

Big Data Science In Spark

Course 73005-EL Labs and Exercises

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017

Big Data Science With Spark

Module 1: Introduction To Big Data No Labs

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017

Big Data Science With Spark

Module 2: Labs for Hadoop Cluster Architecture

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017



A TERADATA COMPANY

Hadoop Architecture Labs – Amazon Web Services Elastic Map Reduce Version

The labs in this directory are designed for working within an Amazon Web Services (AWS) Elastic Map Reduce (EMR) Cluster, which is currently at version 5.8.1. While the general principles are the same when using an Amazon Web Services Elastic Map Reduce cluster instead of Think Big Academy's vagrant instances, the specific details and file locations are different.

These labs familiarize you with four major components of a Hadoop cluster and how to move data among them:

- The local file system
- The Hadoop Distributed File System (HDFS)
- YARN, the cluster operating system
- The Hive Warehouse

Preparing Your Environment

Tools You'll Need On Your Local Computer

Much of the work in this course will be done at the Linux command line on your cluster. That means you will require a three pieces of software on your local laptop or PC:

1. A Unix or Linux terminal emulator.
2. An ssh client.
3. A modern web browser.

piece of secure software that allows you to connect to your cluster over the Internet. The standard tool we use for that is the Unix or Linux Secure Shell or `ssh` .

Macintosh users already have `ssh` installed as part of their operating systems.

Saving Information About Your Cluster

Your instructor has given you an IP address for your Elastic Map Reduce cluster. You should write that down and keep it handy. Your instructor should also have given you a cryptographic key for your cluster.

For the purposes of the labs in this course, we're going to create a shell environment variable called `cluster` that has your cluster's IP address. We're also going to assume you have been given a cryptographic private key called `crypto.pem` which you can use to access your cluster.

First, you

The majority of free disk space on your EMR cluster is in two mounted file systems, `/mnt` and `/mnt1`. We'll be doing most of our work in the local file system on `/mnt`.

Unarchiving Your Data

You should have been given a zip file of the class materials, which we will assume here is named `hadoop-class.zip`. This file should have been placed in one of your AWS Elastic Map Reduce partitions, `/mnt`. You should unzip that file into a new directory called `hadoop-class` and list its contents using the following commands

```
cd /mnt
unzip sparkclass.zip
cd sparkclass
ls
```

You should see the following files in the file listing.

data	exercises	handouts	images	slides
------	-----------	----------	--------	--------

You are now ready to start Lab 1 on AWS



Hadoop Architecture Labs

Lab 1: Examining Nodes On The Cluster

At this point, YARN and the Hadoop Distributed File System (HDFS) are already running on the control node of your cluster. Both of these services have command line and Web interfaces. Let's look at both.

First, connect to the command line on your edge node using `ssh`. However, because your virtual machine is running under vagrant, you'll use the `vagrant ssh` command instead of ordinary `ssh`.

```
vagrant ssh edge
```

You should see output that looks like the following:

```
Last login: Sun Aug 20 18:06:02 2017 from 10.0.2.2
[vagrant@edge ~]$
```

From this point on, we'll simply show the commands we type (as indicated by the `[vagrant@edge vagrant]$` prompt) and the result in the same code block. See what version of yarn you are running using the command `yarn version` and then `yarn application -list` to list all the applications.

```
[vagrant@edge ~]$ yarn version
Hadoop 2.6.0-cdh5.4.2
Subversion http://github.com/cloudera/hadoop -r 15b703c8725733b7b2813d2325659eb7d57e7a3f
Compiled by jenkins on 2015-05-20T00:03Z
Compiled with protoc 2.5.0
From source with checksum de74f1adb3744f8ee85d9a5b98f90d
This command was run using /usr/lib/hadoop/hadoop-common-2.6.0-cdh5.4.2.jar
[vagrant@edge ~]$ yarn application -list
17/08/22 17:42:51 INFO client.RMProxy: Connecting to ResourceManager at control/192.168.33.10:8020
Total number of applications (application-types: [] and states: [SUBMITTED, ACCEPTED, RUNNING])
Application-Id      Application-Name      Application-Type      User
[vagrant@edge ~]$
```

We can get help with all the yarn commands by typing `yarn -help`

```
[vagrant@edge ~]$ yarn -help
Usage: yarn [--config confdir] COMMAND
where COMMAND is one of:
    resourcemanager -format-state-store  deletes the RMStateStore
    resourcemanager                        run the ResourceManager
    nodemanager                          run a nodemanager on each slave
    timelineserver                      run the timeline server
    rmadmin                             admin tools
    version                             print the version
    jar <jar>                           run a jar file
    application                          prints application(s)
                                         report/kill application
    applicationattempt                  prints applicationattempt(s)
                                         report
    container                           prints container(s) report
    node                                prints node report(s)
    queue                               prints queue information
    logs                                dump container logs
    classpath                           prints the class path needed to
                                         get the Hadoop jar and the
                                         required libraries
    daemonlog                           get/set the log level for each
                                         daemon

or
    CLASSNAME                           run the class named CLASSNAME
Most commands print help when invoked w/o parameters.
[vagrant@edge ~]$
```


We can also list the nodes in our cluster with `yarn node -list -all` and query the status of a node using `yarn node -status nodename` as shown below.

```
[vagrant@edge ~]$ yarn node -list -all
17/08/23 15:42:17 INFO client.RMProxy: Connecting to ResourceManager at control/192.168.33.10:8088
Total Nodes:1
      Node-Id      Node-State Node-Http-Address  Number-of-Running-Containers
control:44668      RUNNING      control:8042              0
[vagrant@edge ~]$ yarn node -status control:44668
17/08/23 15:42:35 INFO client.RMProxy: Connecting to ResourceManager at control/192.168.33.10:8088
Node Report :
Node-Id : control:44668
Rack : /default-rack
Node-State : RUNNING
Node-Http-Address : control:8042
Last-Health-Update : Wed 23/Aug/17 03:40:52:695UTC
Health-Report :
Containers : 0
Memory-Used : 0MB
Memory-Capacity : 4096MB
CPU-Used : 0 vcores
CPU-Capacity : 4 vcores
Node-Labels :
```

[vagrant@edge ~]\$

Now we'll look at the same information using the Web interface by going to `http://192.168.33.10:8088` or `http://control:8088` . We reference the control node because that's where the Resource Manager is located. In fact, the `yarn` commands we issued above connected to the control node to get their results.

This is what your output should look like on a brand new cluster.



Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
0	0	0	0	0	0 B	4 GB	0 B	0	4	0	1	0	0	0	0

User Metrics for dr.who

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Memory Used	Memory Pending	Memory Reserved	VCores Used	VCores Pending	VCores Reserved
0	0	0	0	0	0	0	0 B	0 B	0 B	0	0	0


Cluster

Tools

Showing 0 to 0 of 0 entries

First Previous Next Last

If we look under the **Cluster** menu on the left and click on **Nodes** , we should see the nodes that are running, just as we did above on the command line



Cluster

About

Nodes

Applications

NEW

NEW SAVING

SUBMITTED

ACCEPTED

RUNNING

FINISHED

FAILED

KILLED

Scheduler

Tools

Nodes of the cluster

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
0	0	0	0	0	0 B	4 GB	0 B	0	4	0	1	0	0	0	0

User Metrics for dr.who

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Memory Used	Memory Pending	Memory Reserved	VCores Used	VCores Pending	VCores Reserved
0	0	0	0	0	0	0	0 B	0 B	0 B	0	0	0

Show 20 entries

Node Labels

Rack

Node State

Node Address

Node HTTP Address

Last health-update

Health-report

Containers

Mem Used

Mem Avail

VCores Used

VCores Avail

Version

/default-rack	RUNNING	control:44668	control:8042	Wed Aug 23 16:22:52 +0000 2017		0	0 B	4 GB	0	4	2.6.0-cdh5.4.2
---------------	---------	---------------	--------------	--------------------------------	--	---	-----	------	---	---	----------------

Showing 1 to 1 of 1 entries

First Previous 1 Next Last

Finally, if we click on the entry control:8042 under **Node HTTP Address** , we'll get detailed information of our one control node:



ResourceManager

NodeManager

Node Information

List of Applications

List of Containers

Tools

NodeManager information

Total Vmem allocated for Containers

8.40 GB

Vmem enforcement enabled

false

Total Pmem allocated for Container

4 GB

Pmem enforcement enabled

true

Total VCoers allocated for Containers

4

NodeHealthyStatus

true

LastNodeHealthTime

Wed Aug 23 16:24:52 UTC 2017

NodeHealthReport

Node Manager Version:

2.6.0-cdh5.4.2 from 15b703c8725733b7b2813d2325659eb7d57e7a3f by jenkins source checksum e7a085479aa1989b5cecfabea403549 on 2015-05-20T00:09Z

Hadoop Version:

2.6.0-cdh5.4.2 from 15b703c8725733b7b2813d2325659eb7d57e7a3f by jenkins source checksum de74f1adb3744f8ee85d9a5b98f90d on 2015-05-20T00:03Z

This step concludes this lab.



Hadoop Architecture Labs

Lab 2: Querying HDFS

You can also access the Hadoop Distributed File System (HDFS) from both the command line and from the Web. On the command line, we use the `hdfs` command, which has sub-commands just like the `yarn` command did.

Start by asking for help with `hdfs -help`

```
[hadoop@ip-172-31-57-176 mnt]$ hdfs -help
```

```
Usage: hdfs [--config confdir] COMMAND
```

```
where COMMAND is one of:
```

dfs	run a filesystem command on the file systems supported in Hadoop.
namenode -format	format the DFS filesystem
secondarynamenode	run the DFS secondary namenode
namenode	run the DFS namenode
journalnode	run the DFS journalnode
zkfc	run the ZK Failover Controller daemon
datanode	run a DFS datanode
dfsadmin	run a DFS admin client
haadmin	run a DFS HA admin client
fsck	run a DFS filesystem checking utility
balancer	run a cluster balancing utility
jmxget	get JMX exported values from NameNode or DataNode.
mover	run a utility to move block replicas across storage types
oiv	apply the offline fsimage viewer to an fsimage
oiv_legacy	apply the offline fsimage viewer to an legacy fsimage
oev	apply the offline edits viewer to an edits file
fetchdt	fetch a delegation token from the NameNode
getconf	get config values from configuration
groups	get the groups which users belong to
snapshotDiff	diff two snapshots of a directory or diff the current directory contents with a snapshot
lsSnapshottableDir	list all snapshottable dirs owned by the current user Use -help to see options
portmap	run a portmap service
nfs3	run an NFS version 3 gateway
cacheadmin	configure the HDFS cache
crypto	configure HDFS encryption zones
storagepolicies	list/get/set block storage policies
version	print the version

```
Most commands print help when invoked w/o parameters.
```

```
[hadoop@ip-172-31-57-176 mnt]$
```

Begin by printing out the pathname for your local working directory using the `pwd` command, followed by listing its contents using `ls`. Your home directory at `/home/vagrant` should be empty. Then list the contents of your current HDFS directory using the command `hdfs dfs -ls`. You should see different results.

```

[hadoop@ip-172-31-57-176 mnt]$ pwd
/home/vagrant
[hadoop@ip-172-31-57-176 mnt]$ ls
[hadoop@ip-172-31-57-176 mnt]$ hdfs dfs -ls
Found 1 items
drwxr-xr-x   - vagrant supergroup          0 2017-08-20 17:29 .sparkStaging
[hadoop@ip-172-31-57-176 mnt]$

```

Note that your local edge node directory is located at `/home/vagrant` . Your HDFS home directory, on the other hand, is located at `/user/vagrant` . It's important to remember that these are two different file systems: the first is on your local edge node file system, and the second is in your hadoop HDFS file system.

