set(ubuntu, )
14/12/09 22:45:51 INFO Slf4jLogger: Slf4jLogger started
14/12/09 22:45:51 INFO Remoting: Starting remoting
14/12/09 22:45:51 INFO Remoting: Remoting started; listening on addresses
:[akka.tcp://sparkDriver@ip-10-0-1-28.us-west-2.compute.internal:37258]
14/12/09 22:45:51 INFO Remoting: Remoting now listens on addresses:
[akka.tcp://sparkDriver@ip-10-0-1-28.us-west-2.compute.internal:37258]
14/12/09 22:45:51 INFO Utils: Successfully started service 'sparkDriver' on
port 37258.
14/12/09 22:45:51 INFO SparkEnv: Registering MapOutputTracker
14/12/09 22:45:51 INFO SparkEnv: Registering BlockManagerMaster
14/12/09 22:45:51 INFO DiskBlockManager: Created local directory at
/var/lib/spark/rdd/spark-local-20141209224551-9426
14/12/09 22:45:51 INFO Utils: Successfully started service 'Connection
manager for block manager' on port 42719.
14/12/09 22:45:51 INFO ConnectionManager: Bound socket to port 42719 with id
= ConnectionManagerId(ip-10-0-1-28.us-west-2.compute.internal,42719)
14/12/09 22:45:51 INFO MemoryStore: MemoryStore started with capacity 246.0
MB
```

```
14/12/09 22:45:51 INFO BlockManagerMaster: Trying to register BlockManager
14/12/09 22:45:51 INFO BlockManagerMasterActor: Registering block manager
ip-10-0-1-28.us-west-2.compute.internal:42719 with 246.0 MB RAM
14/12/09 22:45:51 INFO BlockManagerMaster: Registered BlockManager
14/12/09 22:45:51 INFO HttpFileServer: HTTP File server directory is
/tmp/spark/ubuntu/spark-246b4b29-cd73-41bf-8467-3186d7f756f0
14/12/09 22:45:51 INFO HttpServer: Starting HTTP Server
14/12/09 22:45:51 INFO Utils: Successfully started service 'HTTP file server'
on port 37220.
14/12/09 22:45:52 INFO Utils: Successfully started service 'SparkUI' on port
4040.
14/12/09 22:45:52 INFO SparkUI: Started SparkUI at
http://ip-10-0-1-28.us-west-2.compute.internal:4040
14/12/09 22:45:52 INFO AppClient$ClientActor: Connecting to master
spark://10.0.1.28:7077...
14/12/09 22:45:52 INFO SparkDeploySchedulerBackend: SchedulerBackend is ready
for scheduling beginning after reached minRegisteredResourcesRatio: 0.0
Created spark context..
Spark context available as sc.
14/12/09 22:45:53 INFO SparkDeploySchedulerBackend: Connected to Spark
cluster with app ID app-20141209224553-0004
14/12/09 22:45:53 INFO AppClient$ClientActor: Executor added:
app-20141209224553-0004/0 on worker-20141209184526-10.0.1.28-34129
(10.0.1.28:34129) with 1 cores
14/12/09 22:45:53 INFO SparkDeploySchedulerBackend: Granted executor ID
app-20141209224553-0004/0 on hostPort 10.0.1.28:34129 with 1 cores, 512.0 MB
RAM
14/12/09 22:45:53 INFO AppClient$ClientActor: Executor updated:
app-20141209224553-0004/0 is now RUNNING
14/12/09 22:46:01 INFO NativeCodeLoader: Loaded the native-hadoop library
14/12/09 22:46:01 INFO HadoopProcessCheck: Native child wrapper found and
usable: /usr/share/dse/hadoop/native/Linux-amd64-64/bin/hadoop-child-wrapper
(DataStax Hadoop Child Wrapper version 1.1)
HiveSQLContext available as hc.
14/12/09 22:46:03 INFO SparkDeploySchedulerBackend: Registered executor:
Actor[akka.tcp://sparkExecutor@10.0.1.28:51630/user/Executor#-1232675106]
with ID 0
14/12/09 22:46:03 INFO BlockManagerMasterActor: Registering block manager
10.0.1.28:43547 with 265.4 MB RAM
CassandraSQLContext available as csc.
Type in expressions to have them evaluated.
Type :help for more information.
```

```
scala>
```

Run the showSchema command to display the schema of all the viewable C* tables:

```
scala> :showSchema
14/12/09 22:46:59 INFO Cluster: New Cassandra host /10.0.1.28:9042 added
14/12/09 22:46:59 INFO CassandraConnector: Connected to Cassandra cluster:
DSEcluster01
14/12/09 22:46:59 INFO LocalNodeFirstLoadBalancingPolicy: Adding host
10.0.1.28 (Analytics)
14/12/09 22:46:59 INFO LocalNodeFirstLoadBalancingPolicy: Adding host
10.0.1.28 (Analytics)
========================================
 Keyspace: HiveMetaStore
========================================
 Table: MetaStore
----------------------------------------
 - key     : String              (partition key column)
 - entity  : String              (clustering column)
 - value   : java.nio.ByteBuffer


========================================
 Keyspace: tinykeyspace
========================================
 Table: keyvaluetable
----------------------------------------
 - word    : String              (partition key column)
 - count   : Int


========================================
 Keyspace: OpsCenter
========================================
 Table: rollups60
----------------------------------------
 - key     : String              (partition key column)
 - column1 : BigInt              (clustering column)
 - value   : java.nio.ByteBuffer

 Table: settings
----------------------------------------
```

```
 - key     : java.nio.ByteBuffer (partition key column)
 - column1 : java.nio.ByteBuffer (clustering column)
 - value   : java.nio.ByteBuffer

 Table: rollups86400
----------------------------------------
 - key     : String              (partition key column)
 - column1 : BigInt              (clustering column)
 - value   : java.nio.ByteBuffer

 Table: events_timeline
----------------------------------------
 - key     : String              (partition key column)
 - column1 : Long                (clustering column)
 - value   : java.nio.ByteBuffer

 Table: pdps
----------------------------------------
 - key     : String              (partition key column)
 - column1 : String              (clustering column)
 - value   : java.nio.ByteBuffer

 Table: rollups7200
----------------------------------------
 - key     : String              (partition key column)
 - column1 : BigInt              (clustering column)
 - value   : java.nio.ByteBuffer

 Table: events
----------------------------------------
 - key     : String              (partition key column)
 - action  : Long
 - level   : Long
 - success : Boolean
 - time    : Long

 Table: bestpractice_results
----------------------------------------
 - key     : String              (partition key column)
 - column1 : BigInt              (clustering column)
 - value   : java.nio.ByteBuffer

 Table: rollups300
```

```
----------------------------------------
 - key     : String              (partition key column)
 - column1 : BigInt              (clustering column)
 - value   : java.nio.ByteBuffer

scala> 14/12/09 22:46:59 INFO CassandraConnector: Disconnected from Cassandra
cluster: DSEcluster01
<you may have to hit enter to get back to the Scala shell>
```

Hmm, that was a lot of output. Let's restrict the schema output to just the `tinykeyspace`:

```
scala> :showSchema tinykeyspace
14/12/09 22:48:11 INFO Cluster: New Cassandra host /10.0.1.28:9042 added
14/12/09 22:48:11 INFO CassandraConnector: Connected to Cassandra cluster:
DSEcluster01
14/12/09 22:48:11 INFO LocalNodeFirstLoadBalancingPolicy: Adding host
10.0.1.28 (Analytics)
14/12/09 22:48:11 INFO LocalNodeFirstLoadBalancingPolicy: Adding host
10.0.1.28 (Analytics)
========================================
 Keyspace: tinykeyspace
========================================
 Table: keyvaluetable
----------------------------------------
 - word  : String (partition key column)
 - count : Int

scala> 14/12/09 22:48:11 INFO CassandraConnector: Disconnected from Cassandra
cluster: DSEcluster01
```

That's better.

You can even drill down to just showing the schema of a specific table/CF within a keyspace:

```
scala> :showSchema tinykeyspace keyvaluetable
14/12/09 22:48:45 INFO Cluster: New Cassandra host /10.0.1.28:9042 added
14/12/09 22:48:45 INFO CassandraConnector: Connected to Cassandra cluster:
DSEcluster01
```

```
14/12/09 22:48:45 INFO LocalNodeFirstLoadBalancingPolicy: Adding host
10.0.1.28 (Analytics)
14/12/09 22:48:45 INFO LocalNodeFirstLoadBalancingPolicy: Adding host
10.0.1.28 (Analytics)
========================================
 Keyspace: tinykeyspace
========================================
 Table: keyvaluetable
----------------------------------------
 - word  : String (partition key column)
 - count : Int

scala> 14/12/09 22:48:45 INFO CassandraConnector: Disconnected from Cassandra
cluster: DSEcluster01
```

Okay, moving on to more interesting things. Time to create a RDD out of a Cassandra table. But first let's try to do it with a C* table that doesn't exist:

```
scala> val keyvalueRDD = sc.cassandraTable("tinykeyspace", "NOGO")
keyvalueRDD:
com.datastax.spark.connector.rdd.CassandraRDD[com.datastax.spark.connector.Ca
ssandraRow] = CassandraRDD[1] at RDD at CassandraRDD.scala:48

scala> keyvalueRDD.count()
java.io.IOException: Table not found: tinykeyspace.NOGO
        at
com.datastax.spark.connector.rdd.CassandraRDD.tableDef$lzycompute(CassandraRD
D.scala:229)
```

Nice, we got a good, clean error there.

Now do it for real with a working column family named keyvaluetable**:**

```
scala> val keyvalueRDD = sc.cassandraTable("tinykeyspace", "keyvaluetable")
keyvalueRDD:
com.datastax.bdp.spark.CassandraRDD[com.datastax.bdp.spark.CassandraRow] =
CassandraRDD[0] at RDD at CassandraRDD.scala:32
```

Try counting how many items are in the RDD:

```
scala> keyvalueRDD.count()
14/12/09 22:51:36 INFO Cluster: New Cassandra host /10.0.1.28:9042 added
14/12/09 22:51:36 INFO CassandraConnector: Connected to Cassandra cluster:
DSEcluster01
```

```
14/12/09 22:51:36 INFO LocalNodeFirstLoadBalancingPolicy: Adding host
10.0.1.28 (Analytics)
14/12/09 22:51:36 INFO LocalNodeFirstLoadBalancingPolicy: Adding host
10.0.1.28 (Analytics)
14/12/09 22:51:36 INFO SharkContext: Starting job: count at <console>:62
14/12/09 22:51:36 INFO DAGScheduler: Got job 0 (count at <console>:62) with 1
output partitions (allowLocal=false)
14/12/09 22:51:36 INFO DAGScheduler: Final stage: Stage 0(count at
<console>:62)
14/12/09 22:51:36 INFO DAGScheduler: Parents of final stage: List()
14/12/09 22:51:36 INFO CassandraConnector: Disconnected from Cassandra
cluster: DSEcluster01
14/12/09 22:51:36 INFO DAGScheduler: Missing parents: List()
14/12/09 22:51:36 INFO DAGScheduler: Submitting Stage 0 (CassandraRDD[2] at
RDD at CassandraRDD.scala:48), which has no missing parents
14/12/09 22:51:36 INFO MemoryStore: ensureFreeSpace(4264) called with
curMem=0, maxMem=257918238
14/12/09 22:51:36 INFO MemoryStore: Block broadcast_0 stored as values in
memory (estimated size 4.2 KB, free 246.0 MB)
14/12/09 22:51:36 INFO MemoryStore: ensureFreeSpace(2312) called with
curMem=4264, maxMem=257918238
14/12/09 22:51:36 INFO MemoryStore: Block broadcast_0_piece0 stored as bytes
in memory (estimated size 2.3 KB, free 246.0 MB)
14/12/09 22:51:36 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory
on ip-10-0-1-28.us-west-2.compute.internal:42719 (size: 2.3 KB, free: 246.0
MB)
14/12/09 22:51:36 INFO BlockManagerMaster: Updated info of block
broadcast_0_piece0
14/12/09 22:51:36 INFO DAGScheduler: Submitting 1 missing tasks from Stage 0
(CassandraRDD[2] at RDD at CassandraRDD.scala:48)
14/12/09 22:51:36 INFO TaskSchedulerImpl: Adding task set 0.0 with 1 tasks
14/12/09 22:51:36 INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0,
10.0.1.28, ANY, 1935 bytes)
14/12/09 22:51:36 INFO ConnectionManager: Accepted connection from
[ip-10-0-1-28.us-west-2.compute.internal/10.0.1.28:49060]
14/12/09 22:51:37 INFO SendingConnection: Initiating connection to
[ip-10-0-1-28.us-west-2.compute.internal/10.0.1.28:43547]
14/12/09 22:51:37 INFO SendingConnection: Connected to
[ip-10-0-1-28.us-west-2.compute.internal/10.0.1.28:43547], 1 messages pending
14/12/09 22:51:37 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory
on 10.0.1.28:43547 (size: 2.3 KB, free: 265.4 MB)
14/12/09 22:51:40 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0)
in 3391 ms on 10.0.1.28 (1/1)
```

```
14/12/09 22:51:40 INFO DAGScheduler: Stage 0 (count at <console>:62) finished
in 3.397 s
14/12/09 22:51:40 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks
have all completed, from pool
14/12/09 22:51:40 INFO SharkContext: Job finished: count at <console>:62,
took 3.583911411 s
res5: Long = 2

scala> 14/12/09 22:51:40 INFO StatsReportListener: Finished stage:
org.apache.spark.scheduler.StageInfo@8146520
14/12/09 22:51:40 INFO StatsReportListener: task runtime:(count: 1, mean:
3391.000000, stdev: 0.000000, max: 3391.000000, min: 3391.000000)
14/12/09 22:51:40 INFO StatsReportListener:     0%      5%      10%
25%50%      75%     90%     95%     100%
14/12/09 22:51:40 INFO StatsReportListener:     3.4 s   3.4 s   3.4 s   3.4 s
3.4 s   3.4 s   3.4 s   3.4 s   3.4 s
14/12/09 22:51:40 INFO StatsReportListener: fetch wait time:(count: 1, mean:
0.000000, stdev: 0.000000, max: 0.000000, min: 0.000000)
14/12/09 22:51:40 INFO StatsReportListener:     0%      5%      10%
25%50%      75%     90%     95%     100%
14/12/09 22:51:40 INFO StatsReportListener:     0.0 ms  0.0 ms  0.0 ms  0.0
ms      0.0 ms  0.0 ms  0.0 ms  0.0 ms  0.0 ms
14/12/09 22:51:40 INFO StatsReportListener: remote bytes read:(count: 1,
mean: 0.000000, stdev: 0.000000, max: 0.000000, min: 0.000000)
14/12/09 22:51:40 INFO StatsReportListener:     0%      5%      10%
25%50%      75%     90%     95%     100%
14/12/09 22:51:40 INFO StatsReportListener:     0.0 B   0.0 B   0.0 B   0.0 B
0.0 B   0.0 B   0.0 B   0.0 B   0.0 B
14/12/09 22:51:40 INFO StatsReportListener: task result size:(count: 1, mean:
864.000000, stdev: 0.000000, max: 864.000000, min: 864.000000)
14/12/09 22:51:40 INFO StatsReportListener:     0%      5%      10%
25%50%      75%     90%     95%     100%
14/12/09 22:51:40 INFO StatsReportListener:     864.0 B 864.0 B 864.0 B 864.0
B      864.0 B 864.0 B 864.0 B 864.0 B 864.0 B
14/12/09 22:51:40 INFO StatsReportListener: executor (non-fetch) time pct:
(count: 1, mean: 85.756414, stdev: 0.000000, max: 85.756414, min: 85.756414)
14/12/09 22:51:40 INFO StatsReportListener:     0%      5%      10%
25%50%      75%     90%     95%     100%
14/12/09 22:51:40 INFO StatsReportListener:     86 %    86 %    86 %    86
%86 %    86 %    86 %    86 %    86 %
14/12/09 22:51:40 INFO StatsReportListener: fetch wait time pct: (count: 1,
mean: 0.000000, stdev: 0.000000, max: 0.000000, min: 0.000000)
```

```
14/12/09 22:51:40 INFO StatsReportListener:     0%      5%      10%
25%50%      75%     90%     95%     100%
14/12/09 22:51:40 INFO StatsReportListener:      0 %     0 %     0 %     0 %
0 %     0 %     0 %     0 %     0 %
14/12/09 22:51:40 INFO StatsReportListener: other time pct: (count: 1, mean:
14.243586, stdev: 0.000000, max: 14.243586, min: 14.243586)
14/12/09 22:51:40 INFO StatsReportListener:     0%      5%      10%
25%50%      75%     90%     95%     100%
14/12/09 22:51:40 INFO StatsReportListener:     14 %    14 %    14 %    14
%14 %    14 %    14 %    14 %    14 %
<you may have to hit ENTER to get back to the scala prompt>
```

Whew, that was a lot of output with stuff about stages, DAG scheduler and statistics. Let's take a minute to exit the Spark shell and downgrade the logging level. There is a JIRA (SPARK-3444) in place to allow Spark shell users to upgrade or downgrade the logging level on the fly from within the shell, but that has not been implemented yet.

Exit the Spark shell:

```
scala> :q
Stopping spark context.
14/12/09 22:53:55 INFO SparkUI: Stopped Spark web UI at
http://ip-10-0-1-28.us-west-2.compute.internal:4040
14/12/09 22:53:55 INFO DAGScheduler: Stopping DAGScheduler
14/12/09 22:53:55 INFO SparkDeploySchedulerBackend: Shutting down all
executors
14/12/09 22:53:55 INFO SparkDeploySchedulerBackend: Asking each executor to
shut down
14/12/09 22:53:56 INFO ConnectionManager: Removing ReceivingConnection to
ConnectionManagerId(ip-10-0-1-28.us-west-2.compute.internal,43547)
14/12/09 22:53:56 INFO ConnectionManager: Removing SendingConnection to
ConnectionManagerId(ip-10-0-1-28.us-west-2.compute.internal,43547)
14/12/09 22:53:56 INFO MapOutputTrackerMasterActor: MapOutputTrackerActor
stopped!
14/12/09 22:53:56 INFO ConnectionManager: Selector thread was interrupted!
14/12/09 22:53:56 INFO ConnectionManager: ConnectionManager stopped
14/12/09 22:53:56 INFO MemoryStore: MemoryStore cleared
14/12/09 22:53:56 INFO BlockManager: BlockManager stopped
14/12/09 22:53:56 INFO BlockManagerMaster: BlockManagerMaster stopped
14/12/09 22:53:56 INFO RemoteActorRefProvider$RemotingTerminator: Shutting
down remote daemon.
```

```
14/12/09 22:53:56 INFO RemoteActorRefProvider$RemotingTerminator: Remote
daemon shut down; proceeding with flushing remote transports.
14/12/09 22:53:56 INFO SharkContext: Successfully stopped SparkContext
14/12/09 22:53:56 INFO Remoting: Remoting shut down
14/12/09 22:53:56 INFO RemoteActorRefProvider$RemotingTerminator: Remoting
shut down.
14/12/09 22:53:56 INFO SparkUI: Stopped Spark web UI at
http://ip-10-0-1-28.us-west-2.compute.internal:4040
14/12/09 22:53:56 INFO SharkContext: SparkContext already stopped
```

Use VIM or your favorite text editor to edit the log4j.properties file:

```
ubuntu@ip-10-0-53-24:~$ sudo vim /etc/dse/spark/log4j.properties
```

Simply change the 2nd line in the from the INFO -> WARN like so:

```
# Set everything to be logged to the console
log4j.rootCategory=WARN, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
<rest of file truncated>
```

Save + quit out of the file.

Re-open the Spark shell and import those 2 rows (notice all the INFO messages are gone):

```
ubuntu@ip-10-0-53-24:~$ dse spark
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.1.0
      /_/
```

```
Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_51)
Type in expressions to have them evaluated.
Type :help for more information.
Creating SparkContext...
Initializing SparkContext with MASTER: spark://10.0.1.28:7077
Created spark context..
Spark context available as sc.
HiveSQLContext available as hc.
CassandraSQLContext available as csc.
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Try displaying the RDD we created earlier:

```
scala> keyvalueRDD.collect()
<console>:24: error: not found: value keyvalueRDD
              keyvalueRDD.collect()
              ^
```

Oh yeah, we closed and re-opened the Spark shell. So, this is a new Spark application. We will need to recreate the RDD then try again:

```
scala> val keyvalueRDD = sc.cassandraTable("tinykeyspace", "keyvaluetable")
keyvalueRDD:
com.datastax.bdp.spark.CassandraRDD[com.datastax.bdp.spark.CassandraRow] =
CassandraRDD[0] at RDD at CassandraRDD.scala:32

scala> keyvalueRDD.collect()
res9: Array[com.datastax.spark.connector.CassandraRow] =
Array(CassandraRow{word: bar, count: 40}, CassandraRow{word: foo, count: 20})
```

Remember that the collect() function sends the full RDD back to the driver, so it must be able to handle that amount of data.

There are different ways to print the contents of the RDD. For example, here we are converting the RDD to an Array, then printing each item manually using a `foreach` loop:

```
scala> keyvalueRDD.toArray.foreach(println)
warning: there were 1 deprecation warning(s); re-run with -deprecation for
details
CassandraRow{word: bar, count: 40}
CassandraRow{word: foo, count: 20}
```

Well, how did I know that I could call the `toArray()` function on the RDD? With tab autocompletion, of course. Try it yourself:

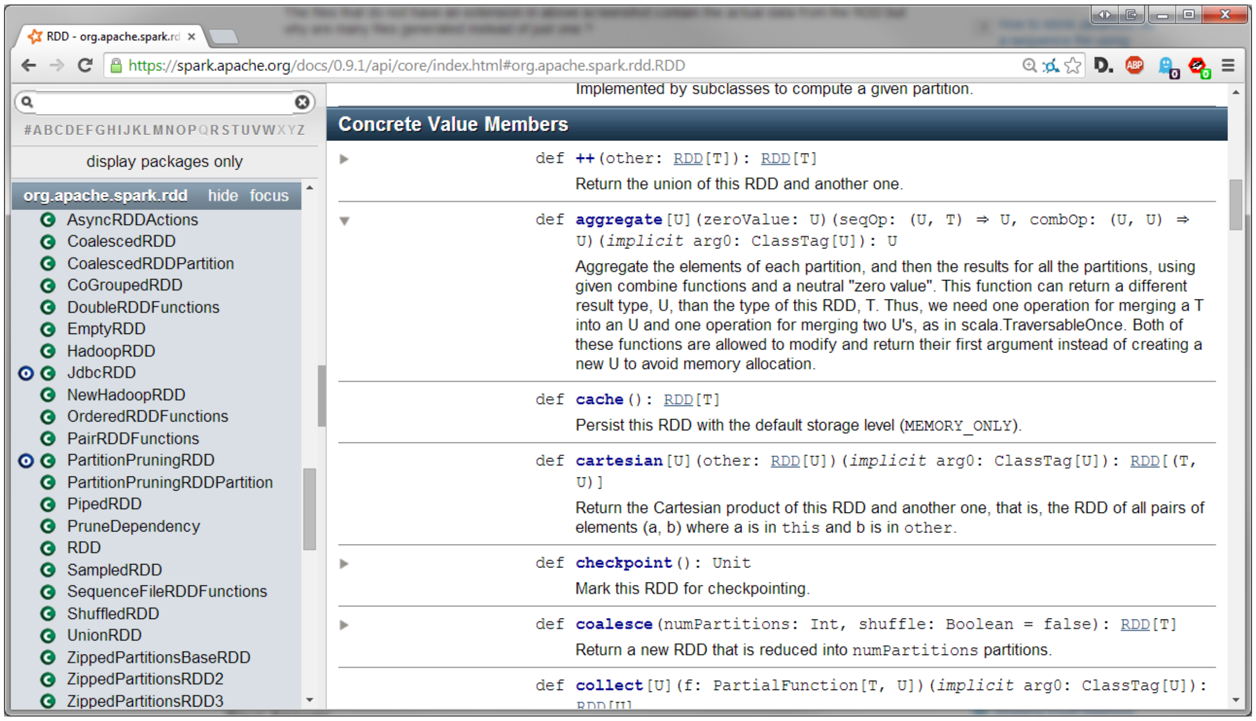
```
scala> keyvalueRDD.<hit tab once>
++                         aggregate                  as
asGraph                    asGraphBuilder             asInstanceOf
cache                      cartesian                  checkColumnsExistence
checkpoint                 coalesce                   collect
columnNames                compute                    context
count                      countApprox                countApproxDistinct
countByValue               countByValueApprox         dependencies
distinct                   fetchSize                  filter
filterWith                 first                      flatMap
flatMapWith                fold                       foreach
foreachPartition           foreachWith                generator
generator_=                getCheckpointFile          getPartitions
getStorageLevel            glom                       groupBy
id                         isCheckpointed             isInstanceOf
iterator                   keyBy                      keyspaceName
map                        mapPartitions
mapPartitionsWithContext
mapPartitionsWithIndex     mapPartitionsWithSplit     mapWith
name                       name_=                     partitioner
partitions                 persist                    pipe
preferredLocations         reduce                     repartition
rowTransformer             sample                     saveAsObjectFile
saveAsTextFile             select                     setGenerator
setName                    sparkContext               splitSize
subtract                   tableDef                   tableName
take                       takeOrdered                takeSample
toArray                    toDebugString              toJavaRDD
toString                   top                        union
unpersist                  verify                     where
zip                        zipPartitions
```

No need to actually select any of the functions above. Instead hit backspace and delete everything at the prompt.

In the output above, I've highlighted some of the common **transformations in blue** and the common **actions in green**, but my highlights are not comprehensive.

I also point out Spark's **RDD persistence capabilities in red**, which will look into later in this lab. These functions are neither transformations nor actions.

As this is not a pure Spark development course, we won't explore the full API in detail here, however if you are interested in what all the above operations do, check out the Spark docs here:
https://spark.apache.org/docs/1.1.0/api/scala/index.html#org.apache.spark.package


RDD - org.apache.spark.rd
https://spark.apache.org/docs/0.9.1/api/core/index.html#org.apache.spark.rdd.RDD

#ABCDEFGHIJKLMNOPQRSTUVWXYZ
display packages only
org.apache.spark.rdd hide focus

- AsyncRDDActions
- CoalescedRDD
- CoalescedRDDPartition
- CoGroupedRDD
- DoubleRDDFunctions
- EmptyRDD
- HadoopRDD
- JdbcRDD
- NewHadoopRDD
- OrderedRDDFunctions
- PairRDDFunctions
- PartitionPruningRDD
- PartitionPruningRDDPartition
- PipedRDD
- PruneDependency
- RDD
- SampledRDD
- SequenceFileRDDFunctions
- ShuffledRDD
- UnionRDD
- ZippedPartitionsBaseRDD
- ZippedPartitionsRDD2
- ZippedPartitionsRDD3

Implemented by subclasses to compute a given partition.

Concrete Value Members

`def ++(other: RDD[T]): RDD[T]`
Return the union of this RDD and another one.

`def aggregate[U](zeroValue: U)(seqOp: (U, T) ⇒ U, combOp: (U, U) ⇒ U)(implicit arg0: ClassTag[U]): U`
Aggregate the elements of each partition, and then the results for all the partitions, using given combine functions and a neutral "zero value". This function can return a different result type, U, than the type of this RDD, T. Thus, we need one operation for merging a T into an U and one operation for merging two U's, as in scala.TraversableOnce. Both of these functions are allowed to modify and return their first argument instead of creating a new U to avoid memory allocation.

`def cache(): RDD[T]`
Persist this RDD with the default storage level (`MEMORY_ONLY`).

`def cartesian[U](other: RDD[U])(implicit arg0: ClassTag[U]): RDD[(T, U)]`
Return the Cartesian product of this RDD and another one, that is, the RDD of all pairs of elements (a, b) where a is in `this` and b is in `other`.

`def checkpoint(): Unit`
Mark this RDD for checkpointing.

`def coalesce(numPartitions: Int, shuffle: Boolean = false): RDD[T]`
Return a new RDD that is reduced into `numPartitions` partitions.

`def collect[U](f: PartialFunction[T, U])(implicit arg0: ClassTag[U]):`

Or you can learn more about the common transformations in Spark 1.1.0 here:
https://spark.apache.org/docs/1.1.0/programming-guide.html#transformations

And the common actions here:
https://spark.apache.org/docs/1.1.0/programming-guide.html#actions
What if you add a new row to the Cassandra table at this point?