That last directory is your HDFS home directory located at `/user/vagrant` . Now list the contents of `/` , `/user` , and `/user/vagrant` using the `hdfs dfs -ls` command.

```

[hadoop@ip-172-31-57-176 mnt]$ hdfs dfs -ls /
Found 4 items
drwxr-xr-x   - hdfs supergroup              0 2017-08-20 20:39 /rstudio
drwxrwxrwt   - hdfs supergroup              0 2017-08-20 17:28 /tmp
drwxrwxrwt   - hdfs supergroup              0 2017-08-20 17:22 /user
drwxrwxrwt   - hdfs supergroup              0 2017-08-20 17:22 /var
[vagrant@edge ~]$ hdfs dfs -ls /user
Found 2 items
drwxrwxrwt   - hdfs   supergroup            0 2017-08-20 17:22 /user/hive
drwxrwxrwt   - vagrant supergroup            0 2017-08-20 17:28 /user/vagrant
[hadoop@ip-172-31-57-176 mnt]$ hdfs dfs -ls /user/vagrant
Found 1 items
drwxr-xr-x   - vagrant supergroup            0 2017-08-20 17:29 /user/vagrant/.sparkStaging
[hadoop@ip-172-31-57-176 mnt]$

```

Note that the final result there is the same as we got using `hdfs dfs -ls` with no argument.

We can accomplish the same tasks using the Web by connecting to the control node on port 50070. If you connect to `http://192.168.33.10:50070` or `http://control:50070` , you should see the NameServer Web interface shown below.

Hadoop
Overview
Datanodes
Snapshot
Startup Progress
Utilities -

Overview

'control:8020' (active)

Started:	Tue Aug 22 17:30:28 UTC 2017
----------	------------------------------

Version:	2.6.0-cdh5.4.2, r15b703c8725733b7b2813d2325659eb7d57e7a3f
Compiled:	2015-05-20T00:03Z by jenkins from Unknown
Cluster ID:	CID-45f388c7-7abf-4def-9b09-659273a032e7
Block Pool ID:	BP-1371444972-192.168.33.10-1503249719043

Summary

Security is off.
Safemode is off.
252 files and directories, 161 blocks = 413 total filesystem object(s).
Heap Memory used 131.61 MB of 291 MB Heap Memory. Max Heap Memory is 889 MB.
Non Heap Memory used 46.81 MB of 47.69 MB Committed Non Heap Memory. Max Non Heap Memory is -1 B.

Configured Capacity:	37.87 GB
DFS Used:	2.23 GB
Non DFS Used:	5.65 GB
DFS Remaining:	29.99 GB
DFS Used%:	5.89%
DFS Remaining%:	79.19%
Block Pool Used:	2.23 GB
Block Pool Used%:	5.89%
DataNodes usages% (Min/Median/Max/stdDev):	5.89% / 5.89% / 5.89% / 0.00%
Live Nodes	1 (Decommissioned: 0)
Dead Nodes	0 (Decommissioned: 0)
Decommissioning Nodes	0
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion	0
Block Deletion Start Time	8/22/2017, 1:30:28 PM

NameNode Journal Status

Current transaction ID: 1190

Journal Manager	State
FileJournalManager(root=/var/lib/hadoop-hdfs/cache/hdfs/dfs/name)	EditLogFileOutputStream(/var/lib/hadoop-hdfs/cache/hdfs/dfs/name/current/edits_inprogress_00000000000000001190)

NameNode Storage

Storage Directory	Type	State
/var/lib/hadoop-hdfs/cache/hdfs/dfs/name	IMAGE_AND_EDITS	Active

Notice the Menus at the top of the screen that allows you to drill down into the DataNodes where the data actually lives. If you select the `Utilities` pull-down menu at the upper right, you can select `Browse the file system` and see the screen below.

Hadoop

Overview

Datanodes

Snapshot

Startup Progress

Utilities

Browse Directory

/

Go!

Permission	Owner	Group	Size	Replication	Block Size	Name
drwxr-xr-x	hdfs	supergroup	0 B	0	0 B	rstudio
drwxrwxrwt	hdfs	supergroup	0 B	0	0 B	tmp
drwxrwxrwt	hdfs	supergroup	0 B	0	0 B	user
drwxrwxrwt	hdfs	supergroup	0 B	0	0 B	var

Hadoop, 2014.

If you click on the `user` directory, you can then select `vagrant` and see the listing of the same file you saw in the command line example.

Browse Directory

/user/vagrant

Go!

Permission	Owner	Group	Size	Replication	Block Size	Name
drwxr-xr-x	vagrant	supergroup	0 B	0	0 B	.sparkStaging

Hadoop, 2014.

Note that unlike at the command line, you don't have utilities for modifying the HDFS file system from the NameNode Web interface; you can only browse the file system.

This step concludes this lab.



A TERADATA COMPANY

Hadoop Architecture Labs

Lab 3: Loading Data Into HDFS

Begin by connecting to the command line on your cluster using the `vagrant ssh` command.

```
ssh -i ~/sparkcourse.pem hadoop@$MY_IP_ADDRESS
```

You should see output that looks like the following:

```
Last login: Sun Aug 20 18:06:02 2017 from 10.0.2.2
[hadoop@ip-172-31-57-176 mnt]$
```

Now change directories to `/vagrant` and examine the contents of that directory:

```
cd /mnt/sparkclass
ls
```

You should see the following contents that live in the edge node's local file system. Your formatting should be prettier than what's shown below due to this document's line length limits.

```
data exercises handouts images slides
[hadoop@ip-172-31-57-176 sparkclass]$
```

Now look at the contents of the HDFS file system current directory using the `hdfs dfs -ls` command. You'll see different results:


```
[hadoop@ip-172-31-57-176 sparkclass]$ hdfs dfs -ls
Found 1 items
drwxr-xr-x  - vagrant supergroup          0 2017-08-20 17:29 .sparkStaging
[hadoop@ip-172-31-57-176 sparkclass]$
```

Let's examine the root directory of HDFS. From this point on, we'll simply show the commands we type (as indicated by the `[vagrant@edge vagrant]$` prompt) and the result in the same code block.

```
[hadoop@ip-172-31-57-176 sparkclass]$ hdfs dfs -ls /
Found 4 items
drwxr-xr-x  - hdfs hadoop          0 2017-08-24 18:36 /apps
drwxrwxrwt  - hdfs hadoop          0 2017-08-24 18:39 /tmp
drwxr-xr-x  - hdfs hadoop          0 2017-08-24 18:36 /user
drwxr-xr-x  - hdfs hadoop          0 2017-08-24 18:36 /var
[hadoop@ip-172-31-57-176 sparkclass]$
```

We're now going to add the contents of the local data directory to the HDFS directory `/data`. We do this using the `hdfs dfs -put` command and then examine the `/data` directory with `hdfs dfs -ls /data` command. The first command will take a minute or two to copy roughly 1.5GB of data.

```
[hadoop@ip-172-31-57-176 sparkclass]$ hdfs dfs -put data /
[hadoop@ip-172-31-57-176 sparkclass]$ hdfs dfs -ls /data
Found 22 items
-rw-r--r--   1 hadoop hadoop      10244 2017-08-29 11:15 /data/.DS_Store
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:15 /data/bag-o-words
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:15 /data/big-r-data
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:15 /data/data
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:15 /data/dividends
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:15 /data/dividends-flat
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:15 /data/employees
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:15 /data/employees-pig
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:16 /data/ge
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:16 /data/logs
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:16 /data/ml-100k
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:16 /data/movies
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:16 /data/orders-pig
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:16 /data/python
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:16 /data/shakespeare
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:16 /data/shakespeare_wc
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:16 /data/spark-resources-data
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:16 /data/sparkml-data
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:16 /data/stocks
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:16 /data/stocks-flat
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:16 /data/twitter
drwxr-xr-x   - hadoop hadoop         0 2017-08-29 11:16 /data/twitter-pig
[hadoop@ip-172-31-57-176 sparkclass]$
```

These directories contain source datasets that we can use throughout this Hadoop course. However, not many of these datasets are large ones. To allow us to give our Hadoop cluster a bit more exercise, we'll load some multi-million line datasets regarding US airline flights that we've downloaded from the FAA. These datasets are in the local file directory `flightdata`. We'll add that dataset to the `/data` directory on HDFS using another `put` command.

One way you could do that is as follows.

```
[hadoop@ip-172-31-57-176 sparkclass]$ hdfs dfs -mkdir /data/flightdata
[hadoop@ip-172-31-57-176 sparkclass]$ ls flightdata
aircraft-registrations  parquet-trimmed  tickets
[hadoop@ip-172-31-57-176 sparkclass]$ hdfs dfs -put flightdata/* /data/flightdata
```

Just to demonstrate how we would remove a dataset, we'll delete that directory and its contents from HDFS and load the data in a simpler way.

```
[hadoop@ip-172-31-57-176 sparkclass]$ hdfs dfs -rm -R /data/flightdata
17/08/23 20:40:03 INFO fs.TrashPolicyDefault: Namenode trash configuration: Deletion interval :
Deleted /data/flightdata
[hadoop@ip-172-31-57-176 sparkclass]$ hdfs dfs -put flightdata /data
```

This step concludes the lab.

Big Data Science With Spark

Module 3: Spark Hive Fundamentals No Lab Notes

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017

Big Data Science With Spark

Module 4: Big Data Bottlenecks No Labs

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017

Big Data Science With Spark

Module 5: Spark Architecture And Concepts No Labs

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017

Big Data Science With Spark

Module 6: Spark In Scala Labs and Exercises

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017



SparkML

Lab 01: Creates Personal Movie Ratings

This lab's only goal is to create a file of your personal ratings on a scale of 1 to 5 for the following movies:

- Toy Story (1995)
- Independence Day (a.k.a. ID4) (1996)
- Dances with Wolves (1990)
- Star Wars: Episode VI - Return of the Jedi (1983)
- Mission: Impossible (1996)
- Ace Ventura: Pet Detective (1994)
- Die Hard: With a Vengeance (1995)
- Batman Forever (1995)
- Pretty Woman (1990)
- Men in Black (1997)
- Dumb & Dumber (1994)

Most people today will find these movies quite old. They are chosen because they map to movies in the ratings database we'll be using to build our recommendation engine.

To rate your movies, you should perform the following steps on your cluster:

```
cd /mnt/sparkclass/exercises/SparkML-In-Depth
./rateMovies.py
```

The script will create a data file called `personalRatings.txt`.

Once you've rated those movies, you have completed this lab.



A TERADATA COMPANY

Spark in Scala

Lab 01: Spark Basics Walkthrough

This file gets you started in Spark in Scala.

Turn down the logging level

By default Spark versions before 2.0, many of Spark's logging parameters are set to `INFO` leading to extremely verbose messages printed to the console. In fact, if you don't turn these down, the prompt will soon scroll off the screen!

On Amazon EMR

If you are running a version of Spark before 2.0, edit `/usr/lib/spark/conf/log4j.properties` on the cluster with your text editor of choice (`emacs` , `vi` , `nano`) to selectively change `INFO` entries to `WARN` . To change them all easily:

```
sed -i.bak 's/INFO/WARN/' /usr/lib/spark/conf/log4j.properties
```

On Vagrant

If you are running a version of Spark before 2.0, edit `/vagrant/latest-spark/conf/log4j.properties` on the cluster with your text editor of choice (`emacs` , `vi` , `nano`) to selectively change `INFO` entries to `WARN` . To change them all easily:

```
sed -i.bak 's/INFO/WARN/' /vagrant/latest-spark/conf/log4j.properties
```

Launch Spark's interactive environment

Begin by starting the Spark shell from the Linux Shell command line

```
spark-shell --master yarn-client          // start in client mode
```

At the `scala>` prompt, examine the Spark Context you were handed when you started Spark by typing `sc`. Throughout this lab, we'll show what you type immediately after the `scala>` prompt followed by what the spark-shell returns.

```
scala> sc
res7: org.apache.spark.SparkContext = org.apache.spark.SparkContext@2bc52b08

scala> sc.isLocal
res8: Boolean = false

scala>
```

Note that tab completion works by default in the Scala shell to show which methods and fields are available on the SparkContext `sc`. Try some of them out to explore the SparkContext:

```
scala> sc.appName
res11: String = Spark shell

scala> sc.getExecutorMemoryStatus
res12: scala.collection.Map[String,(Long, Long)] = Map(172.31.57.176:45326 -> (434582323,434582323), 172.31.57.176:45326 -> (434582323,434582323))

scala> sc.hadoopConfiguration
res13: org.apache.hadoop.conf.Configuration = Configuration: core-default.xml, core-site.xml, hdfs-default.xml, hdfs-site.xml, mapred-default.xml, mapred-site.xml, yarn-default.xml, yarn-site.xml

scala> sc.version
res14: String = 2.2.0

scala> spark
res15: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@50aa482f
```

Play with an RDD

Now create a simple, small RDD by hand:

```
val smallprimes = sc.parallelize(Array(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97))
```

Try some basic operations on the RDD. Note that a function takes no arguments, no parentheses are needed after the function name.

```
scala> smallprimes.count
res23: Long = 25

scala> smallprimes.min
res24: Int = 2

scala> smallprimes.max
res25: Int = 97

scala> smallprimes.sum
res26: Double = 1060.0

scala> smallprimes.collect
res27: Array[Int] = Array(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97)
```

Read data from HDFS

A more realistic scenario is to create an RDD from data read from disk. Spark can natively access HDFS and S3 in addition to the local file system. Try reading in the stock quote data from HDFS:

```
val rdd = sc.textFile("hdfs:///data/stocks-flat/input")
rdd.first
rdd.count
```

Simple Transformations and Actions

Let's first filter the data set so we only see AAPL records:

```
val aapl = rdd.filter( line => line.contains("AAPL") )  
aapl.count
```



A TERADATA COMPANY

Spark in Scala

First Program In Spark (02-Spark-First-Program)

We saw in our walkthrough that we count the number of stock quotes for AAPL using code that looks like this:

```
val rdd = sc.textFile("hdfs:///data/stocks-flat/input")
val aapl = rdd.filter( line => line.contains("AAPL") )
aapl.count
```

Try using that pattern to count the number of lines in the file /data/shakespeare/input that contain the word "king". You'll want to use the toLowerCase function before the contains function to ensure all the text is lower case.

One solution looks like this:

```
val rdd = sc.textFile("hdfs:///data/shakespeare/input")
val kings = rdd.filter( line => line.toLowerCase().contains("king"))
kings.count
```

You should have gotten 4773 lines that reference a king in all of Shakespeare. That's almost 3% of the 175,376 lines in all of Shakespeare's plays and poems.



A TERADATA COMPANY

Spark in Scala

Spark Transformations Program

In this lab, you should type the following into your spark-shell:

```
val fib = Array(1, 2, 3, 5, 8, 13, 21, 34)
```

Then, using the REPL, use both named functions and anonymous functions to do the following:

- Compute all the squares
- Return those squares that are divisible by 3

You'll want to use the `.map` and `.filter` transformations on `fib` to invoke your functions

The answer is

```
// Compute all the squares and sum all the values provided
```

```
scala> def square(x: Int) = x * x  
square: (x: Int)Int
```

```
scala> def divisible(x: Int) = (x % 3 == 0)  
divisible: (x: Int)Boolean
```

```
scala> fib.map(square).filter(divisible)  
res18: Array[Int] = Array(9, 441)
```

```
scala> fib.map(x => x*x).filter(_ % 3 == 0)  
res19: Array[Int] = Array(9, 441)
```




A TERADATA COMPANY

Spark in Scala

Spark Functions, Anonymous and Otherwise

Simple operations in Scala

Try typing the following lines at the spark-shell:

```
1 + 1           // Should be Int = 2
1.0 + 1         // Should be Double = 2.0
"This is a string" // String = This is a string
```

Allocate a mutable variable using `var`

```
var x = 2 + 2
println(x)
x = 3 + 3
println(x)
```

However, an immutable, which is created using `val`, can't be changed once it is created.

```
val x = 2 + 2
println(x)
x = 3 + 3
println(x)
```

You should see

```
<console>:21: error: reassignment to val
    x = 3 + 3
      ^
```

Function Definitions In Spark

In this little exercise, we'll define a simple function in the most straightforward, obvious way and then show how Scala's ability to infer context and types allows us to be more succinct.

Let's start by defining a simple add function and testing it.

```
def add(x:Int, y:Int):Int = {      // Takes two Int arguments and
                                  // returns an Int
    return x + y
}
println(add(42,13))
```

You should see 55 as your result.

Now let's shorten things a bit. Let's use implicit typing.

```
// Implicit typing and return
def add(x:Int, y:Int) = {          //result type is inferred
    x + y                          // "return" keyword is optional
}
```

Further, curly braces are optional on single line blocks, so our function now just can be:

```
def add(x:Int, y:Int) = x + y
```

Not only is it shorter, but it's more readable too.

Anonymous Functions or Function Literals

First, create a named function that simply prepends "Hello " to a name. Then apply that greeting to a list of names using the `map` function.

```
scala> def greeting(x: String) = "Hello " + x
greeting: (x: String)String

scala> val names = List("Joe", "Mary", "Barbara")
names: List[String] = List(Joe, Mary, Barbara)

scala> names.map(greeting)
res2: List[String] = List(Hello Joe, Hello Mary, Hello Barbara)
```

Because we can infer the types, we can use the underscore shortcut for this function definition.

```
scala> names.map("Hello " + _)
res7: List[String] = List(Hello Joe, Hello Mary, Hello Barbara)
```

Now let's get rid of the name greeting by using an anonymous function.

```
scala> names.map((x: String) => "Hello " + x)
res3: List[String] = List(Hello Joe, Hello Mary, Hello Barbara)
```

Define maximize to compute the maximum of two integers using a function literal instead of def. Function literals are also referred to as anonymous or lambda functions

```
scala> val maximize = (a: Int, b: Int) => if (a > b) a else b
maximize: (Int, Int) => Int = <function2>

scala> maximize(5, 3)
res4: Int = 5
```

Define doubler as an immutable variable whose value is a function. Note we aren't using def for our function definition.

```
scala> val doubler = (x: Int) => x * 2 // This assigns a function literal to doubler
doubler: Int => Int = <function1>

scala> doubler(4)
res5: Int = 8
```

OK, now use the `_` placeholder for the doubler function argument. Read this as "doubler is defined as a

function literal that takes an integer and returns an integer; the function definition value multiplies its argument times 2.

```
val doubler: (Int) => Int = _ * 2
doubler(4)
```

Doubler's value is a function literal. We can now pass this value to other functions that take a function as an argument

Many ways exist to define functions and methods. Here are several.

```
val even = (i: Int) => { i % 2 == 0 } // explicit long form
val even: (Int) => Boolean = i => { i % 2 == 0 }
val even: Int => Boolean = i => ( i % 2 == 0 )
val even: Int => Boolean = i => i % 2 == 0
val even: Int => Boolean = _ % 2 == 0 // _ means first argument

// implicit result approach
val add = (x: Int, y: Int) => { x + y }
val add = (x: Int, y: Int) => x + y

// explicit result approach
val add: (Int, Int) => Int = (x,y) => { x + y }
val add: (Int, Int) => Int = (x,y) => x + y
```

We can also use `_` as the argument placeholder for our even number tester.

```
scala> val nums = Array(1, 2, 3, 4, 5)
nums: Array[Int] = Array(1, 2, 3, 4, 5)

scala> nums.map(_ % 2 == 0)
res8: Array[Boolean] = Array(false, true, false, true, false)
```

This may be harder to understand; it's a reducing function similar to that used in MapReduce. This anonymous function adds its two arguments together. The net result is that it sums all the elements of the immutable `nums`.

```
scala> nums.reduceLeft(_ + _) // first plus second argument
res9: Int = 15
```

These literal definitions are incredibly valuable in Spark because many of the Spark operations take functional arguments. Most of these will never be assigned to a variable. For example the following finds list entries that have an "f" in them using an anonymous function/function literal

```
scala> val myList = List("foo", "bar", "fight")
mylist: List[String] = List(foo, bar, fight)

scala> myList.filter(_.contains("f"))
res10: List[String] = List(foo, fight)
```

We could have written that as follows, but it's longer and less clear

```
scala> val myList = List("foo", "bar", "fight")
mylist: List[String] = List(foo, bar, fight)

scala> val ffilter = (s: String) => s.contains("f")           // define a named function to search
ffilter: String => Boolean = <function1>

scala> myList.filter(ffilter)                                // will generate the same result as previous
res11: List[String] = List(foo, fight)
```

First full example: Textsearch

Load error messages from a log into memory, then interactively search for various patterns.

The file `log.txt` has the following 5 lines:

```
ERROR      php: dying for unknown reasons
WARN       dave, are you angry at me?
ERROR      did mysql just barf?
WARN       xylons approaching
ERROR      mysql cluster: replace with spark cluster
```

Our objective is to count all the error messages (not warnings) that have reference *mysql* or *php*.

```

val lines = sc.textFile("hdfs:///data/logs/log.txt")

// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()

// actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()

```

Now Wordcount

The periods in this dataflow pipeline are at the ends of lines so that we can execute this code in the interactive spark-shell without modification.

Type this into spark-shell and see how it works for you.

```

val textFile = sc.textFile("hdfs:///data/shakespeare/input")
val counts = textFile.flatMap(line => line.split(" ")).
    map(word => (word, 1)).
    reduceByKey(_ + _)
counts.saveAsTextFile("hdfs:///tmp/shakespeare-wc-scala")

```

Estimate Pi in Spark

This program generates 100,000 x and y variables between 0 and 1. It then counts the ratio of those that fall within a unit circle over the number of total samples; that value should approximate pi/4.

Try various values of NUM_SAMPLES to see how the computed value and runtimes vary.

```

val NUM_SAMPLES = 100000
val count = sc.parallelize(1 to NUM_SAMPLES).map{i =>
    val x = Math.random()
    val y = Math.random()
    if (x*x + y*y < 1) 1 else 0
}.reduce(_ + _)
println("Pi is roughly " + 4.0 * count / NUM_SAMPLES)

```




Spark in Scala

Spark-Wordcount

This exercise walks through the classic *Wordcount* example in Spark in both Scala and Python.