Switch to the Cassandra CQL shell and insert a new row into the table named `keyvaluetable`:

```
cqlsh> INSERT INTO tinykeyspace.keyvaluetable (word, "count") VALUES ('abc', 
99);
```

Now if you run the `collect()` action on the same `keyvalueRDD`, do you think it'll show the new 3rd row for 'abc'? Think about this.. what behavior will you expect below? Run the following command from the scala Spark shell:

```
scala> keyvalueRDD.collect()
```

```
res5: Array[com.datastax.bdp.spark.CassandraRow] = Array(CassandraRow[word:
bar, count: 40], CassandraRow[word: abc, count: 99], CassandraRow[word: foo,
count: 20])
```

Although some of the Apache Spark literature may state that RDDs are immutable as you can see above, the details are more nuanced. If the underlying storage that feeds the RDD is immutable, like HDFS (let's say you never run updates on any of the files), then the RDD is basically immutable as well since it'll always represent the same, exact underlying data.

However, in the examples above, when we run `keyvalueRDD.collect()`, what's actually happening is that each time you run `collect()` a Directed Acyclic Graph (DAG) is being executed. Every transformation you run builds the DAG metadata in the background. Each Spark action executes a DAG of stages to be performed on the cluster. Compare this to MapReduce, which creates a DAG with only two predefined stages - Map and Reduce. Spark DAGs can contain any number of stages, with more complicated operations launching more stages.

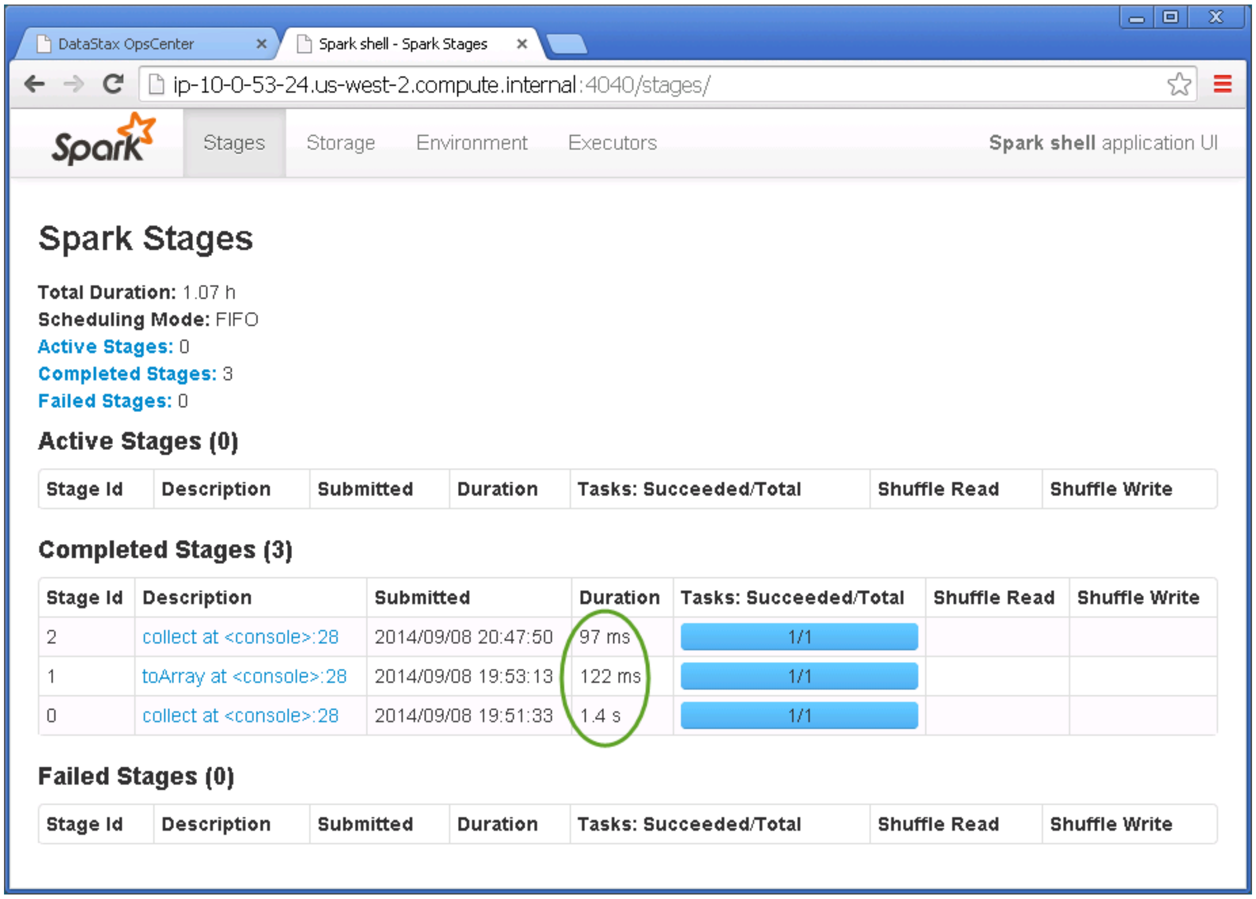
Typically, when you run an action, a DAG gets executed that has to re-read data each time all the way from disk and recreate the RDD (unless the OS buffer caches it, but Spark won't be aware of this).

However, it is possible to persist your chosen RDDs in memory, so that if you plan on running multiple actions on a common RDD, Spark will try it's best to keep the RDD in memory so that future operations (transformations & actions) against the cached RDD will return much faster. This idea of memory persistence is what makes Spark up to 100x faster than traditional MapReduce in some use cases. This way data that has to be passed between different stages of computation don't have to spill to disk and be read back from disk by the next stage. Instead Spark can operate at memory speeds.

Let's try to persist some RDDs to memory!

But first take a look at the currently running "Spark shell" application's Spark Stages execution times.

Open a Chrome or FireFox browser window and go to:
`http://<your EC2 instance's public DNS hostname>:4040/stages/`


DataStax OpsCenter
Spark shell - Spark Stages
ip-10-0-53-24.us-west-2.compute.internal:4040/stages/
Stages
Storage
Environment
Executors
Spark shell application UI

Spark Stages

Total Duration: 1.07 h
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 3
Failed Stages: 0

Active Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write

Completed Stages (3)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write
2	collect at <console>:28	2014/09/08 20:47:50	97 ms	1/1		
1	toArray at <console>:28	2014/09/08 19:53:13	122 ms	1/1		
0	collect at <console>:28	2014/09/08 19:51:33	1.4 s	1/1		

Failed Stages (0)

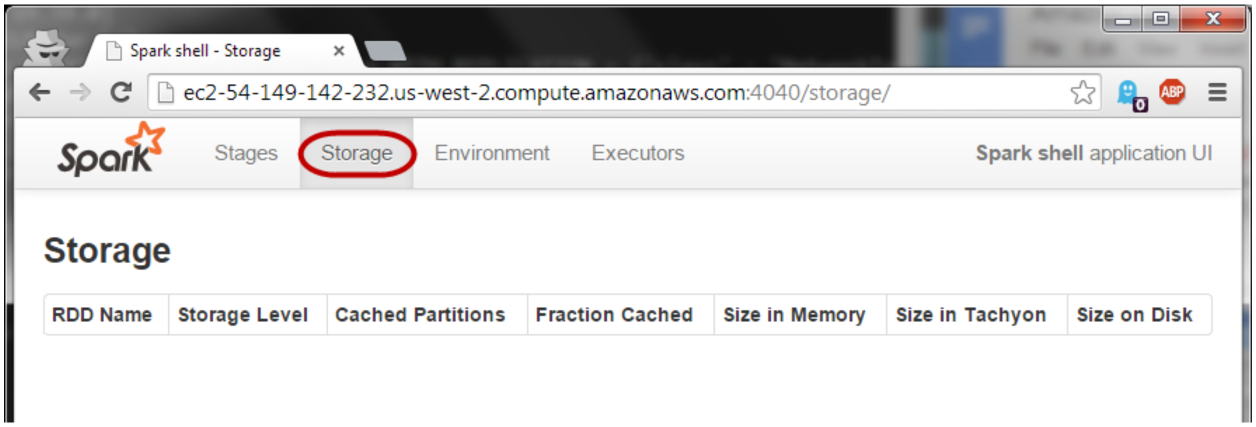
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write

The very first action you ran on keyvalueRDD (`keyvalueRDD.collect()`) took ~1.4 seconds to run, but the next 2 subsequent actions only took about 100-150 ms to complete.

Persist the `keyvalueRDD` to memory like so:

```
scala> keyvalueRDD.persist()
res6: org.apache.spark.rdd.RDD[com.datastax.bdp.spark.CassandraRow] =
CassandraRDD[0] at RDD at CassandraRDD.scala:32
```

In the Spark UI that you have open, click on the Storage tab:


Spark shell - Storage
ec2-54-149-142-232.us-west-2.compute.amazonaws.com:4040/storage/
Spark
Stages
Storage
Environment
Executors
Spark shell application UI

Storage

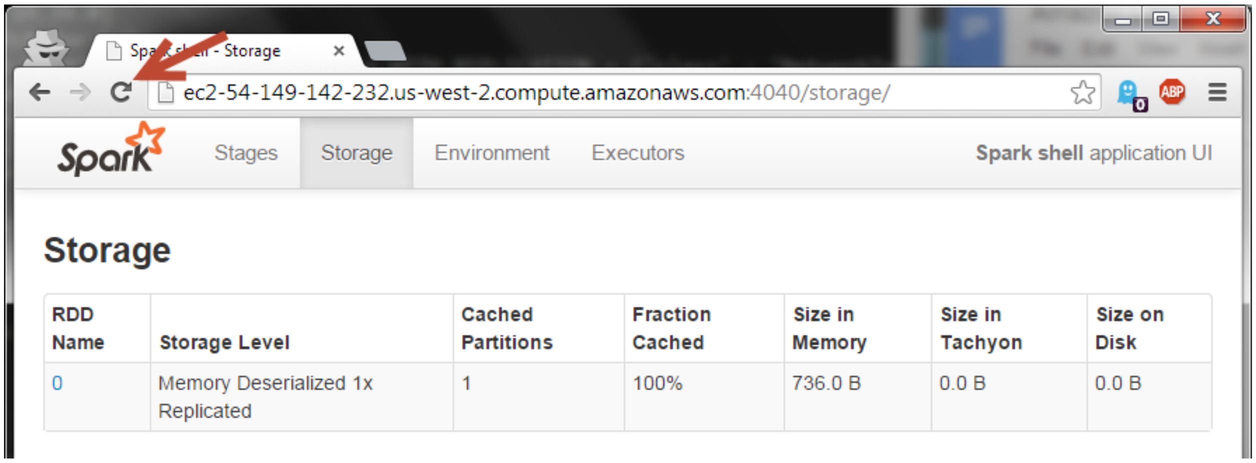
RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk

Hmm, even after we ran a `.persist()` we still don't see the RDD listed under storage. Did it even get persisted to memory??

Okay, let's try running an action on the maybe persisted RDD:

```
scala> keyvalueRDD.collect()
res7: Array[com.datastax.bdp.spark.CassandraRow] = Array(CassandraRow[word:
bar, count: 40], CassandraRow[word: abc, count: 99], CassandraRow[word: foo,
count: 20])
```

Now refresh the Spark Storage UI:

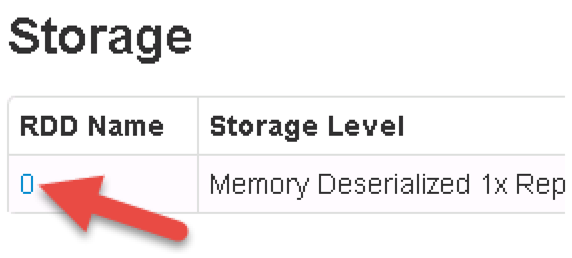

Spark shell - Storage
ec2-54-149-142-232.us-west-2.compute.amazonaws.com:4040/storage/
Stages
Storage
Environment
Executors
Spark shell application UI

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
0	Memory Deserialized 1x Replicated	1	100%	736.0 B	0.0 B	0.0 B

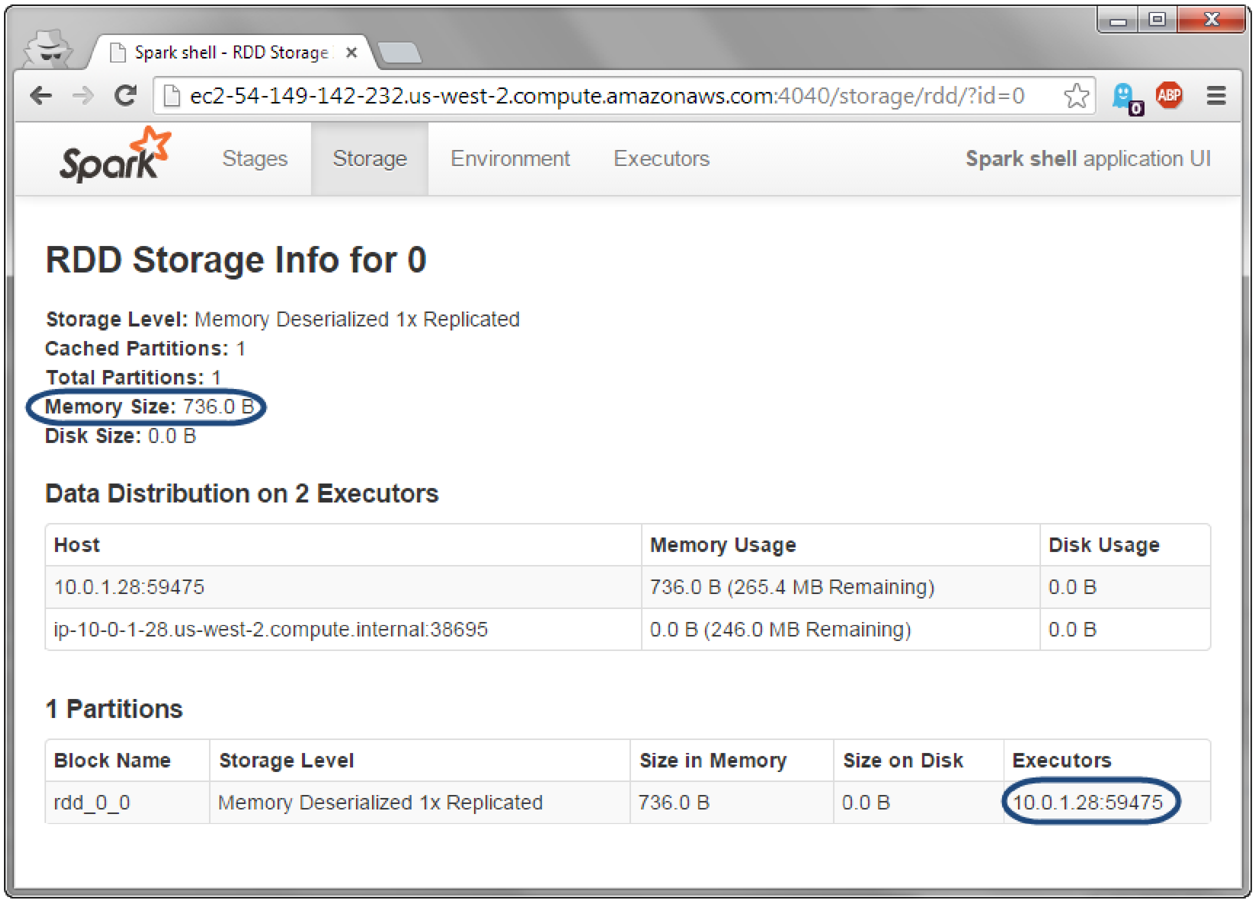
And Ta-Da! The RDD is now persisted. So, what happened is that when we ran the `.persist()` operation, the RDD didn't actually get persisted right away. Instead, remember that Spark executes everything except for actions lazily. We had to actually execute a DAG with the `.collect()` action, before the `.persist()` step actually took place.