Before running this or other examples, you should turn down Spark's default logging. See the slides in [Spark-in-Scala](#) or [Spark-in-Python](#) for how to do this.

Run the Spark shell

Scala:

```
$ spark-shell --master yarn-client
```

Python:

```
$ pyspark --master yarn-client
```

Note the Spark context will already be initialized and available to you as `sc`.

Read the data from HDFS

First, let's load in our Shakespeare text and apply a regular expression to split on non-word characters:

Scala:


```
val text = sc.textFile("hdfs:///data/shakespeare/input/")
val words = text.flatMap( line => line.split("\\W+") )
```

Python:

```
text = sc.textFile("hdfs:///data/shakespeare/input/")
words = text.flatMap( lambda line: line.split() )
```

Construct key-value pairs

Generate the "(word, 1)" key-value pairs:

Scala:

```
val kv = words.map( word => (word.toLowerCase(), 1) )
```

Python:

```
kv = words.map( lambda word: (word.lower(), 1) )
```

Group by key and sum

Use `reduceByKey` to group the data by key and apply the addition operator to add the values:

Scala:

```
val totals = kv.reduceByKey( (v1, v2) => v1 + v2 )
```

Python:

```
totals = kv.reduceByKey( lambda v1, v2: v1 + v2 )
```

Execute the job and save the results

Scala:

```
scala> totals.saveAsTextFile("hdfs:///user/hadoop/spark-wc")

scala> totals.take(10)
res17: Array[(String, Int)] = Array((pinnacle,3), (bone,21), (lug,3), (vailing,3), (bombast,4),

scala> words.max()
res18: String = zwaggered
```

Python:

```
>>> totals.saveAsTextFile('hdfs:///user/hadoop/spark-wc-py')

>>> totals.take(10)
[(u'fawn', 14), (u'sending:', 1), (u'mustachio', 1), (u'philadelphos,', 1), (u'protested,', 1),
```

Examine the output on disk

```
$ hadoop fs -cat spark-wc/part-00000 | head -20
(pinnacle,3)
(bone,21)
(lug,3)
(vailing,3)
(bombast,4)
(gaping,11)
(hem,10)
(forsooth,48)
(stinks,1)
(been,738)
(fuller,2)
(jade,16)
(countervail,3)
(jove,98)
(crying,36)
(breath,238)
(battering,3)
(contemptible,5)
(swain,24)
(clients,3)
```

Note the lack of sorting. Spark does not perform the same merge-sort as MapReduce during its shuffles.

Data pipeline syntax

Both Scala and Python allow you to chain function calls making it possible to write both Spark jobs much more compactly:

Scala:

```
val text = sc.textFile("hdfs:///data/shakespeare/input/")
val totals = text.flatMap( line => line.split("\\W+") ).map( word => (word.toLowerCase(), 1) )

totals.saveAsTextFile("hdfs:///user/hadoop/spark-wc2")
```

Python:

```
text = sc.textFile("hdfs:///data/shakespeare/input/")
totals = text.flatMap( lambda line: line.split() ).map( lambda word: (word.lower(), 1) ).reduceByKey(_+_).groupByKey().mapValues(sum)

totals.saveAsTextFile('hdfs:///user/hadoop/spark-wc-py2')
```



SparkSQL in Scala

Lab01: Basic SparkSQL Walkthrough

At this point, we assume that you are running a `spark-shell` on your cluster. You should type the commands after the `scala>` prompt and you should see the outputs shown.

Begin by reading in a dataset. Hint: It's not exactly big data. In fact, it just looks like this:

```
[hadoop@ip-172-31-57-176 sparkclass]$ hdfs dfs -cat /data/spark-resources-data/people.json
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

That said, let's read it in so we have some data to apply SparkSQL to.

```
scala> val df = spark.read.json("/data/spark-resources-data/people.json")
df: org.apache.spark.sql.DataFrame = [age: bigint, name: string]
```

Now that we have people read in as a DataFrame, we can now query it in different ways.

```
// Displays the content of the DataFrame to stdout
df.show()
+----+-----+
| age|   name|
+----+-----+
| null|Michael|
|  30|   Andy|
|  19|  Justin|
+----+-----+
```

```
// Print the schema in a tree format
df.printSchema()
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
+-----+
|   name|
+-----+
|Michael|
|   Andy|
|  Justin|
+-----+

// Select everybody, but increment the age by 1
df.select(df.select($"name", df("age") + 1).show()
+-----+-----+
|   name|(age + 1)|
+-----+-----+
|Michael|    null|
|   Andy|    31|
|  Justin|    20|
+-----+-----+

// Select people older than 21
df.filter(df("age") > 21).show()
+---+-----+
|age|name|
+---+-----+
| 30|Andy|
+---+-----+

// Count people by age
df.groupBy("age").count().show()
+---+-----+
|age|count|
+---+-----+
| 19|    1|
|null|    1|
| 30|    1|
+---+-----+
```

Read A Table From The Hive Warehouse

If you ran the Hive-Tables2 and Hive-LoadingData queries in the Hive exercises, you should have a Hive table called stocks which has a lot of data loaded into 4 partitions

Remember that you if you read that into a database with your name on it (DB), you have to change to that DB using the `use yourname;` command. The following will simply assume you are using the default Hive database.

```
// Existing Spark session is assumed to be in spark
scala> spark.sql("USE default")           // Select my database
res0: org.apache.spark.sql.DataFrame = []

scala> val stocks = spark.sql("SELECT * FROM STOCKS")
stocks: org.apache.spark.sql.DataFrame = [ymd: string, price_open: float ... 7 more fields]

scala> stocks.show(5) // Will only show the first 5 rows
```

ymd	price_open	price_high	price_low	price_close	volume	price_adj_close	exchg	symbol
2015-06-22	127.49	128.06	127.08	127.61	33833500	127.61	NASDAQ	AAPL
2015-06-19	127.71	127.82	126.4	126.6	54181300	126.6	NASDAQ	AAPL
2015-06-18	127.23	128.31	127.22	127.88	35241100	127.88	NASDAQ	AAPL
2015-06-17	127.72	127.88	126.74	127.3	32768500	127.3	NASDAQ	AAPL
2015-06-16	127.03	127.85	126.37	127.6	31404000	127.6	NASDAQ	AAPL

```
only showing top 5 rows

scala> stocks.count()
res4: Long = 40547
```

Reading In A Large Dataset

We'd like to read in an FAA dataset that has all the US airline flights between 2010 and 2016. You could do that in SQL with the following command, but I don't recommend it; it has a lot of columns.

```
// This is one way to create a s3flights dataframe.
// val creates3flightsSQL = "CREATE EXTERNAL TABLE s3flights (
  Year INT,
  Quarter INT,
  Month INT,
```

DayOfMonth INT,
DayOfWeek INT,
FlightDate STRING,
UniqueCarrier STRING,
AirlineID INT,
Carrier STRING,
TailNum STRING,
FlightNum INT,
OriginAirportID INT,
OriginAirportSeqID INT,
OriginCityMarketID INT,
Origin STRING,
OriginCityName STRING,
OriginState STRING,
OriginStateFips INT,
OriginStateName STRING,
OriginWac INT,
DestAirportID INT,
DestAirportSeqID INT,
DestCityMarketID INT,
Dest STRING,
DestCityName STRING,
DestState STRING,
DestStateFips INT,
DestStateName STRING,
DestWac INT,
CRSDepTime INT,
DepTime INT,
DepDelay INT,
DepDelayMinutes INT,
DepDel15 INT,
DepartureDelayGroups INT,
DepTimeBlk STRING,
TaxiOut INT,
WheelsOff INT,
WheelsOn INT,
TaxiIn INT,
CRSArrTime INT,
ArrTime INT,
ArrDelay INT,
ArrDelayMinutes INT,
ArrDel15 INT,
ArrivalDelayGroups INT,
ArrTimeBlk STRING,
Cancelled TINYINT,

CancellationCode STRING,
Diverted TINYINT,
CRSElapsedTime INT,
ActualElapsedTime INT,
AirTime INT,
Flights INT,
Distance INT,
 DistanceGroup INT,
CarrierDelay INT,
WeatherDelay INT,
NASDelay INT,
SecurityDelay INT,
LateAircraftDelay INT,
FirstDepTime INT,
TotalAddGTime INT,
LongestAddGTime INT,
DivAirportLandings INT,
DivReachedDest INT,
DivActualElapsedTime INT,
DivArrDelay INT,
DivDistance INT,
Div1Airport STRING,
Div1AirportID INT,
Div1AirportSeqID INT,
Div1WheelsOn INT,
Div1TotalGTime INT,
Div1LongestGTime INT,
Div1WheelsOff INT,
Div1TailNum STRING,
Div2Airport STRING,
Div2AirportID INT,
Div2AirportSeqID INT,
Div2WheelsOn INT,
Div2TotalGTime INT,
Div2LongestGTime INT,
Div2WheelsOff INT,
Div2TailNum STRING,
Div3Airport STRING,
Div3AirportID INT,
Div3AirportSeqID INT,
Div3WheelsOn INT,
Div3TotalGTime INT,
Div3LongestGTime INT,
Div3WheelsOff INT,
Div3TailNum STRING,

```

    Div4Airport STRING,
    Div4AirportID INT,
    Div4AirportSeqID INT,
    Div4WheelsOn INT,
    Div4TotalGTime INT,
    Div4LongestGTime INT,
    Div4WheelsOff INT,
    Div4TailNum STRING,
    Div5Airport STRING,
    Div5AirportID INT,
    Div5AirportSeqID INT,
    Div5WheelsOn INT,
    Div5TotalGTime INT,
    Div5LongestGTime INT,
    Div5WheelsOff INT,
    Div5TailNum STRING
)
STORED AS PARQUET LOCATION 's3://think.big.academy.aws/ontime/parquet'
//val s3flights = hiveContext.sql(creates3flightsSQL)

```

The reason I don't recommend this approach is that SQL requires us to specify all the columns and their types. That information is already in the parquet file, so it seems redundant.

We can take advantage of SparkSQL's intelligence by simply loading the parquet file as a dataframe and letting it infer the table schema.

We happen to already have this file in a slightly trimmed down form in HDFS. It's at

```
hdfs:///data/flightdata/parquet-trimmed
```

```

scala> val s3flights = spark.read.parquet("hdfs:///data/flightdata/parquet-trimmed")
s3flights: org.apache.spark.sql.DataFrame = [year: int, quarter: int ... 62 more fields]

```

Now let's select only some of the fields. Further, we're going to filter it to only be the year 2013. Note that the use of a data pipeline allows Spark to lazily evaluate s3flights and only pull those records that are of interest into memory.

Note that because this is a dataframe, our filter tests MUST use === for equality, not == The good news: this runs pretty fast, much faster than the SQL equivalent.

```
scala> val flights = s3flights.select("year", "month", "dayofmonth", "carrier", "tailnum",
    "actualelapsedtime", "origin", "dest", "deptime", "arrdelayminutes").
    filter(s3flights("year") === 2013)
flights: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [year: int, month: int ... 8

scala> flights.show(5)
17/08/29 19:59:40 WARN Utils: Truncated the string representation of a plan since it was too l
+---+---+-----+-----+---+---+-----+---+---+---+---+
|year|month|dayofmonth|carrier|tailnum|actualelapsedtime|origin|dest|deptime|arrdelayminutes|
+---+---+-----+-----+---+---+-----+---+---+---+---+
|2013|  1|    18|DL|N325US|    184|PHL|MSP|    758|         0|
|2013|  1|    18|DL|N325NB|    172|FLL|LGA|    657|         0|
|2013|  1|    18|DL|N649DL|    190|LGA|ATL|   1657|        24|
|2013|  1|    18|DL|N130DL|    251|SLC|ATL|    953|        43|
|2013|  1|    18|DL|N651DL|    171|BOS|ATL|    711|         0|
+---+---+-----+-----+---+---+-----+---+---+---+---+
only showing top 5 rows

scala> flights.printSchema
root
|-- year: integer (nullable = true)
|-- month: integer (nullable = true)
|-- dayofmonth: integer (nullable = true)
|-- carrier: string (nullable = true)
|-- tailnum: string (nullable = true)
|-- actualelapsedtime: integer (nullable = true)
|-- origin: string (nullable = true)
|-- dest: string (nullable = true)
|-- deptime: integer (nullable = true)
|-- arrdelayminutes: integer (nullable = true)
```

Now let's do some simple operations on this DataFrame. How many flights did each carrier fly in 2013? That's a one-liner:

```
scala> flights.groupBy("carrier").count.show
```

```
+-----+-----+
|carrier|  count|
+-----+-----+
|      UA| 505798|
|      AA| 537891|
|      EV| 748696|
|      B6| 241777|
|      DL| 754670|
|      OO| 626359|
|      F9|  75612|
|      YV| 140922|
|      US| 412373|
|      MQ| 439865|
|      HA|  72286|
|      AS| 154743|
|      FL| 173952|
|      VX|  57133|
|      WN|1130704|
|      9E| 296701|
+-----+-----+
```

Let's now compute average delay by carrier and destination

```
scala> flights.groupBy("carrier", "dest").mean("arrdelayminutes").show
```

```
+-----+-----+-----+
|carrier|dest|avg(arrdelayminutes)|
+-----+-----+-----+
|      DL|STL|      8.348766061594942|
|      DL|MSY|      8.593764258555133|
|      AS|IAH|      8.534246575342467|
|      EV|JAX|     19.50561403508772|
|      EV|LFT|     12.419161676646707|
|      EV|SYR|     18.71390798519302|
|      B6|SRQ|     12.782278481012659|
|      US|ROC|      9.662983425414364|
|      UA|JFK|     14.447821229050279|
|      VX|MCO|      4.666026871401152|
|      VX|LAS|     11.598548621190131|
|      WN|ALB|     14.44125144843569|
|      WN|BWI|     11.236715445573436|
|      OO|TUL|     12.621647058823529|
|      AS|SLC|      4.045955882352941|
|      DL|OAK|      3.1056768558951964|
|      MQ|HSV|     17.594059405940595|
|      US|ORD|     15.945858981533297|
|      WN|MAF|     11.323456790123457|
|      OO|EAU|     13.131884057971014|
+-----+-----+-----+
```

only showing top 20 rows

Now by carrier, destination and origin!

```
scala> val delays = flights.groupBy("carrier", "dest", "origin").mean("arrdelayminutes")
delays: org.apache.spark.sql.DataFrame = [carrier: string, dest: string ... 2 more fields]
```

```
scala> delays.printSchema
```

root

```
|-- carrier: string (nullable = true)
|-- dest: string (nullable = true)
|-- origin: string (nullable = true)
|-- avg(arrdelayminutes): double (nullable = true)
```

Note that we've created a new column named avg(arrdelayminutes) Let's sort this to show the worst flights in terms of average delays in 2013

```
scala> delays.sort(desc("avg(arrdelayminutes)")).show(5)
```

```
+-----+-----+-----+-----+
|carrier|dest|origin|avg(arrdelayminutes)|
+-----+-----+-----+-----+
|      EV|EYW|  MIA|          375.0|
|      EV|GPT|  MSY|          315.0|
|      UA|DEN|  MSN|          285.0|
|      EV|RIC|  PIT|          145.0|
|      EV|AEX|  LFT|          138.0|
+-----+-----+-----+-----+
```

only showing top 5 rows

The worst delays were between Key West and Miami, FL Who knew?

Persisting DataFrames

Now let's look at some timing. Let's time the following operation using the `spark.time` method:

```
spark> spark.time(flights.groupBy("carrier", "dest", "origin").
  mean("arrdelayminutes").
  sort(desc("avg(arrdelayminutes)")).show())
```

```
+-----+-----+-----+
|carrier|dest|origin|avg(arrdelayminutes)|
+-----+-----+-----+
|      EV|EYW|  MIA|          375.0|
|      EV|GPT|  MSY|          315.0|
|      UA|DEN|  MSN|          285.0|
|      EV|RIC|  PIT|          145.0|
|      EV|AEX|  LFT|          138.0|
|      UA|SFO|  MSN|          132.0|
|      YV|ITO|  OGG|          132.0|
|      EV|ATL|  JFK|          128.0|
|      UA|MSN|  ORD|          123.5|
|      UA|STL|  EWR|          121.0|
|      EV|XNA|  LGA|          119.0|
|      OO|IAH|  PBI|          117.0|
|      EV|BTV|  BNA|          114.0|
|      F9|COS|  DEN|          111.0|
|      EV|MEM|  BOS|          110.0|
|      EV|GRK|  SAT|          109.0|
|      OO|PBI|  IAH|          107.0|
|      OO|ORD|  LGA|          107.0|
|      UA|CLE|  IAD|          104.0|
|      UA|LAX|  SEA|          103.0|
+-----+-----+-----+
```

only showing top 20 rows

Time taken: 18743 ms

Now persist the flights dataset by typing `flights.persist()`. Then See how long that command take now.

```
spark> spark.time(flights.groupBy("carrier", "dest", "origin").
  mean("arrdelayminutes").
  sort(desc("avg(arrdelayminutes)")).show())
```

```
+-----+-----+-----+-----+
|carrier|dest|origin|avg(arrdelayminutes)|
+-----+-----+-----+-----+
|      EV|EYW|  MIA|          375.0|
|      EV|GPT|  MSY|          315.0|
|      UA|DEN|  MSN|          285.0|
|      EV|RIC|  PIT|          145.0|
|      EV|AEX|  LFT|          138.0|
|      UA|SFO|  MSN|          132.0|
|      YV|ITO|  OGG|          132.0|
|      EV|ATL|  JFK|          128.0|
|      UA|MSN|  ORD|          123.5|
|      UA|STL|  EWR|          121.0|
|      EV|XNA|  LGA|          119.0|
|      OO|IAH|  PBI|          117.0|
|      EV|BTV|  BNA|          114.0|
|      F9|COS|  DEN|          111.0|
|      EV|MEM|  BOS|          110.0|
|      EV|GRK|  SAT|          109.0|
|      OO|ORD|  LGA|          107.0|
|      OO|PBI|  IAH|          107.0|
|      UA|CLE|  IAD|          104.0|
|      UA|LAX|  SEA|          103.0|
+-----+-----+-----+-----+
```

only showing top 20 rows

Time taken: 16630 ms

Persistence doesn't seem to help much, does it?

Actually it helps more than you think. Remember that Scala uses lazy evaluation for its results. When we told Spark to persist the `flights` DataFrame, it evaluated that lazily; it has no idea whether that DataFrame has been computed yet or not -- it just remembered that it should cache it when it next computes it. Therefore, the second run of the command was the first one that cached `flights`.

If this is the case, we should see substantially better performance if we run the command one more time


```
spark> spark.time(flights.groupBy("carrier", "dest", "origin").
  mean("arrdelayminutes").
  sort(desc("avg(arrdelayminutes)")).show())
```

```
+-----+-----+-----+-----+
|carrier|dest|origin|avg(arrdelayminutes)|
+-----+-----+-----+-----+
|      EV|EYW|  MIA|          375.0|
|      EV|GPT|  MSY|          315.0|
|      UA|DEN|  MSN|          285.0|
|      EV|RIC|  PIT|          145.0|
|      EV|AEX|  LFT|          138.0|
|      UA|SFO|  MSN|          132.0|
|      YV|ITO|  OGG|          132.0|
|      EV|ATL|  JFK|          128.0|
|      UA|MSN|  ORD|          123.5|
|      UA|STL|  EWR|          121.0|
|      EV|XNA|  LGA|          119.0|
|      OO|IAH|  PBI|          117.0|
|      EV|BTV|  BNA|          114.0|
|      F9|COS|  DEN|          111.0|
|      EV|MEM|  BOS|          110.0|
|      EV|GRK|  SAT|          109.0|
|      OO|ORD|  LGA|          107.0|
|      OO|PBI|  IAH|          107.0|
|      UA|CLE|  IAD|          104.0|
|      UA|LAX|  SEA|          103.0|
+-----+-----+-----+-----+
```

only showing top 20 rows

Time taken: 3614 ms

Sure enough, that did the trick. The third time we ran that command, it ran in roughly a fifth of the time taken for the first run.

This step concludes this lab.



SparkSQL in Scala

Lab02: SparkSQL Case Classes and Schemas

This lab will help us understand how to use case classes to define a DataFrame's schema.

Case Classes

We're going to work with a similar file to the one we did in the previous lab. This time the file is at `/data/spark-resources-data/people.txt` and instead of being JSON, it's a comma-delimited file.

To create the schema, we're going to define a case class in Scala that provides names and types for each column.

Please note: Case classes in Scala 2.10, the version of Scala used with Spark versions before 2.0, can support only up to 22 fields. You can use custom classes that implement the Product interface to work around that limit. Alternatively, you may wish to simply upgrade to Spark 2.0 with Scala 2.11, which does not impose this limitation on case classes.

Before we begin our Scala program, we must import a couple of spark libraries so that we can create Spark sessions and convert from RDDs to DataFrames and back again.

```
import org.apache.spark.sql._
import spark.implicits._
```

With that note out of the way, let's define a `Person` case class with two columns, a name and an age.

```
case class Person(name: String, age: Long)
```

Encoders

For those of you who are Object Oriented Programming gurus, you'll be interested to know that Encoders are automatically created for case classes. That means I can create a case class and then immediately convert it to a DataSet or DataFrame like this:

```
scala> val caseClassDF = Seq(Person("Andy", 32)).toDF()
caseClassDF: org.apache.spark.sql.DataFrame = [name: string, age: bigint]

scala> caseClassDF.show()
+-----+-----+
| name | age |
+-----+-----+
| Andy | 32 |
+-----+-----+
```

And in fact, encoders are provided for most common object types. You can use those encoders by importing `spark.implicits._` as we did at the beginning of this lab. That allows us to do implicit mappings from Sequences, Arrays, and Lists to DataFrames and DataSets.

```
val primitiveDS = Seq(1, 2, 3).toDS()
primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)
```

Populating Our Case Class

Once the case class is defined, we can now read in an RDD and use `map` to assign various bits of text to the columns. Then we'll convert our `Person` object to a DataFrame. Finally, we'll register our DataFrame as a temporary SQL table named `people` so that we can query it using—you guessed it—SQL.

Note we've put the dots at the ends of lines instead of the more traditional position at the beginning so that you can simply copy and paste this code into the spark-shell.