Click on the RDD Named 0 to drill one level deeper into details:



Storage

RDD Name	Storage Level
0	Memory Deserialized 1x Rep


Spark shell - RDD Storage
ec2-54-149-142-232.us-west-2.compute.amazonaws.com:4040/storage/rdd/?id=0
Stages
Storage
Environment
Executors
Spark shell application UI

RDD Storage Info for 0

Storage Level: Memory Deserialized 1x Replicated
Cached Partitions: 1
Total Partitions: 1
Memory Size: 736.0 B
Disk Size: 0.0 B

Data Distribution on 2 Executors

Host	Memory Usage	Disk Usage
10.0.1.28:59475	736.0 B (265.4 MB Remaining)	0.0 B
ip-10-0-1-28.us-west-2.compute.internal:38695	0.0 B (246.0 MB Remaining)	0.0 B

1 Partitions

Block Name	Storage Level	Size in Memory	Size on Disk	Executors
rdd_0_0	Memory Deserialized 1x Replicated	736.0 B	0.0 B	10.0.1.28:59475

Now you can see that this RDD is persisted 100% to memory and is occupying **736 bytes** of RAM. You can also see in the bottom right corner that the RDD is made of just 1 partition, which is located on the executor **10.0.1.28:59475**.

Note that it is possible to unpersist an RDD with the `.unpersist()` operation.

Now that the RDD is persisted, let's try to change the underlying Cassandra table by adding a new row and seeing if it shows up in future Spark actions against the keyvalueRDD.

Switch to the Cassandra CQL shell and insert a new row with the row key 'afterpersistance' into the table named keyvaluetable:

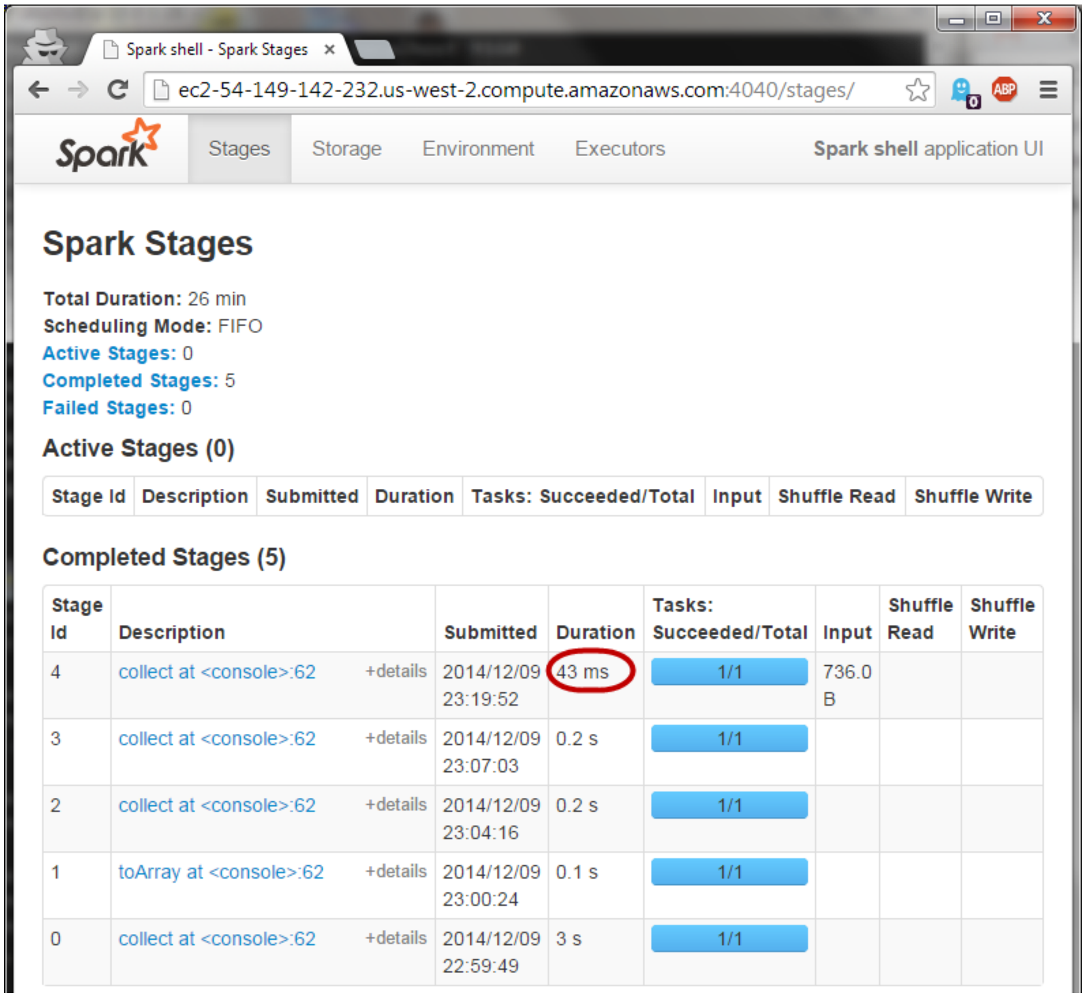
```
cqlsh> INSERT INTO tinykeyspace.keyvaluetable (word, "count") VALUES
('afterpersistance', 42);
```

Now if you run the `collect()` action on the same `keyvalueRDD`, do you think it'll show the new 4th row for '`afterpersistance`'? Think about this.. what behavior will you expect below? Run the following command from the scala Spark shell:

```
scala> keyvalueRDD.collect()
res12: Array[com.datastax.bdp.spark.CassandraRow] = Array(CassandraRow[word:
bar, count: 40], CassandraRow[word: abc, count: 99], CassandraRow[word: foo,
count: 20])
```

Hmm, we don't see the new 4th row with a value of 42..

Switch over to the Spark Stages UI (refresh the page if you have to) and locate Stage id #4. Notice that this time the `.collect()` action completed significantly faster than any of the previous times… just ~24 ms! In this last action, the RDD was cached in memory, so analysis time was significantly faster.


Spark shell - Spark Stages
ec2-54-149-142-232.us-west-2.compute.amazonaws.com:4040/stages/
Spark
Stages
Storage
Environment
Executors
Spark shell application UI

Spark Stages

Total Duration: 26 min
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 5
Failed Stages: 0

Active Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write

Completed Stages (5)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
4	collect at <console>:62 +details	2014/12/09 23:19:52	43 ms	1/1	736.0 B		
3	collect at <console>:62 +details	2014/12/09 23:07:03	0.2 s	1/1			
2	collect at <console>:62 +details	2014/12/09 23:04:16	0.2 s	1/1			
1	toArray at <console>:62 +details	2014/12/09 23:00:24	0.1 s	1/1			
0	collect at <console>:62 +details	2014/12/09 22:59:49	3 s	1/1			

Let's try something a bit radical.

Switch to the Cassandra CQL shell and quit out of cqlsh:

```
cqlsh> quit;
ubuntu@ip-10-0-18-129:~$
```

List the process ids (pids) of the running Java processes:

```
ubuntu@ip-10-0-18-129:~$ sudo jps
19623 CoarseGrainedExecutorBackend
7600 DseSparkMaster
18810 SparkReplMain
6748 jar
8402 DseSparkWorker
23233 Jps
4078 DseDaemon
```

Notice that there is a process named `CoarseGrainedExecutorBackend`. Note down the unique pid for it (*the # will be different for your environment*), as you'll need it for the next step.

Now terminate that process abruptly:

```
ubuntu@ip-10-0-18-129:~$ sudo kill -9 19623
```

In the other SSH window, where the Spark shell was running, you should now see this error:

```
scala> 14/09/05 02:36:21 ERROR TaskSchedulerImpl: Lost executor 0 on
10.0.18.129: remote Akka client disassociated
```

Back at the cqlsh/jps window, run the jps command again and notice that the Executor has been restarted under a different pid:

```
ubuntu@ip-10-0-18-129:~$ sudo jps
7600 DseSparkMaster
24271 CoarseGrainedExecutorBackend
18810 SparkReplMain
6748 jar
8402 DseSparkWorker
24385 Jps
4078 DseDaemon
```

The other Spark pids remain unchanged.

So, what happened? Why did we terminate the executor?

Well, the reason is not to just demonstrate that the Spark Worker will restart any abruptly killed Executors, but rather to demo the next point...

In the scala Spark shell window, just hit enter once to get back to the scala shell:

```
scala> 14/09/05 02:36:21 ERROR TaskSchedulerImpl: Lost executor 0 on
10.0.18.129: remote Akka client disassociated<hit enter once>

scala>
```

Run the `.collect()` action against the `keyvalueRDD` again:

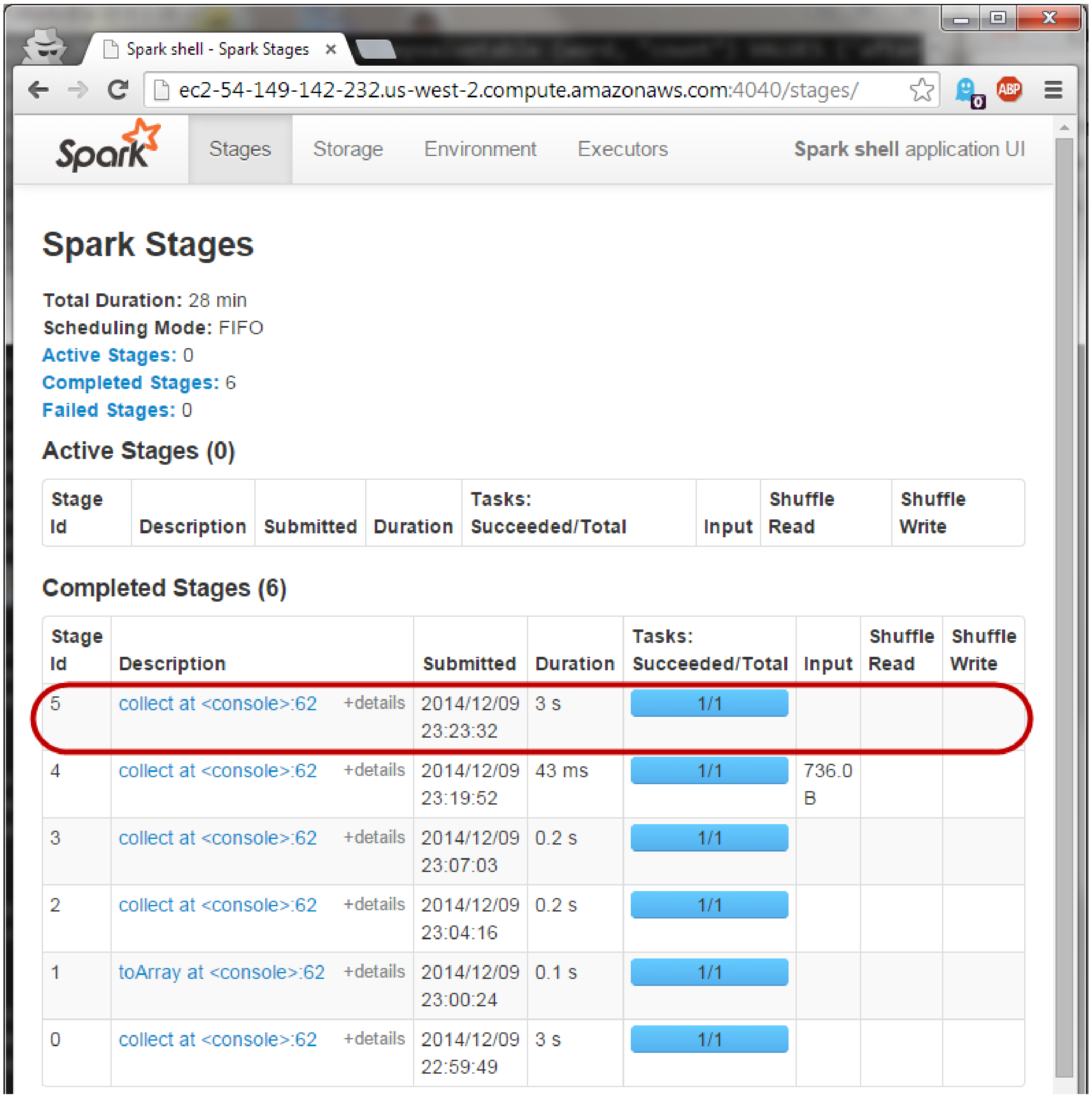
```
scala> keyvalueRDD.collect()
res13: Array[com.datastax.bdp.spark.CassandraRow] = Array(CassandraRow[word:
bar, count: 40], CassandraRow[word: abc, count: 99], CassandraRow[word: foo,
count: 20], CassandraRow[word: afterpersistance, count: 42])
```

Did you notice that the `'afterpersistance' row` that we had added earlier to Cassandra is now part of the `keyvalueRDD`?

What happened here is that by terminating the Spark Executor, the `keyvalueRDD` was also destroyed because it lived as a deserialized Java object in the heap of that Executor JVM.

By launching a fresh action against the RDD, Spark was forced to recreate the RDD, by reading all the data from Cassandra again. In that process a fresh RDD representation of the C* table was actually created.