```
// Create an RDD of Person objects and register it as a table.
val people = sc.textFile("hdfs:///data/spark-resources-data/people.txt").
  map(_ .split(",")). // split by commas
  map(p => Person(p(0), p(1).trim.toInt)). // first field is the person string, second is an int
  toDF()

people.registerTempTable("people")
```

Now if our data already has a structure that maps to our case class, we can save some time by simply telling the `spark.read` command to read the file as that case class type.

If you recall, we had a file called `people.json` that maps nicely to our case class. Therefore we can just say:

```
val path = "hdfs:///data/spark-resources-data/people.json"
val peopleDS = spark.read.json(path).as[Person]
peopleDS.show()
```

And we now have a People DataFrame that looks like this:

age	name
null	Michael
30	Andy
19	Justin

SQL Statements

SQL statements can be run by using the sql methods provided by sqlContext.

```
val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")
```

The results of SQL queries are DataFrames and support all the normal RDD operations. The columns of a row in the result can be accessed by number.

```
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

That should produce `Name: Justin`

The following statements produce the same output. In the first case we reference the column name by ordinal, while in the second, we reference it by field name.

```
teenagers.map(teenager => "Name: " + teenager(0)).show()
teenagers.map(teenager => "Name: " + teenager.getAs[String]("name")).show()
```

Both statements should produce.

value
Name: Justin

This step concludes the lab.



SparkSQL in Scala

Lab03: SparkSQL Case Classes and Schemas

This lab will demonstrate how Spark SQL can use Hive tables. If you have already created the `employees` and `stocks` Hive tables as part of the Spark Hive Fundamentals module, you may skip directly to the section labeled **Querying Hive Tables from Spark SQL**. However, we recommend you read through this next section anyway because interacting with Hive is a powerful feature of Spark. Further, Spark's intelligence simplifies some aspects of table manipulation over Hive's SQL-based approach.

Creating Hive Tables

The Hive exercises create two tables for later use:

1. `employees` is a very simple table of employees that uses complex data types such as maps and structs
2. `stocks` is a Hive table of roughly 2 million stock prices partitioned by stock exchange and stock symbol.

We'll use two different techniques to create these tables

Employees

We'll create `employees` from a text file by using Spark SQL statements to create the schema just as we would in Hive. Unlike in the Hive example, we'll save some time by creating this table as an *external* table (i.e., one where we simply provide the location of the file that backs the table).

We'll build out the SQL string first and then invoke the `sql` method using our `spark` session variable.

```

val emptable = """CREATE EXTERNAL TABLE IF NOT EXISTS employees
  (name string,
   salary float,
   subordinates array<string>,
   deductions map<string, float>,
   address struct<street:string, city:string, state:string, zip:int>)
row format delimited
lines terminated by '\n'
stored as textfile location '/data/employees/input/'''

spark.sql(emptable)
spark.sql("select * from employees").show

```

You should see the results:

name	salary	subordinates	deductions	address
John Doe	100000.0	[Mary Smith, Todd...	Map(Federal Taxes...	[1 Michigan Ave.,...
Mary Smith	80000.0	[Bill King]	Map(Federal Taxes...	[100 Ontario St.,...
Todd Jones	70000.0	[]	Map(Federal Taxes...	[200 Chicago Ave....
Bill King	60000.0	[]	Map(Federal Taxes...	[300 Obscure Dr.,...
Boss Man	200000.0	[John Doe, Fred F...	Map(Federal Taxes...	[1 Pretentious Dr...
Fred Finance	150000.0	[Stacy Accountant]	Map(Federal Taxes...	[2 Pretentious Dr...
Stacy Accountant	60000.0	[]	Map(Federal Taxes...	[300 Main St.,Nap...

Stocks

To keep things simple, we're not going to expect that all our partitions for the stocks table have already been created. Instead, we're going to read in a flat input file and have Spark create a partitioned table in Hive from that file.

Here are the HDFS input files we are going to read:

```
/data/stocks-flat/input/NASDAQ_daily_prices_A.csv  
/data/stocks-flat/input/NASDAQ_daily_prices_I.csv  
/data/stocks-flat/input/NYSE_daily_prices_G.csv  
/data/stocks-flat/input/NYSE_daily_prices_I.csv
```

Read these in using Spark's native .csv reader, which was added to the distribution in Spark 2.0.0. We'll then add column names as arguments to the `toDF` function

```
val stocks = spark.read.format("csv").load("/data/stocks-flat/input/").  
  toDF("exchg",  
    "symbol",  
    "ymd",  
    "price_open",  
    "price_high",  
    "price_low",  
    "price_close",  
    "volume",  
    "price_adj_close")
```

Now we'll write this table into Hive. In the process, we'll specify that it should be partitioned by `exchg` and `symbol`. We'll then ask Spark to describe the table for us.

```
spark.sql("drop table stocks") // delete if already exists  
stocks.write.partitionBy("exchg", "symbol").saveAsTable("stocks")  
spark.sql("describe stocks").show(50)
```

You should see the following description showing that the table has been properly partitioned.

col_name	data_type	comment
ymd	string	null
price_open	string	null
price_high	string	null
price_low	string	null
price_close	string	null
volume	string	null
priceadjclose	string	null
exchg	string	null
symbol	string	null
# Partition Information		
# col_name	data_type	comment
exchg	string	null
symbol	string	null

SQL Queries

With the tables all set up, we can now do normal SQL queries on our tables. So let's get to it.

First, let's find those employees who live in Zip Code 60500.

```
spark.sql("SELECT name FROM employees WHERE address.zip = 60500").show()
```

You should see

name
Boss Man
Fred Finance

That was a piece of cake. Let's now transition to `stocks`, which is a bit more of a Big Data dataset at more than 2 million rows. Actually, let's count how many rows there actually are.

```
val stks = spark.read.table("stocks")
stks.count
```

The value should be 2,131,092.

One of the nice things about the SQL interface is that types get converted on the fly based on context. If we look at the description of the stock table above, every column was a string type. We're now going to do some numeric comparisons.

Up until this point, we've invoked `sql` as a method on our Spark Session. However, SQL is used so commonly that you can leave off the spark session reference.

```
sql("""SELECT ymd, price_open, price_close FROM stocks
WHERE symbol = 'AAPL' AND exchg = 'NASDAQ' LIMIT 20""").show()
```

Output should be

ymd	price_open	price_close
2015-06-22	127.489998	127.610001
2015-06-19	127.709999	126.599998
2015-06-18	127.230003	127.879997
2015-06-17	127.720001	127.300003
2015-06-16	127.029999	127.599998
2015-06-15	126.099998	126.919998
2015-06-12	128.190002	127.169998
2015-06-11	129.179993	128.589996
2015-06-10	127.919998	128.880005
2015-06-09	126.699997	127.419998
2015-06-08	128.899994	127.800003
2015-06-05	129.5	128.649994
2015-06-04	129.580002	129.360001
2015-06-03	130.660004	130.119995
2015-06-02	129.860001	129.960007
2015-06-01	130.279999	130.539993
2015-05-29	131.229996	130.279999
2015-05-28	131.860001	131.779999
2015-05-27	130.339996	132.039993
2015-05-26	132.600006	129.619995

Now let's see

```
sql("SELECT year(s.ymd), avg(s.price_close) FROM stocks s WHERE s.symbol = 'AAPL' AND s.exchg
```

You should now see

year(CAST(ymd AS DATE))	avg(CAST(price_close AS DOUBLE))
1990	37.56175417786561
2003	18.54476169444443
2007	128.2739047848606
2015	124.3063555169492
2013	472.63488080952385
1997	17.966775490118593
1988	41.53902472332016
1994	34.08054893650793
2014	295.4023412182538
2004	35.52694387301588
1982	19.142774332015808
1996	24.919478582677176
1989	41.65872438095236
1998	30.564851150793665
1985	20.195169592885374
2012	576.0497200880001
2009	146.81412911904766
1995	40.54017670238094
1980	30.442332307692308
2001	20.219431697580646

Big Data Science With Spark

Module 7: Spark Machine Learning Through Examples Labs and Exercises

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017



SparkML

Lab 01: Creates Personal Movie Ratings

This lab's only goal is to create a file of your personal ratings on a scale of 1 to 5 for the following movies:

- Toy Story (1995)
- Independence Day (a.k.a. ID4) (1996)
- Dances with Wolves (1990)
- Star Wars: Episode VI - Return of the Jedi (1983)
- Mission: Impossible (1996)
- Ace Ventura: Pet Detective (1994)
- Die Hard: With a Vengeance (1995)
- Batman Forever (1995)
- Pretty Woman (1990)
- Men in Black (1997)
- Dumb & Dumber (1994)

Most people today will find these movies quite old. They are chosen because they map to movies in the ratings database we'll be using to build our recommendation engine.

To rate your movies, you should perform the following steps on your cluster:

```
cd /mnt/sparkclass/exercises/SparkML-In-Depth
./rateMovies.py
```

The script will create a data file called `personalRatings.txt`.

Once you've rated those movies, you have completed this lab.



SparkML

Lab 02: Ingest and Understand the ML-100K Data

Objective

In this lab, we're going to familiarize ourselves with the ML-100K data set. Our objective is to ingest and join the three tables together so that we can see how users rated a wide variety of movies. We'll be using these datasets in our next lab where we try to recommend movies to a new user based on their movie preferences.

Background

As background, MovieLens 100k (ml-100k) consists of 100,000 ratings of movies. It includes three types of files about:

1. the movies themselves (`u.item`)
2. the users who did the ratings (`u.user`)
3. the ratings created by the users (`u.data`)

All the data is in text format separated by vertical bars or tabs, depending on the file.

All of these files are preloaded into HDFS at ``hdfs:///data/ml-100k`. Here are the contents of that HDFS directory


```
/data/ml-100k/README
/data/ml-100k/allbut.pl
/data/ml-100k/mku.sh
/data/ml-100k/u.data
/data/ml-100k/u.genre
/data/ml-100k/u.info
/data/ml-100k/u.item
/data/ml-100k/u.occupation
/data/ml-100k/u.user
/data/ml-100k/u1.base
/data/ml-100k/u1.test
/data/ml-100k/u2.base
/data/ml-100k/u2.test
/data/ml-100k/u3.base
/data/ml-100k/u3.test
/data/ml-100k/u4.base
/data/ml-100k/u4.test
/data/ml-100k/u5.base
/data/ml-100k/u5.test
/data/ml-100k/ua.base
/data/ml-100k/ua.test
/data/ml-100k/ub.base
/data/ml-100k/ub.test
```

Lab01: Characterizing The ML-100K Users

Our first job is reading in these files and trying to understand them. Let's start with the data about the actual users stored in the file `u.user`. The user data has fields separated by vertical bars. Here's what a few lines of `u.data` look like:

index	age	gender	occupation	zip
1	24	M	technician	85711
2	53	F	other	94043
3	23	M	writer	32067

Let's parse this and see how many users there are in total.

```
val user_data = sc.textFile("hdfs:///data/ml-100k/u.user")
user_data.count
```

You should have 943 users.

Now let's characterize all the fields, looking for distinct values of each.

```
val user_fields = user_data.map(line => line.split("\\|"))
user_fields.take(1)
val num_users = user_fields.map(fields => fields(0)).count
val num_genders = user_fields.map(fields => fields(2)).distinct.count
val num_occupations = user_fields.map(fields => fields(3)).distinct.count
val num_zipcodes = user_fields.map(fields => fields(4)).distinct.count
println("Users: %d, genders: %d, occupations: %d, ZIP codes: %d".format(num_users, num_genders,
```

We should be seeing output that looks like

```
Users: 943, genders: 2, occupations: 21, ZIP codes: 795
```

Challenge 1

Try to figure out how many users there are by occupation. Think about it for a moment and try to write RDD code for this. Occupation is field 3 of the data set. Hint: this is going to be a little like wordcount.

Answer 1

To do a count by occupation, we'll want to create key-value pairs of the form `(occupation, 1)` and then reduce those. So the code to do this will look like this:

```
val count_by_occupation = user_fields.map(fields => (fields(3), 1)).
reduceByKey(_+_).collect()
```

or, if you don't want to bother creating tuples, you can use `countByValue` instead:

```
val count_by_occupation2 = user_fields.
  map(fields => fields(3)).
  countByValue
```

Using A Case Class for User Occupations

We know from the module on Spark SQL that we can use case classes to assign schemas to files like `u.users`. Let's do that now. We'll create a case class called `Occ` that represents each row of `u.users`.

```
case class Occ(index: Int,  
  age: Int,  
  gender: String,  
  occupation: String,  
  zip: String)
```

With that in place, we can now parse the file into a dataframe of Occ objects.

```
val user_data = sc.textFile("hdfs:///data/ml-100k/u.user") // read from HDFS by default  
user_data.first  
val user_fields = user_data.map(line => line.split("\\|"))  
user_fields.take(5)  
val users_array = user_fields.  
  map(p => Occ(p(0).toInt, p(1).toInt, p(2), p(3), p(4)))  
val users_df = users_array.toDF()  
users_df.show  
val occupation_count = users_df.groupBy("occupation").count.sort(desc("count"))  
occupation_count.show
```

That gives us output that looks like this:

occupation	count
student	196
other	105
educator	95
administrator	79
engineer	67
programmer	66
librarian	51
writer	45
executive	32
scientist	31
artist	28
technician	27
marketing	26
entertainment	18
healthcare	16
retired	14
lawyer	12
salesman	12
none	9
homemaker	7

Ingesting Movie Names

At this point you have enough information to ingest the movie descriptions contained in the file `u.item` and the ratings contained in `u.data`. You'll need to know that the field separators for the movie descriptions are

vertical bars, while the ratings fields are separated by tabs.

Here's what the first three rows of the movie dataset, `u.item`, look like:

1	Toy Story (1995)	01-Jan-1995	http://us.imdb.com/M/title-exact?Toy%20Story%20(1995)	0	0	0	1
2	GoldenEye (1995)	01-Jan-1995	http://us.imdb.com/M/title-exact?GoldenEye%20(1995)	0	1	1	0
3	Four Rooms (1995)	01-Jan-1995	http://us.imdb.com/M/title-exact?Four%20Rooms%20(1995)	0	0	0	0

At present, we really only care about the item number (the first field) and the movie name (the second field). We're going to ignore the rest of the fields.

We can ingest this `u.item` file in a couple of ways. One way is to replicate what we did with the users dataset. We read the text file, scan for vertical bars, put the fields in the appropriate case class and then convert to a data frame. That code looks like this:

```
// input format movieid|Title|Genres
case class Movie(movieid: Int, title: String)

// function to parse movie record into Movie class
def parseMovie(str: String): Movie = {
    val fields = str.split("\\|")
    Movie(fields(0).toInt, fields(1))
}

// Let's join the movie file in so we can get titles

val movies_df = sc.textFile("/data/ml-100k/u.item").map(parseMovie).toDF()
```

Alternatively, we can use our Spark SQL skills and ask the `read` function to read this file as a .csv file with vertical bar as our separator and with a schema that specifies only the first two fields. That code would look like this:

```
// Alternative way of reading in movies_df

import org.apache.spark.sql.types._

val schema = new StructType().
  add($"movieid".long.copy(nullable = false)).
  add($"title".string)

val movies_df = spark.read.
  schema(schema).
  option("sep", "|").
  csv("/data/ml-100k/u.item")
```

Both options produce the same results, which look like this

movieid	title
1	Toy Story (1995)
2	GoldenEye (1995)
3	Four Rooms (1995)
4	Get Shorty (1995)
5	Copycat (1995)

And so on for about 1682 rows.

Ingesting the Ratings

The ratings file located in `/data/ml-100k/u.data` looks like this:

196	242	3	881250949
186	302	3	891717742
22	377	1	878887116
244	51	2	880606923
166	346	1	886397596
298	474	4	884182806
115	265	2	881171488
253	465	5	891628467
305	451	3	886324817

Clearly we need to decode this structure.

First, the fields are separated by tabs. The field types in order are:

```
userid: Int
itemid: Int
rating: Int
timestamp: String
```

Once again, we could ingest this data frame either with a case class or just by calling `spark.read` with the proper arguments. Again, we'll show it both ways.

First, using a case class

```
case class RatingObj(userid: Int, itemid: Int, rating: Int, timestamp: String)
val ratings_fields = ratings_data.map(line => line.split("\\t"))
val ratings_array = ratings_fields.map(p => RatingObj(p(0).toInt, p(1).toInt, p(2).toInt, p(3)))
val ratings_df = ratings_array.toDF()
```

Next, using a schema and `spark.read.csv`.

```
val ratingsschema = new StructType().
  add($"userid".long.copy(nullable = false)).
  add($"itemid".long.copy(nullable = false)).
  add($"rating".long).
  add($"timestamp".string)

val ratings_df2 = spark.read.
  schema(ratingsschema).
  option("sep", "\t").
  csv("/data/ml-100k/u.data")
```

Now we should have our ratings in a nice DataFrame

userid	itemid	rating	timestamp
196	242	3	881250949
186	302	3	891717742
22	377	1	878887116
244	51	2	880606923
166	346	1	886397596
298	474	4	884182806

Let's do a quick sanity check and compute the average ratings for each itemid. We'll even sort the results by average rating.

```
val ratings_means = ratings_df.groupBy("itemid").
  agg(avg(ratings_df("rating")).alias("avg_rating"))

ratings_means.orderBy(desc("avg_rating")).show
```

Joining Ratings With Movies

We now have the three major tables ingested. First, we'd like to join together the movies and the ratings so that we can see which movie titles were the highest rated.

We'll use a join operation to do this, and the join code is pretty straightforward. We'll want to join the `itemid`

field in the `ratings_df` table with the `movieid` field in the `movies_df` table. Note that we use the `===` operator to compare the values of the fields instead of the equality of the objects involved.

We also will cache the result for future queries.

```
val ratings_with_titleDF = ratings_df.  
  join(movies_df, ratings_df("itemid") === movies_df("movieid")).cache
```

The Rating For Best Movie Is....

We can now compute the average rating by movie title and sort the results in descending order. Let's do this using the Spark API first:

```
ratings_with_titleDF.  
  groupBy($"title").  
  agg(avg($"rating").alias("avg_rating")).  
  orderBy(desc("n")).show(10)
```

We see this as the result:

title	avg_rating
Great Day in Harl...	5.0
Prefontaine (1997)	5.0
Star Kid (1997)	5.0
Saint of Fort Was...	5.0
Aiqing wansui (1994)	5.0
Marlene Dietrich:...	5.0
Entertaining Ange...	5.0
Santa with Muscle...	5.0
They Made Me a Cr...	5.0
Someone Else's Am...	5.0
Pather Panchali (...)	4.625

Maya Lin: A Stron...	4.5
Some Mother's Son...	4.5
Anna (1996)	4.5
Everest (1998)	4.5
Close Shave, A (1...	4.491071428571429
Schindler's List ...	4.466442953020135
Wrong Trousers, T...	4.466101694915254
Casablanca (1942)	4.45679012345679
Wallace & Gromit:...	4.447761194029851
Shawshank Redempt...	4.445229681978798
Rear Window (1954)	4.3875598086124405
Usual Suspects, T...	4.385767790262173
Star Wars (1977)	4.3584905660377355
12 Angry Men (1957)	4.344

Conclusion: the users rating these movies have terrible taste: Star Kid rates higher than Casablanca? Really?

Maybe some of the movies weren't rated many times, allowing them to have surprisingly high ratings. We can check that by adding the count of the number of ratings to the resulting table. Let's update our query.

```
ratings_with_titleDF.  
  groupBy($"title").  
  agg(avg($"rating").alias("avg_rating"),  
       count($"rating").alias("n")).  
  orderBy(desc("avg_rating")).show(25)
```

And we now see:

title	avg_rating	n
They Made Me a Cr...	5.0	1

Someone Else's Am...	5.0	1
Entertaining Ange...	5.0	1
Saint of Fort Was...	5.0	2
Star Kid (1997)	5.0	3
Great Day in Harl...	5.0	1
Prefontaine (1997)	5.0	3
Marlene Dietrich:...	5.0	1
Aiqing wansui (1994)	5.0	1
Santa with Muscle...	5.0	2
Pather Panchali (...)	4.625	8
Maya Lin: A Stron...	4.5	4
Everest (1998)	4.5	2
Anna (1996)	4.5	2
Some Mother's Son...	4.5	2
Close Shave, A (1...	4.491071428571429	112
Schindler's List ...	4.466442953020135	298
Wrong Trousers, T...	4.466101694915254	118
Casablanca (1942)	4.45679012345679	243
Wallace & Gromit:...	4.447761194029851	67
Shawshank Redempt...	4.445229681978798	283
Rear Window (1954)	4.3875598086124405	209
Usual Suspects, T...	4.385767790262173	267
Star Wars (1977)	4.3584905660377355	583
12 Angry Men (1957)	4.344	125

OK I feel better now. Most of those obscure movies only had a few ratings.

By the way, We could also get this same result with a `sql` statement as follows:

```
sql("""select first(title),
      avg(rating) as avg_rating,
      count(itemid) from ratings
      group by itemid, movieid, title
      order by avg_rating DESC""").show
```

So for our final analysis, let's rank the top 10 movies by average rating, but only for those movies with 200 or more reviews. We can do that by simply adding a filter operation to the end of the chain.

```
ratings_with_titleDF.
  groupBy($"title").
  agg(avg($"rating").alias("avg_rating"),
      count($"rating").alias("n")).
  orderBy(desc("avg_rating")).where("n >= 200").show(10)
```

Now we see

title	avg_rating	n
Schindler's List ...	4.47	298
Casablanca (1942)	4.46	243
Shawshank Redempt...	4.45	283
Usual Suspects, T...	4.39	267
Rear Window (1954)	4.39	209
Star Wars (1977)	4.36	583
Silence of the La...	4.29	390
One Flew Over the...	4.29	264
To Kill a Mocking...	4.29	219
Godfather, The (1...	4.28	413

And with that result, our faith in our fellow moviegoers is restored.



SparkML

Lab 03: Building a Recommendation Engine

In this lab, we'll build out a simple recommendation engine based on our 100,000 movie ratings.