Go to, or refresh the Spark Stages UI:


Spark shell - Spark Stages
ec2-54-149-142-232.us-west-2.compute.amazonaws.com:4040/stages/
Stages
Storage
Environment
Executors
Spark shell application UI

Spark Stages

Total Duration: 28 min
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 6
Failed Stages: 0

Active Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write

Completed Stages (6)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
5	collect at <console>:62 +details	2014/12/09 23:23:32	3 s	1/1			
4	collect at <console>:62 +details	2014/12/09 23:19:52	43 ms	1/1	736.0 B		
3	collect at <console>:62 +details	2014/12/09 23:07:03	0.2 s	1/1			
2	collect at <console>:62 +details	2014/12/09 23:04:16	0.2 s	1/1			
1	toArray at <console>:62 +details	2014/12/09 23:00:24	0.1 s	1/1			
0	collect at <console>:62 +details	2014/12/09 22:59:49	3 s	1/1			

In the Spark Stages UI above, notice that the latest `.collect()` action that we ran, took ~3 seconds to complete. This is because Spark could not rely on reading an in-memory RDD to quickly compute the results of the action for us.

Switch to the cqlsh/jps window and run the following commands to insert a new row into C*:

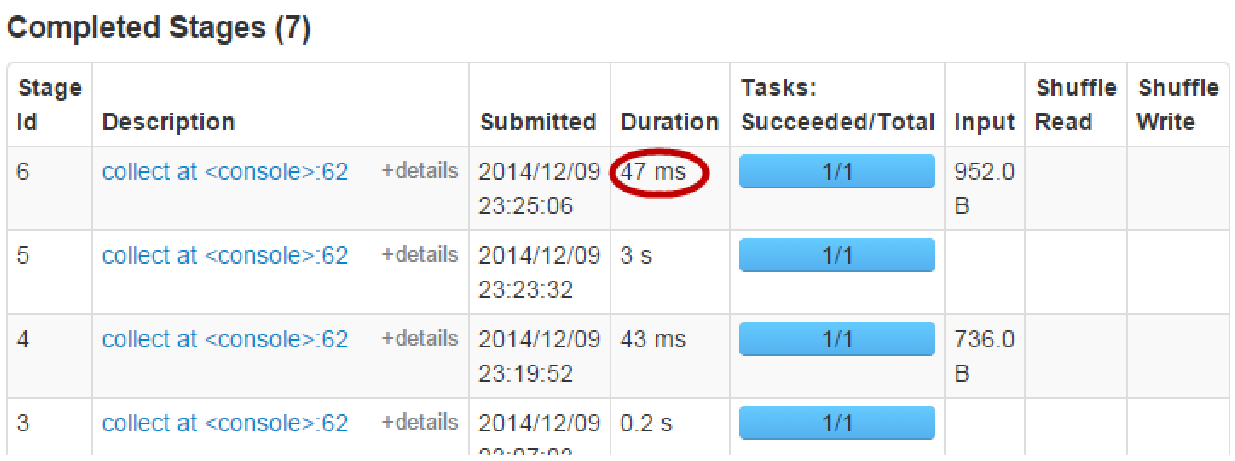
```
ubuntu@ip-10-0-18-129:~$ cqlsh localhost 9160
Connected to DSEcluster01 at localhost:9160.
[cqlsh 4.1.1 | Cassandra 2.0.11.83 | DSE 4.6.0 | CQL spec 3.1.1 | Thrift
protocol 19.39.0]
Use HELP for help.

cqlsh> INSERT INTO tinykeyspace.keyvaluetable (word, "count") VALUES
('afterradical', 1000);
```

Switch to the Spark shell window and run another `.collect()` action:

```
scala> keyvalueRDD.collect()
res14: Array[com.datastax.bdp.spark.CassandraRow] = Array(CassandraRow[word:
bar, count: 40], CassandraRow[word: abc, count: 99], CassandraRow[word: foo,
count: 20], CassandraRow[word: afterpersistance, count: 42])
```

Notice that the new `'afterradical'` row doesn't show up in the RDD because the RDD was already persisted in memory from the earlier `.collect()` we did right after the Executor restart.

You can also confirm this by refreshing the Spark Stages UI and noticing that the duration of the last .collect() action that we ran was only ~47 ms. So, obviously the new Executor that started under the new pid went ahead and cached the RDD after it ran the first DAG that required that RDD (remember the first DAG after the respawn took 3 secs to execute):



Completed Stages (7)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
6	collect at <console>:62	+details	2014/12/09 23:25:06	47 ms	1/1	952.0 B		
5	collect at <console>:62	+details	2014/12/09 23:23:32	3 s	1/1			
4	collect at <console>:62	+details	2014/12/09 23:19:52	43 ms	1/1	736.0 B		
3	collect at <console>:62	+details	2014/12/09	0.2 s	1/1			

Before concluding this section, let's find the Spark log file that captures the fact that the Spark Executor abruptly terminated and was restarted.

Switch to the cqlsh/jps window and quit from the CQL shell:

```
cqlsh> quit;
ubuntu@ip-10-0-18-129:~$
```

Tail the last 3 lines from Spark's `master.log`:

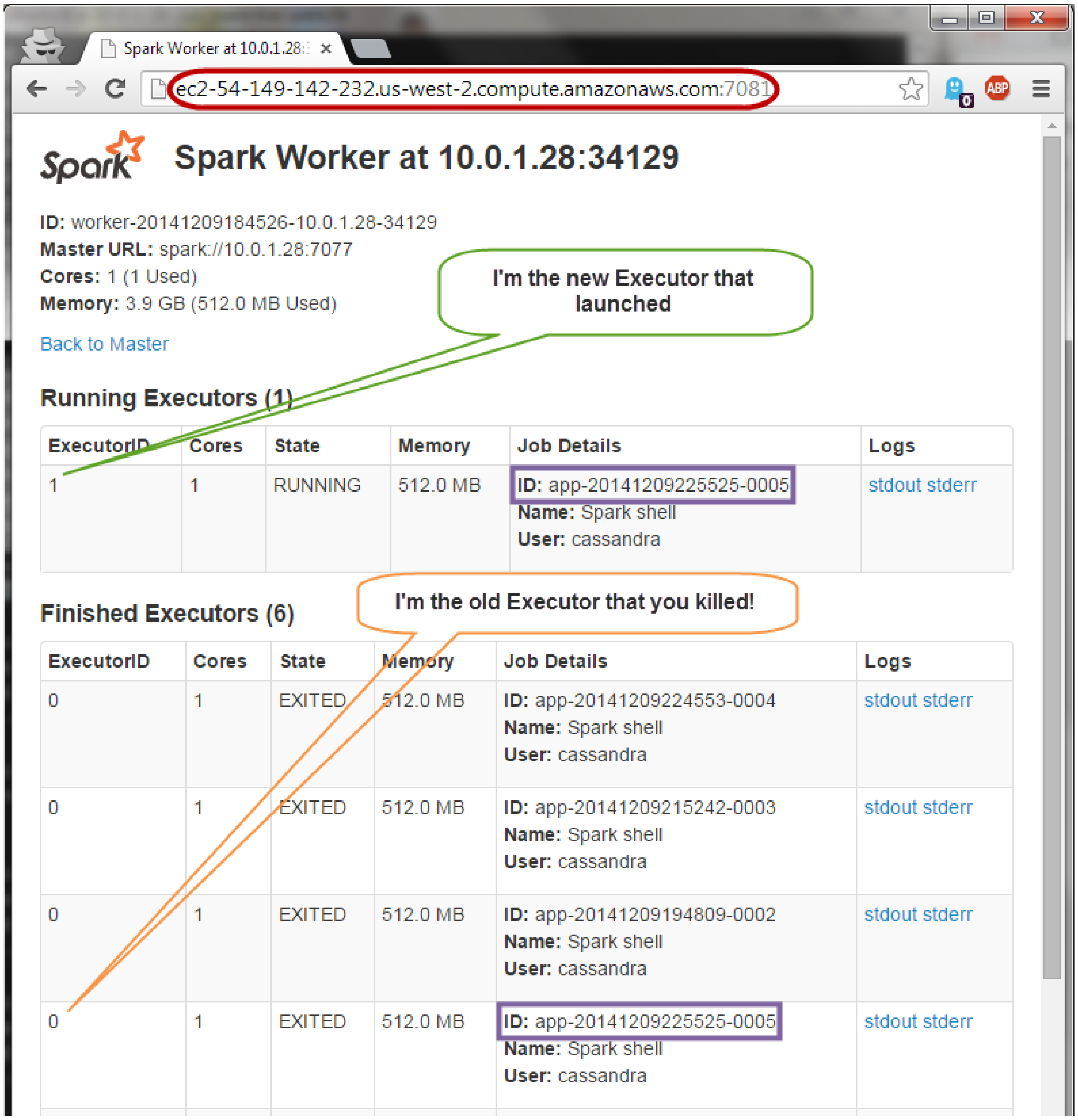
```
ubuntu@ip-10-0-18-129:~$ tail -3 /var/log/spark/master.log
INFO 2014-12-09 22:55:26 com.datastax.spark.connector.cql.CassandraConnector:
Disconnected from Cassandra cluster: DSEcluster01
INFO 2014-12-09 23:22:58 org.apache.spark.deploy.master.DseSparkMaster:
Removing executor app-20141209225525-0005/0 because it is EXITED
INFO 2014-12-09 23:22:58 org.apache.spark.deploy.master.DseSparkMaster:
Launching executor app-20141209225525-0005/1 on worker
worker-20141209184526-10.0.1.28-34129
```

You should see that the 1st line was logged at a much earlier timestamp that then next two lines. The first line shows the original Spark Executor disconnecting gracefully.

The next two lines are around the same timestamp. They show that the SparkMaster has detected a failure of the first executor. Then the SparkMaster asked the worker to start a new Executor.

Load the Spark Worker UI on port 7081, and you can correlate the two Executor IDs from the master.log file to the same IDs in the UI:

```
http://<your EC2 instance's public DNS hostname>:7081
```



Spark Worker at 10.0.1.28:
ec2-54-149-142-232.us-west-2.compute.amazonaws.com:7081
I'm the new Executor that launched
I'm the old Executor that you killed!

Spark Worker at 10.0.1.28:34129

ID: worker-20141209184526-10.0.1.28-34129
Master URL: spark://10.0.1.28:7077
Cores: 1 (1 Used)
Memory: 3.9 GB (512.0 MB Used)

Back to Master

Running Executors (1)

ExecutorID	Cores	State	Memory	Job Details	Logs
1	1	RUNNING	512.0 MB	**ID:** app-20141209225525-0005 **Name:** Spark shell **User:** cassandra	stdout stderr

Finished Executors (6)

ExecutorID	Cores	State	Memory	Job Details	Logs
0	1	EXITED	512.0 MB	**ID:** app-20141209224553-0004 **Name:** Spark shell **User:** cassandra	stdout stderr
0	1	EXITED	512.0 MB	**ID:** app-20141209215242-0003 **Name:** Spark shell **User:** cassandra	stdout stderr
0	1	EXITED	512.0 MB	**ID:** app-20141209194809-0002 **Name:** Spark shell **User:** cassandra	stdout stderr
0	1	EXITED	512.0 MB	**ID:** app-20141209225525-0005 **Name:** Spark shell **User:** cassandra	stdout stderr

While we're here, there is another log file to be aware of. The Spark Worker's log file. Tail the last 3 lines of this file also, and you'll be able to correlate the timestamps from the Spark Master's log file:

```
ubuntu@ip-10-0-18-129:~$ tail -3 /var/log/spark/worker.log
INFO 2014-12-09 23:22:58 org.apache.spark.deploy.worker.DseSparkWorker:
Executor app-20141209225525-0005/0 finished with state EXITED message Command
exited with code 137 exitStatus 137
INFO 2014-12-09 23:22:58 org.apache.spark.deploy.worker.DseSparkWorker: Asked
to launch executor app-20141209225525-0005/1 for Spark shell
INFO 2014-12-09 23:23:00 org.apache.spark.deploy.worker.ExecutorRunner:
Launch command: "/usr/lib/jvm/java-7-oracle/bin/java" "-cp"
":/etc/dse/spark:/usr/share/dse/spark/lib/JavaEWAH-0.3.2.jar:/usr/share/dse/s
park/lib/ST4-4.0.4.jar:/usr/share/dse/spark/lib/activation-1.1.jar
<rest of the output from this line was truncated>
```

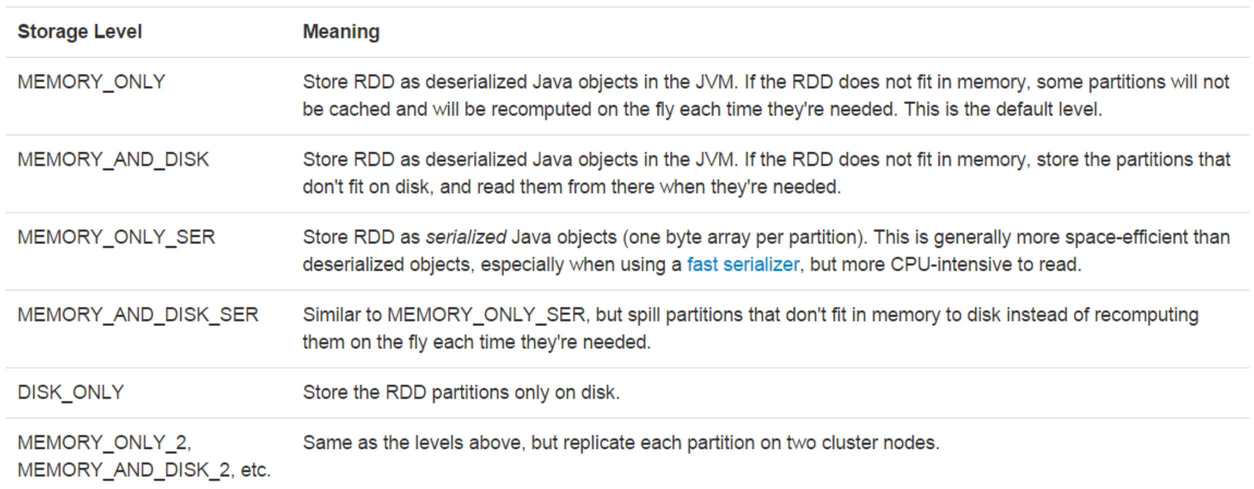
Switch to the Spark shell window and quit out the shell/application in order to get a fresh start with a new application instance before the next section:

```
scala> :q
Stopping spark context.
ubuntu@ip-10-0-18-129:~$
```

You should know that there are various persistence levels in Spark such as MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY, etc. You can read more about the pros & cons of each here:

https://spark.apache.org/docs/1.1.0/programming-guide.html#rdd-persistence

Some of the persistence levels included in Spark 1.0:



Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.

You can also define your own storage level (say, with replication factor of 3 instead of 2), then use the function factor method apply() of the StorageLevel singleton object. See the Apache docs for more info on this.

Run More Spark Commands

At this point, you should still have 2 SSH windows running, both with just the linux cmd lines (where we'll eventually start cqlsh & Spark shell again). You should also have a browser window open with probably the Spark UI loaded.

Switch to the cqlsh/jps window and run the following commands to insert a new row into C*:

```
ubuntu@ip-10-0-18-129:~$ cqlsh localhost 9160
Connected to DSEcluster01 at localhost:9160.
[cqlsh 4.1.1 | Cassandra 2.0.11.83 | DSE 4.6.0 | CQL spec 3.1.1 | Thrift
protocol 19.39.0]
Use HELP for help.
```

Check what's in the keyvaluetable column family so far:

```
cqlsh> SELECT * FROM tinykeyspace.keyvaluetable;

 word              | count
-------------------+-------
               bar |    40
               abc |    99
               foo |    20
 afterpersistance |    42
      afterradical |  1000

(5 rows)
```

Switch to the Spark shell window and this time start the Spark shell by passing in a custom `cassandra.input.split.size`:

```
ubuntu@ip-10-0-53-24:~$ dse spark -Dspark.cassandra.input.split.size=2
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.1.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_51)
Type in expressions to have them evaluated.
Type :help for more information.
Creating SparkContext...
Initializing SparkContext with MASTER: spark://10.0.1.28:7077
Created spark context..
Spark context available as sc.
HiveSQLContext available as hc.
CassandraSQLContext available as csc.


Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

The cassandra.input.split.size parameter defaults to 100,000. This is the “Approximate number of rows in a single Spark partition. The higher the value, the fewer Spark tasks are created. Increasing the value too much may limit the parallelism level.”

More details about the spark.cassandra.input.split.size and other options that can be passed during startup, can be found here:
http://www.datastax.com/documentation/datastax_enterprise/4.6/datastax_enterprise/spark/sparkCassProps.html

Our example of 2 as the input split size is quite extreme and is only being done for lab demonstration purposes. Further more, the split sizes are based on *estimation* not exact values. The estimation accuracy is limited to resolution of Cassandra index. Particularly for very small data sets, the estimations are likely to be off by large amount.

Load the Casssandra table into an RDD again:

```
scala> val keyvalueRDD = sc.cassandraTable("tinykeyspace", "keyvaluetable")
keyvalueRDD:
com.datastax.bdp.spark.CassandraRDD[com.datastax.bdp.spark.CassandraRow] =
CassandraRDD[0] at RDD at CassandraRDD.scala:32
```

Here's an interesting operation to run... partitions:

```
scala> keyvalueRDD.partitions
res4: Array[org.apache.spark.Partition] =
Array(CassandraPartition(0,Set(/10.0.1.28),Vector(CqlTokenRange(token("word")
> ?,WrappedArray(-9223372036854765808)), CqlTokenRange(token("word") <=
?,WrappedArray(-9223372036854765808))),127))
```

Quote from O'Reilly's "Learning Spark": "Remember that each RDD is split into multiple partitions, which may be computed on different nodes of the cluster. The ability to always recompute an RDD is actually why RDDs are called "resilient". When a machine holding RDD data fails, Spark uses this ability to recompute the missing partitions, transparent to the user."

There is also a partitions.size function to figure out how many partitions an RDD is split into:

```
scala> keyvalueRDD.partitions.size
res4: Int = 1
```

Hmm, even though we wanted a max of 2 rows per RDD partition, it seems that the setting did not exactly take effect. Why is it making just one RDD partition? Shouldn't it have made 3 partitions? Two partitions with 2 rows each and one partition with 1 row?

Again, I think this is a bug that DataStax will be fixing at some point...

Before continuing, let's exit out of the Spark shell and re-enter it with the default `cassandra.input.split.size` to see how the out of `.partitions` and `.partitions.size` may differ.

Quit out of the Spark shell:

```
scala> :q
Stopping spark context.
ubuntu@ip-10-0-18-129:~$
```

Re-launch the Spark shell with the default `cassandra.input.split.size` of 100,000:

```
ubuntu@ip-10-0-53-24:~$ dse spark
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.1.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_51)
Type in expressions to have them evaluated.
Type :help for more information.
Creating SparkContext...
Initializing SparkContext with MASTER: spark://10.0.1.28:7077
Created spark context..
Spark context available as sc.
HiveSQLContext available as hc.
CassandraSQLContext available as csc.
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Load the Casssandra table into an RDD again:

```
scala> val keyvalueRDD = sc.cassandraTable("tinykeyspace", "keyvaluetable")
keyvalueRDD:
com.datastax.bdp.spark.CassandraRDD[com.datastax.bdp.spark.CassandraRow] =
CassandraRDD[0] at RDD at CassandraRDD.scala:32
```

Here's an interesting operation to run... partitions:

```
scala> keyvalueRDD.partitions
res4: Array[org.apache.spark.Partition] =
Array(CassandraPartition(0,Set(/10.0.1.28),Vector(CqlTokenRange(token("word")
> ?,WrappedArray(-9223372036854765808)), CqlTokenRange(token("word") <=
?,WrappedArray(-9223372036854765808))),127))
```

There are 2 elements in the output above. Every element in a vector defines a token-range which determines partitions to be fetched from C* with a single CQL query. Spark passes this information to appropriate nodes so they know which data to fetch.

Run the size operation again:

```
scala> keyvalueRDD.partitions.size
```

```
res3: Int = 1
```

Now that our `input.split` is really large (100,000 rows), you can see that the RDD is now composed of just one partition. So all 5 rows fit in that 1 partition.

Try pulling just the first element out of the RDD:

```
scala> keyvalueRDD.first()
res6: com.datastax.spark.connector.CassandraRow = CassandraRow{word: bar,
count: 40}
```

Or try taking 2 elements:

```
scala> keyvalueRDD.take(2)
res7: Array[com.datastax.spark.connector.CassandraRow] =
Array(CassandraRow{word: bar, count: 40}, CassandraRow{word: abc, count: 99})
```

Remember that both first() and take() are actions in Spark since they actually return a result.

Notice that Spark does not give you an error if you try taking more elements than are in the RDD:

```
scala> keyvalueRDD.take(7)
res8: Array[com.datastax.spark.connector.CassandraRow] =
Array(CassandraRow{word: bar, count: 40}, CassandraRow{word: abc, count: 99},
CassandraRow{word: foo, count: 20}, CassandraRow{word: afterpersistance,
count: 42}, CassandraRow{word: afterradical, count: 1000})
```

You can also get just the column names in the RDD:

```
scala> keyvalueRDD.columnNames
res12: com.datastax.spark.connector.ColumnSelector = AllColumns
```

Note: The above command used to show this in DSE 4.5 / Spark 0.9:

```
res7: Seq[String] = Stream(word, count)
```

* **Correction, use** `keyvalueRDD.tableDef.allColumns`

Let's make a new RDD out of just the first row:

```
scala> val firstRow = keyvalueRDD.first
firstRow: com.datastax.bdp.spark.CassandraRow = CassandraRow[word: bar,
count: 40]
```

And then retrieve just one column (named count) from that row:

```
scala> firstRow.get[Int]("count")
res8: Int = 40
```

Result 8 above is actually stored in a variable that you can use. For example:

```
scala> res8.<hit tab once>
%               &               *               +               -               /
>               >=              >>              >>>             ^
asInstanceOf
isInstanceOf    toByte          toChar          toDouble        toFloat
toInt
toLong          toShort         toString        unary_+         unary_-
unary_~
|

scala> res8 > 20
res9: Boolean = true

scala> res8 > 50
res10: Boolean = false
```

You can also cast the data you pull out of the column as a long or string:

```
scala> firstRow.get[Long]("count")
res11: Long = 40

scala> firstRow.get[String]("count")
res12: String = 40
```

Instead of mapping your Cassandra rows to objects of CassandraRow class, you can directly unwrap column values into tuples of your desired type:

```
scala> val myTypeOfArray = sc.cassandraTable[(String, Int)]("tinykeyspace",
"keyvaluetable").select("word", "count").toArray
warning: there were 1 deprecation warning(s); re-run with -deprecation for
details
```

```
myTypeOfArray: Array[(String, Int)] = Array((bar,40), (abc,99), (foo,20),
(afterpersistance,42), (afterradical,1000))
```

You can invoke as on the CassandraRDD to map rows to objects with a user-defined function that adds 5 to the count column:

```
scala> val plusFive = keyvalueRDD.select("word", "count").as((w: String, c:
Int) => (w, c + 5)).toArray
warning: there were 1 deprecation warning(s); re-run with -deprecation for
details
plusFive: Array[(String, Int)] = Array((bar,45), (abc,104), (foo,25),
(afterpersistance,47), (afterradical,1005))
```

The above command transformed the RDD into a scala Array with the .toArray operator at the end. However, we can keep the RDD type intact by not doing the conversion:

```
scala> val plusFiveRDD = keyvalueRDD.select("word", "count").as((w: String,
c: Int) => (w, c + 5))
plusFiveRDD: com.datastax.bdp.spark.CassandraRDD[(String, Int)] =
CassandraRDD[6] at RDD at CassandraRDD.scala:32

scala> plusFiveRDD.collect
res13: Array[(String, Int)] = Array((bar,45), (abc,104), (foo,25),
(afterpersistance,47), (afterradical,1005))
```

Let's try to save the contents of the plusFiveRDD back to Cassandra.

```
scala> plusFiveRDD.saveToCassandra("tinykeyspace", "plusfivetable",
SomeColumns("word", "count"))
java.io.IOException: Table not found: tinykeyspace.plusfivetable2
        at
com.datastax.spark.connector.writer.TableWriter$$anonfun$10.apply(TableWriter
.scala:165)
        at
```

Hmm, what happened?

Empty iterator… well, that's not the most helpful error message, but the issue is that we have not yet created the `plusfivetable` in Cassandra under the `tinykeyspace`. This has been fixed in DSE 4.5.2 (still unreleased) and gives a proper error message.
Switch to the Cassandra CQL shell and create the `plusfivetable` in Cassandra:

```
cqlsh> CREATE TABLE tinykeyspace.plusfivetable (word text PRIMARY KEY, count
int);
```

Switch back to the Spark shell and reattempt saving the `plusFiveRDD` to Cassandra:

```
scala> plusFiveRDD.saveToCassandra("tinykeyspace", "plusfivetable",
SomeColumns("word", "count"))
```

Great. Now from the Cassandra CQL shell, try to display the contents of the plusfivetable:

```
cqlsh> SELECT * FROM tinykeyspace.plusfivetable;

 word              | count
-------------------+-------
               bar |    45
               abc |   104
               foo |    25
 afterpersistance |    47
      afterradical |  1005

(5 rows)
```

Excellent. You successfully saved the RDD to Cassandra!

Let's try a few more slightly more advanced commands. How about reading C* collection columns from Spark?

Switch to the Cassandra CQL shell and create a `userstable` in Cassandra:

```
cqlsh> CREATE TABLE tinykeyspace.userstable (user_id text PRIMARY KEY,
first_name text, last_name text, emails set<text>);
```

```
cqlsh> INSERT INTO tinykeyspace.userstable (user_id, first_name, last_name,
emails) VALUES ('howardr', 'Howard', 'Roark', {'h@gmail.com',
'hroark@aol.com'});

cqlsh> INSERT INTO tinykeyspace.userstable (user_id, first_name, last_name,
emails) VALUES ('jgalt', 'John', 'Galt', {'j@gmail.com', 'jgalt@aol.com'});

cqlsh> SELECT * FROM tinykeyspace.userstable;

 user_id | emails                              | first_name | last_name
---------+-------------------------------------+------------+-----------
   jgalt |   {'j@gmail.com', 'jgalt@aol.com'}  |       John |      Galt
 howardr | {'h@gmail.com', 'hroark@aol.com'}   |     Howard |     Roark

(2 rows)

cqlsh>
```

Switch back to the Spark shell and let's attempt reading the new C* userstable:

```
scala> val user_row = sc.cassandraTable("tinykeyspace",
"userstable").toArray.apply(1)
warning: there were 1 deprecation warning(s); re-run with -deprecation for
details
user_row: com.datastax.spark.connector.CassandraRow = CassandraRow{user_id:
howardr, emails: {h@gmail.com,hroark@aol.com}, first_name: Howard, last_name:
Roark}

scala> user_row.getList[String]("emails")
res18: Vector[String] = Vector(h@gmail.com, hroark@aol.com)

scala> user_row.get[List[String]]("emails")
res19: List[String] = List(h@gmail.com, hroark@aol.com)

scala> user_row.get[Seq[String]]("emails")
res20: Seq[String] = List(h@gmail.com, hroark@aol.com)

scala> user_row.get[String]("emails")
res21: String = [h@gmail.com, hroark@aol.com]

scala> val user_rows = sc.cassandraTable("tinykeyspace",
"userstable").toArray
```

```
user_rows: Array[com.datastax.bdp.spark.CassandraRow] =
Array(CassandraRow[user_id: jgalt, emails: [j@gmail.com, jgalt@aol.com],
first_name: John, last_name: Galt], CassandraRow[user_id: howardr, emails:
[h@gmail.com, hroark@aol.com], first_name: Howard, last_name: Roark])

scala> user_rows.length
res25: Int = 2

scala> println(user_rows.deep.mkString("\n"))
CassandraRow[user_id: jgalt, emails: [j@gmail.com, jgalt@aol.com],
first_name: John, last_name: Galt]
CassandraRow[user_id: howardr, emails: [h@gmail.com, hroark@aol.com],
first_name: Howard, last_name: Roark]
```

You can also restrict the number of fetched columns. This is critical for performance reasons. Notice the use of `.select()` below:

```
scala> val selective_cols = sc.cassandraTable("tinykeyspace",
"userstable").select("user_id").toArray
warning: there were 1 deprecation warning(s); re-run with -deprecation for
details
selective_cols: Array[com.datastax.bdp.spark.CassandraRow] =
Array(CassandraRow[user_id: jgalt], CassandraRow[user_id: howardr])

scala> println(selective_cols.deep.mkString("\n"))
CassandraRow[user_id: jgalt]
CassandraRow[user_id: howardr]
```

A cassandraRDD has a special method that passes an arbitrary CQL condition to filter the row set on the server. A different type of RDD in Spark may not necessarily have the intelligence cassandraRDDs do in this respect. Typically, to filter rows (from say HDFS), Spark does provide a filter transformation. However, this filter transformation fetches all

rows from C* first and then filters them in Spark. Some CPU cycles are wasted serializing and deserializing objects that should be excluded from the result. A cassandraRDD avoids this overhead by sending passing an arbitrary CQL condition to filter the rows server side. Let's demonstrate this:

```
scala> sc.cassandraTable("tinykeyspace", "userstable").select("user_id",
"first_name").where("first_name = ?", "Howard").toArray.foreach(println)
warning: there were 1 deprecation warning(s); re-run with -deprecation for
details
14/12/10 00:10:54 WARN TaskSetManager: Lost task 0.0 in stage 11.0 (TID 11,
10. 0.1.28): java.io.IOException: Exception during preparation of SELECT
"user_id",  "first_name" FROM "tinykeyspace"."userstable" WHERE
token("user_id") > ? AND f irst_name = ? ALLOW FILTERING: No indexed columns
present in by-columns clause  with Equal operator
    com.datastax.spark.connector.rdd.CassandraRDD.createStatement(Cassandra
RDD.scala:331)
    com.datastax.spark.connector.rdd.CassandraRDD.com$datastax$spark$connec
tor$rdd$CassandraRDD$$fetchTokenRange(CassandraRDD.scala:338)
    com.datastax.spark.connector.rdd.CassandraRDD$$anonfun$13.apply(Cassand
raRDD.scala:362)
<rest of error message truncated>
```

What do you think went wrong??

Switch to the Cassandra CQL shell and add a secondary index to `userstable` for the first_name col:

```
cqlsh> CREATE INDEX firsty ON tinykeyspace.userstable (first_name);
```

Switch back to the Spark shell and let's again attempt reading the `userstable` with the special CQL filtering:

```
scala> sc.cassandraTable("tinykeyspace", "userstable").select("user_id",
"first_name").where("first_name = ?", "Howard").toArray.foreach(println)
warning: there were 1 deprecation warning(s); re-run with -deprecation for
details
```

```
CassandraRow[user_id: howardr, first_name: Howard]
```

That's better.

The following 4 steps demonstrate creating a new RDD of numbers and applying a map and reduce function to it:

```
scala> val numbers = sc.parallelize(List(1, 2, 4, 5, 7, 20))
numbers: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[26] at
parallelize at <console>:25

scala> numbers.partitions.size
res34: Int = 2

scala> val numbers_times_2 = numbers.map(x => x * 2)
numbers_times_2: org.apache.spark.rdd.RDD[Int] = MappedRDD[27] at map at
<console>:27

scala> val sum = numbers_times_2.reduce((x, y) => x + y)
sum: Int = 78
```

The Dangers of GroupByKey

Let's look at two different ways to compute word counts in Spark, one using `reduceByKey` and the other using `groupByKey`:

```
scala> val words = Array("one", "two", "two", "three", "three", "three")
words: Array[String] = Array(one, two, two, three, three, three)
```

```
scala> val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))
wordPairsRDD: org.apache.spark.rdd.RDD[(String, Int)] = MappedRDD[29] at map
at <console>:27
```

Now let’s compute word count using two techniques, first reduceByKey, then groupByKey:

```
scala> val wordCountsWithReduce = wordPairsRDD.reduceByKey(_ + _).collect()
wordCountsWithReduce: Array[(String, Int)] = Array((one,1), (three,3),
(two,2))

scala> val wordCountsWithGroup = wordPairsRDD.groupByKey().map(t => (t._1,
t._2.sum)).collect()
wordCountsWithGroup: Array[(String, Int)] = Array((one,1), (three,3),
(two,2))
```

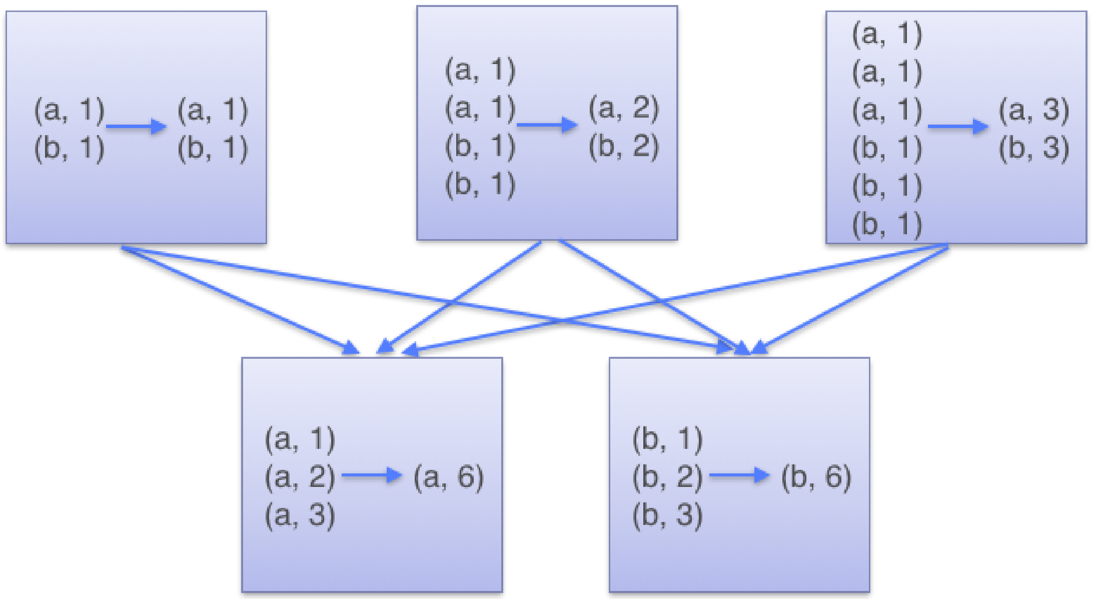
Well, they both got us similar results, but the way these two techniques work under the hood are very different.

While both of these functions will produce the correct answer, the `reduceByKey` example works much better on a large dataset. That's because Spark knows it can combine output with a common key on each partition before shuffling the data.

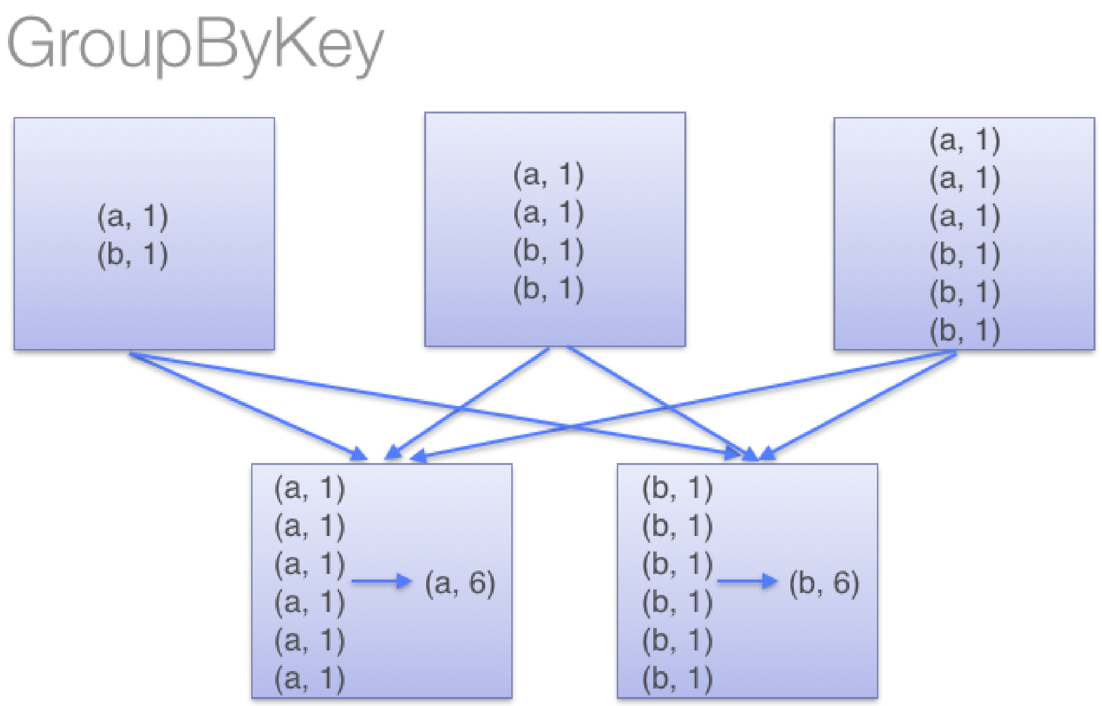
Look at the diagram below to understand what happens with `reduceByKey`. Notice how pairs on the same machine with the same key are combined (by using the lamdba function passed into `reduceByKey`) before the data is shuffled. Then the lamdba function is called again to reduce all the values from each partition to produce one final result.

Note, the keys in the example diagrams below are ‘a’ or ‘b’ (which differs from the code we wrote above where the keys were ‘one’, ‘two’ or ‘three’.

ReduceByKey



On the other hand, when calling `groupByKey` - all the key-value pairs are shuffled around. This can result in an out of memory error.



You can imagine that for a much larger dataset size, the difference in the amount of data you are shuffling becomes more exaggerated and different between `reduceByKey` and `groupByKey`.

Here are more functions to prefer over `groupByKey`:

- `combineByKey` can be used when you are combining elements but your return type differs from your input value type.
- `foldByKey` merges the values for each key using an associative function and a neutral "zero value".

Spark Streaming

The final section of this lab will demonstrate how to write a simple Spark Streaming application to count the # of words in real time coming from the netcat utility. We will print the results of the Spark Streaming application every 3 seconds to the Scala REPL, but also persist it to Cassandra.

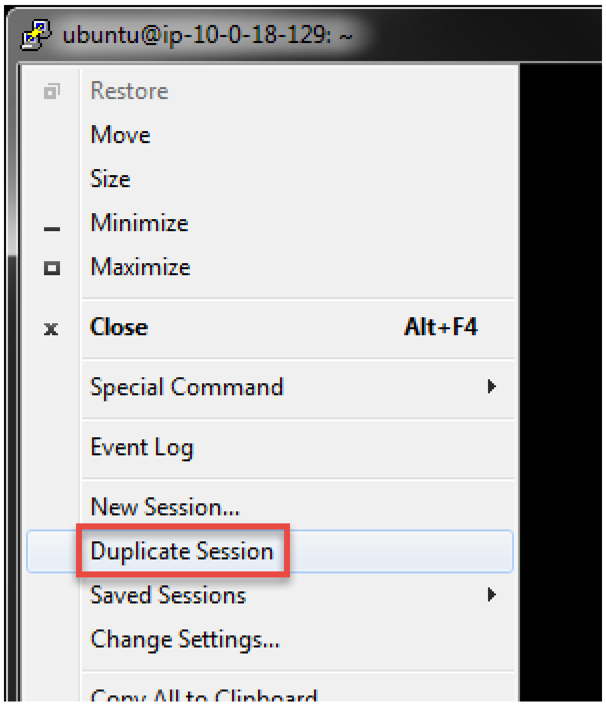
Switch to the Cassandra CQL shell and add a new column family/table named 'streaming_test' inside the keyspace 'tinykeyspace':

```
cqlsh> CREATE TABLE tinykeyspace.streaming_test (word text PRIMARY KEY, count int);

cqlsh> SELECT * FROM tinykeyspace.streaming_test;

(0 rows)
```

Okay, the table is created and it's empty. Excellent. ***Leave this window open!***

Start a NEW linux shell prompt and let's start our netcat utility:

If you're using PuTTY on Windows, you can clone your window by choosing the "Duplicate Session" option:



Netcat (often abbreviated to nc) is a computer networking service for reading from and writing to network connections using TCP or UDP.

Start the netcat utility on port 999:

```
ubuntu@ip-10-0-1-28:~$ nc -lk 9999
```

Then, into the nc application, copy and paste the following line and hit enter:

```
one two two three three three four four four four
```

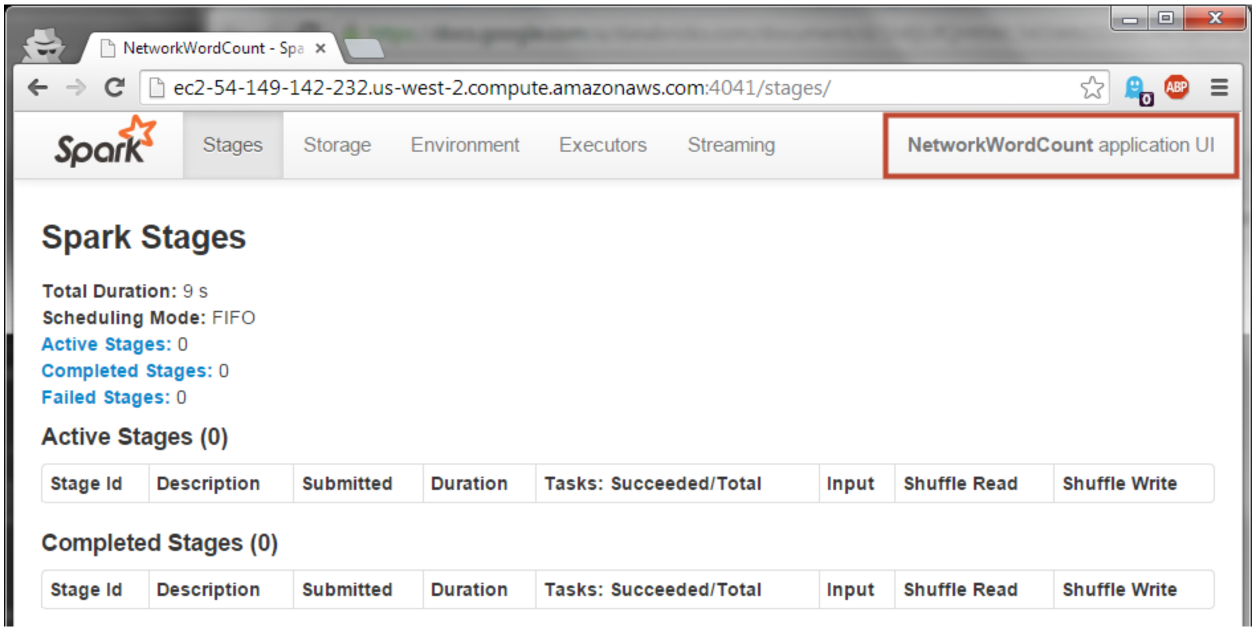
Switch to the Spark shell prompt (3rd window) and let's start the Scala REPL:

```
ubuntu@ip-10-0-1-28:~$ dse spark
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.1.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_51)
Type in expressions to have them evaluated.
Type :help for more information.
Creating SparkContext...
Initializing SparkContext with MASTER: spark://10.0.1.28:7077
Created spark context..
Spark context available as sc.
HiveSQLContext available as hc.
CassandraSQLContext available as csc.
Type in expressions to have them evaluated.
Type :help for more information.

scala> import org.apache.spark._
import org.apache.spark._

Xscala> import org.apache.spark.streaming._
import org.apache.spark.streaming._

Xscala> import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.StreamingContext._
scala> import com.datastax.spark.connector.streaming._
```

```
import com.datastax.spark.connector.streaming._

scala> import com.datastax.spark.connector.cql.CassandraConnector
import com.datastax.spark.connector.cql.CassandraConnector

scala> val conf = new
SparkConf().setMaster("local[3]").setAppName("NetworkWordCount")
conf: org.apache.spark.SparkConf = org.apache.spark.SparkConf@e107f51

scala> val ssc = new StreamingContext(conf, Seconds(3))
14/12/10 00:36:16 WARN Utils: Service 'SparkUI' could not bind on port 4040.
Attempting port 4041.
ssc: org.apache.spark.streaming.StreamingContext =
org.apache.spark.streaming.StreamingContext@58ceac1a
```

Notice here that the SparkUI for our Streaming Application could not bind on 4040 (because the Spark Scala Shell is already using it), so the streaming app has binded to the next port #, 4041, instead.

You can quickly check out the UI for this application if you'd like:


NetworkWordCount - Spa
ec2-54-149-142-232.us-west-2.compute.amazonaws.com:4041/stages/
Spark
Stages
Storage
Environment
Executors
Streaming
NetworkWordCount application UI

Spark Stages

Total Duration: 9 s
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 0
Failed Stages: 0

Active Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write

Completed Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write

Anyway, moving back to our scala shell, let's continue building our app:

```
scala> val lines = ssc.socketTextStream("localhost", 9999)
lines: org.apache.spark.streaming.dstream.ReceiverInputDStream[String] =
org.apache.spark.streaming.dstream.SocketInputDStream@3070232f

scala> val words = lines.flatMap(_.split( " "))
```

```
words: org.apache.spark.streaming.dstream.DStream[String] =
org.apache.spark.streaming.dstream.FlatMappedDStream@11b8b62f

scala> val pairs = words.map(word => (word, 1))
pairs: org.apache.spark.streaming.dstream.DStream[(String, Int)] =
org.apache.spark.streaming.dstream.MappedDStream@5a5cb1cb

scala> val wordCounts = pairs.reduceByKey(_ + _)
wordCounts: org.apache.spark.streaming.dstream.DStream[(String, Int)] =
org.apache.spark.streaming.dstream.ShuffledDStream@2040c7d9

scala> wordCounts.saveToCassandra("tinykeyspace", "streaming_test",
SomeColumns("word", "count"))

scala> wordCounts.print()

scala> ssc.start()

scala> 14/12/10 00:38:30 WARN BlockManager: Block input-0-1418171910600
already exists on this machine; not re-adding it
-------------------------------------------
Time: 1418171911000 ms
-------------------------------------------
(two,2)
(one,1)
(three,3)
(four,4)
```

The Scala prompt will keep updating now every 3 seconds. If you're wondering what the BlockManager warning is, here is a quote from T.D. (creator of Spark Streaming) in the following User Mailing list thread:
http://apache-spark-user-list.1001560.n3.nabble.com/streaming-questions-td3281.html

"Spark Streaming is designed to replicate the received data within the machines in a Spark cluster for fault-tolerance. However, when you are running in the local mode, since there is only one machine, the "blocks" of data arent able to replicate. This is expected and safe to ignore in local mode."

Switch to the Cassandra CQL shell and check if the word counts for the 4 words are in the Cassandra Table:

```
cqlsh> SELECT * FROM tinykeyspace.streaming_test;

 word  | count
-------+-------
 three |     3
   one |     1
   two |     2
  four |     4

(4 rows)
```

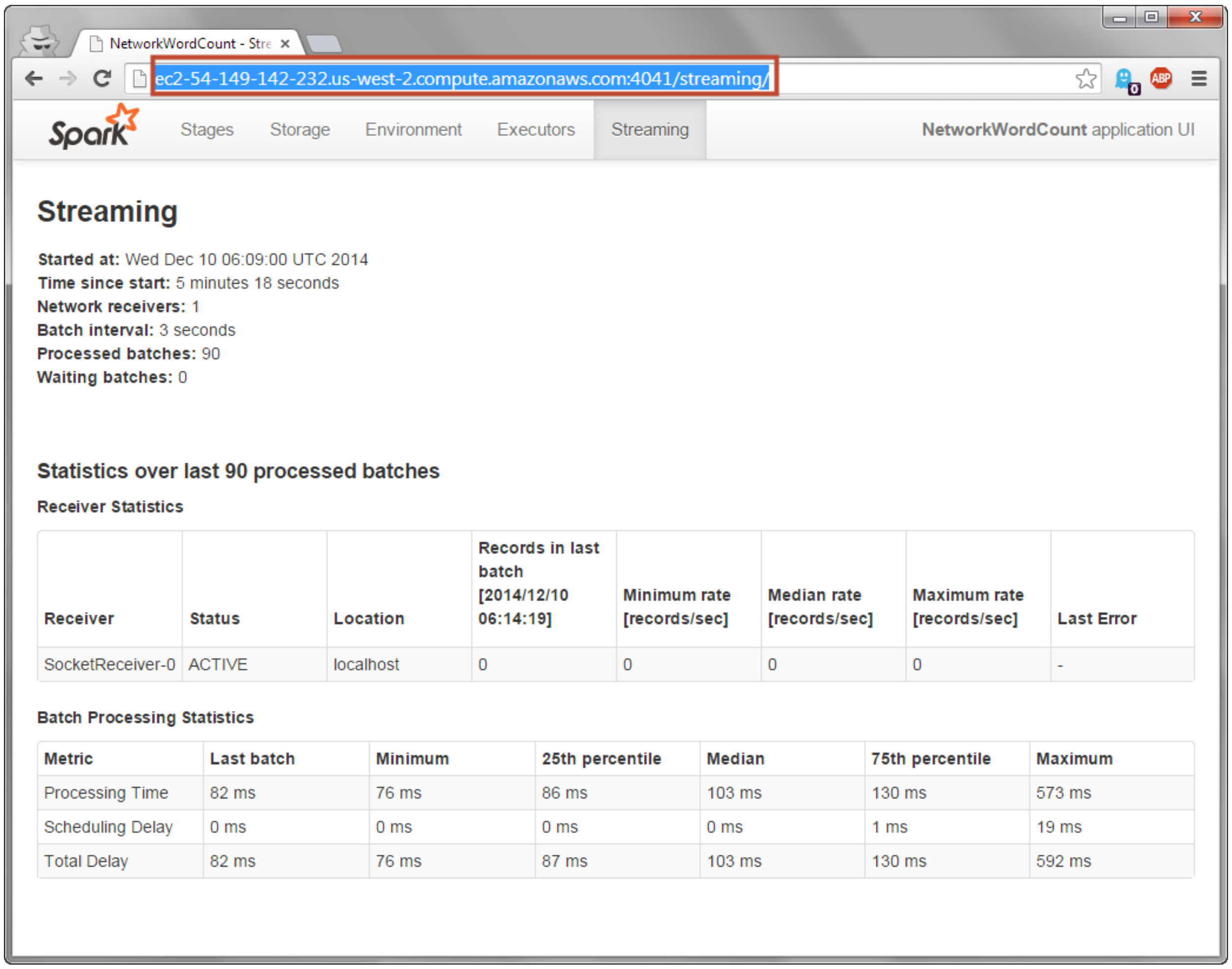
Great. Switch to the linux shell prompt where netcat is running and enter a few more words:

```
dog dog cat cat cat

14/12/10 06:13:07 WARN BlockManager: Block input-0-1418191987400 already
exists on this machine; not re-adding it
-------------------------------------------
Time: 1418191989000 ms
-------------------------------------------
(cat,3)
(dog,2)

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

So far, so good. Continue to the next page...

Now go to the following port 4041/streaming URL to see the Spark Streaming UI:

```
http://<your EC2 instance's public DNS hostname>:4041/streaming/
```


NetworkWordCount - Stre
ec2-54-149-142-232.us-west-2.compute.amazonaws.com:4041/streaming/
Spark
Stages
Storage
Environment
Executors
Streaming
NetworkWordCount application UI

Streaming

Started at: Wed Dec 10 06:09:00 UTC 2014
Time since start: 5 minutes 18 seconds
Network receivers: 1
Batch interval: 3 seconds
Processed batches: 90
Waiting batches: 0

Statistics over last 90 processed batches

Receiver Statistics

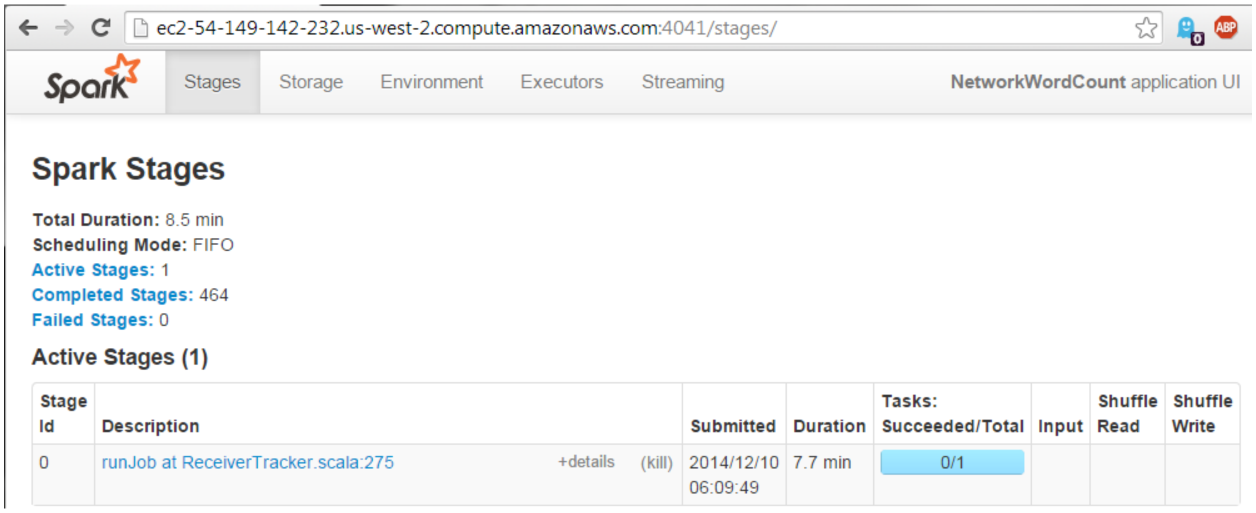
Receiver	Status	Location	Records in last batch [2014/12/10 06:14:19]	Minimum rate [records/sec]	Median rate [records/sec]	Maximum rate [records/sec]	Last Error
SocketReceiver-0	ACTIVE	localhost	0	0	0	0	-

Batch Processing Statistics

Metric	Last batch	Minimum	25th percentile	Median	75th percentile	Maximum
Processing Time	82 ms	76 ms	86 ms	103 ms	130 ms	573 ms
Scheduling Delay	0 ms	0 ms	0 ms	0 ms	1 ms	19 ms
Total Delay	82 ms	76 ms	87 ms	103 ms	130 ms	592 ms

Notice that the UI shows "Processed batches: #" depending on how long it has been running. Each batch is a 3 second window. You can also see statistics about the app in this UI.

Switch to the Spark Stages UI:
`http://<your EC2 instance's public DNS hostname>:4041/stages/`


ec2-54-149-142-232.us-west-2.compute.amazonaws.com:4041/stages/
Spark
Stages
Storage
Environment
Executors
Streaming
NetworkWordCount application UI

Spark Stages

Total Duration: 8.5 min
Scheduling Mode: FIFO
Active Stages: 1
Completed Stages: 464
Failed Stages: 0

Active Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
0	runJob at ReceiverTracker.scala:275 +details (kill)	2014/12/10 06:09:49	7.7 min	0/1			

Notice that there is is long-running receiver here that is continuously looking for fresh data on port 9999.

Next switch to the linux shell prompt where netcat is running and just the following word:
`four`

What do you think will happen now? Will the row record in Cassandra keeping track of the word four increase from 4 to 5?

Switch to the Cassandra CQL shell and check if the word counts for the 4 words are in the Cassandra Table:

```
cqlsh> SELECT * FROM tinykeyspace.streaming_test;

 word  | count
-------+-------
 three |     3
   one |     1
   two |     2
   cat |     3
   dog |     2
  four |     1

(6 rows)
```

Hmm, maybe not what you were expecting… increase of increasing the count value for row "four", we instead replaced the original count with the new count of "1".

We could have used a 'counter column' type in Cassandra to increase the value of this column by whatever new counts we are seeing every 3 seconds, but this is a bit beyond the scope of this lab. I suggest looking at the "CQL for Cassandra" PDF from Datastax and find the "Using a counter" column for a simple explanation of how to use and update counter columns.

Let's demonstrate one last thing in Spark SQL. What happens if we break the incoming network pipe at port 9999?

Next switch to the linux shell prompt where netcat is running and hit the following keyboard shortcut to exit out of netcat:

```
<hit CTRL + C>
^C
```

Switch to the Spark shell prompt and you should see the following error:

```
-------------------------------------------
Time: 1418192745000 ms
-------------------------------------------

14/12/10 06:25:46 WARN ReceiverSupervisorImpl: Restarting receiver with delay
2000 ms: Error connecting to localhost:9999
java.net.ConnectException: Connection refused
        at java.net.PlainSocketImpl.socketConnect(Native Method)
        at
java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:339)
```

Switch back to the linux shell prompt where netcat was running but is now terminated and relaunch netcat and enter a couple of words:

```
ubuntu@ip-10-0-1-28:~$ nc -lk 9999
clouds clouds
```

Switch to the Spark shell prompt and you should see the new output with (clouds, 2)

```
14/12/10 06:25:56 WARN BlockManager: Block input-0-1418192756600 already
exists on this machine; not re-adding it
-------------------------------------------
Time: 1418192757000 ms
-------------------------------------------
(clouds,2)
```

Go ahead and terminate the Spark Streaming App and the Scala Shell REPL with CTRL + C:

```
<hit CTRL + C>
ubuntu@ip-10-0-1-28:~$
```

You should now be back at the Ubuntu shell.

Switch to the Cassandra CQL shell and gracefully quit:

```
cqlsh> quit;
ubuntu@ip-10-0-1-28:~$
```

This concludes the Spark + Cassandra integration lab!