Preparation

Here's we're simply going to read in three files and register them as tables. See Lab 02 for definitions of these files and how they work. While we did this in Lab 02, we want to ensure that we have their contents as DataFrames

```
import org.apache.spark.sql.functions._
// Import Spark SQL data types
import org.apache.spark.sql._
// Import mllib recommendation data types
import org.apache.spark.mllib.recommendation.{ALS, MatrixFactorizationModel, Rating}
import org.apache.spark.util.random

// input format MovieID|Title|Genres
case class Movie(movieid: Int, title: String)

// input format is UserID|Gender|Age|Occupation|Zip-code
case class User(userid: Int, age: Int, gender: String, occupation: String, zip: String)

// function to parse movie record into Movie class
def parseMovie(str: String): Movie = {
  val fields = str.split("\\|")
  Movie(fields(0).toInt, fields(1))
}
// function to parse input into User class
```

```

def parseUser(str: String): User = {
    val fields = str.split("\\|")
    assert(fields.size == 5)
    User(fields(0).toInt, fields(1).toInt, fields(2).toString,
        fields(3).toString, fields(4).toString)
}

// function to parse input UserID\tMovieID\tRating
// Into org.apache.spark.mllib.recommendation.Rating class
def parseRating(str: String): Rating = {
    val fields = str.split("\\t")
    Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble)
}

//////////
// Ingest the data //
//////////

// OK, now read in the user and movie info files
val users = sc.textFile("hdfs:///data/ml-100k/u.user").map(parseUser)
val usersDF = sc.textFile("hdfs:///data/ml-100k/u.user").map(parseUser).toDF()
val moviesDF = sc.textFile("hdfs:///data/ml-100k/u.item").map(parseMovie).toDF()
val fileratings = sc.textFile("hdfs:///data/ml-100k/u.data").map(parseRating)
val myratings = sc.textFile("file:/mnt/sparkclass/exercises/SparkML-In-Depth/personalRatings.txt")
val ratings = fileratings
val ratingsDF = fileratings.toDF()

// register the DataFrames as a temp table for SQL queries
ratingsDF.registerTempTable("ratings")
moviesDF.registerTempTable("movies")
usersDF.registerTempTable("users")

```

Examining Our Data Along With A Few Surprises

Let's just check a bit of our data to ensure it's as we expect.

```

val usersDF = spark.read.table("users")
val moviesDF = spark.read.table("movies")
val ratingsDF = spark.read.table("ratings")

// Look at the top 10 rows of usersDF
usersDF.show(5)

```

This should look something like:

movieid	title
1	Toy Story (1995)
2	GoldenEye (1995)
3	Four Rooms (1995)
4	Get Shorty (1995)
5	Copycat (1995)

However, you may instead see a different set of movies, such as this:

movieid	title
839	Loch Ness (1995)
840	Last Man Standing...
841	Glimmer Man, The ...
842	Pollyanna (1960)
843	Shaggy Dog, The (...)

We actually can't predict what output you'll get. All we know is that you'll get the 5 of the movies in the dataset.

Why Can't We Predict What Movies You'll See?

The inability to predict how your table will be ordered is one of the effects of Spark being designed for maximum parallelism. Your Spark program runs on multiple processors and cores in parallel. Unlike in MapReduce, those cores don't synchronize their efforts unless we explicitly ask them to do so.

In this case, when we asked Spark to write out our table to the Hive Warehouse, it allowed each Spark executor to write its own file to the output directory without synchronization. You can in fact see how it wrote out these tables by looking in the Hive warehouse from the shell (permission and ownership details have been omitted below for print clarity):


```
hdfs dfs -ls /user/hive/warehouse/movies
/user/hive/warehouse/movies/_SUCCESS
/user/hive/warehouse/movies/part-00000-8e9dadd4-2abb-4329-a436-624df661978a-c000.snappy.parquet
/user/hive/warehouse/movies/part-00001-8e9dadd4-2abb-4329-a436-624df661978a-c000.snappy.parquet
```

Each of these files contains half of the table in compressed binary form. Which half you see first in your `show` results is determined only by which Spark executor wrote its results first. And since the Spark executors run in parallel, the order ends up being a matter of chance.

Examining Other Tables

While acknowledging this nondeterminism, examine the other tables you've read in using show commands such as the following:

```
usersDF.show(10)
moviesDF.show(10)
ratingsDF.show(10)
```

Analyzing The Ratings

Before we build out our recommendation engine, let's just look at a few specific movies using `filter`. If you're more used to SQL, you may prefer to use the second version.

```
moviesDF.filter(moviesDF("title").contains("Star Wars")).show
sql("""SELECT movieid, title FROM movies
      WHERE title LIKE \"Star Wars%\"""").show
```

You'll find the 1977 Star Wars has movieid 50.

movieid	title
50	Star Wars (1977)

Join the `moviesDF` DataFrame together with `ratingsDF`. You'll want the `movieid` column in `moviesDF` to match the `product` column in `ratingsDF`. Call that result `ratings_with_titleDF`, and then show the results of filtering that by the title *Star Wars*.

```
val ratings_with_titleDF = ratingsDF.join(moviesDF, ratingsDF("product") === moviesDF("movieid"))
ratings_with_titleDF.show(5)
// See ratings for Star Wars now
val starwars = ratings_with_titleDF
  .filter($"title".contains("Star Wars"))
  .show(5)
```

You'll see something like this, although the actual rows you see may be different due to parallelism.

user	product	rating	movieid	title
290	50	5.0	50	Star Wars (1977)
79	50	4.0	50	Star Wars (1977)
2	50	5.0	50	Star Wars (1977)
8	50	5.0	50	Star Wars (1977)
274	50	5.0	50	Star Wars (1977)

Now do the following steps:

1. join in the `usersDF` DataFrame, linking the `user` field in `ratings_with_titleDF` with the `userid` field in the `userDF` DataFrame. We'll call that `denorm_ratingsDF`.
2. Filter `denorm_ratingsDF` for `title` containing *Star Wars* and assign that to val `starwars`.
3. Show and persist `starwars`.
4. Group `denorm_ratingsDF` by the field `gender` and show the average rating by gender.
5. Check those results against the average of the total corpus of movie ratings grouped by `gender`.

Here's the code to do this.

```

val denorm_ratingsDF = ratings_with_titleDF.
  join(usersDF, ratings_with_titleDF("user") === usersDF("userid"))
val starwars = denorm_ratingsDF.
  filter(denorm_ratingsDF("title").
    contains("Star Wars"))
starwars.show(5)
starwars.persist
starwars.groupBy("gender").agg(avg($"rating")).show

// check the results against the total set of movies for sanity
denorm_ratingsDF.groupBy("gender").agg(avg($"rating")).show // compare with total set

```

The Star Wars ratings are:

gender	avg(rating)
F	4.245033112582782
M	4.398148148148148

While the overall ratings are

gender	avg(rating)
F	3.5315073815073816
M	3.5292889846485322

Draw your own conclusions from this data.

Building A Recommendations Engine

Like in many supervised machine learning algorithms, we'll use a multi-step process to build our recommendation engine.

1. Randomly split our data into training and test data sets.
2. Build a Alternating Least Squares (ALS) model based on the training data set that creates a known result (in this case, a rating). See the course slides for a discussion of what ALS does and why we use it.
3. Apply that model to user data to predict a new result (i.e., make a recommendation)
4. Measure how closely the predicted results match the actual rating results in the test set.

A key point in this exercise is that while we've focused on DataFrames up until this point, the Spark Recommendation Engine tool requires `Ratings` objects, which are defined in the package `org.apache.spark.mllib.recommendation`. Conveniently, our preparation for this lab has created a `ratings` RDD made up of `Ratings` objects. As an aside, this is a common pattern with Spark machine learning libraries: usually you have to create the proper type of objects, not just DataFrames, to use them.

Step 1: Randomly Split Our Data Into Training and Test Sets

Here, we'll use the `randomSplit` method for `RDD` objects to create our training and test sets. We'll use 80% of the data for training and leave 20% for testing. We'll also cache both of those RDDs for later use.

```
val splits = ratings.randomSplit(Array(0.8, 0.2), 0L)
val trainingRDD = splits(0).cache()
val testRatingsRDD = splits(1).cache()
```

Now add our ratings to the training set

```
val trainingRatingsRDD = trainingRDD.union(myratings).cache()

val numTraining = trainingRatingsRDD.count()
val numTest = testRatingsRDD.count()
println(s"Training: $numTraining, test: $numTest.")
```

You should see Training: 80018, test: 19993 or so, depending on your random number seed (we set ours to 0L).

Step 2: Build a Alternating Least Squares (ALS) Model Using The Training Set

All the hard work of parallelization and computation gets done in the ALS library. All we have to do is to set the parameters.

```
// Set up the parameters for training

val rank = 10           // number of hidden features in approximation matrices
val iterations = 10     // iterations of model to run
val lambda = 0.01       // tunable parameter controlling regularization and fitting

val model = ALS.train(trainingRatingsRDD, rank, iterations, lambda)
```

The value model returns is of type `MatrixFactorization`, which consists of two objects: `userFeatures` and `productFeatures`. These correspond to the two matrices whose dot product creates the full ratings matrix.

Step 3: Apply Our Model To User Data To Predict A Result (i.e., make a recommendation)

Before we actually do the prediction, which is just a method we invoke on our `model` object, let's look at a specific user. We'll arbitrarily use user 0, and first let's find out which movies he or she rated by extracting them out of `trainingRatingsRDD`.

```
// We'll test it by using it on our ratings (user 0)
val user0ratings = trainingRatingsRDD.filter(rating => rating.user == 0)
val user0ratingsDF = user0ratings.toDF()

/// these are the user 0 ratings used to train the model
user0ratingsDF.join(moviesDF,
  user0ratingsDF("product") === moviesDF("movieid")).show
```

Now let's look at the top 3 predictions for user 0.

```
val topRecForUser0 = model.recommendProducts(0, 3)
topRecForUser0.map(rating => (movieTitles(rating.product), rating.rating)).foreach(println)
```

We get back something that looks like this:

```
(Boxing Helena (1993),18.402586059612382)
(Harlem (1993),17.438677456080185)
(Joy Luck Club, The (1993),16.45317941043561)
```

Step 4. Measure How Close The Predicted Results Are To The Actual Rating Results In The Test Set

Now let's try to evaluate this model. How well does it perform? To do this, we'll have it do predictions for the entire test data set. Once we have a set of predictions, we'll then compare those with the actual ratings using a Mean Absolute Error (MAE) function.

We have to strip off the rating to get our predictions in our test set, so we'll use a case class to get a user-

product pair from testRatingsRDD.

```
val testUserProductRDD = testRatingsRDD.map {  
  case Rating(user, product, rating) => (user, product)  
}
```

Now we can just ask for a prediction for every user-product pair in the test data set.

```
// remember that testRatingsRDD is our test sample of all our data  
val predictionsForTestRDD = model.predict(testUserProductRDD)  
predictionsForTestRDD.take(5).mkString("\n")
```

You'll get back something like:

```
Rating(368,320,1.9506349488976316)  
Rating(3,320,3.256979002151988)  
Rating(21,320,4.715583624693077)  
Rating(752,752,3.453001416009629)  
Rating(883,752,3.3110704894373333)
```

Now how do we compare these results with the actual ratings given by the users? We'll create a key which is the user-product pair as a tuple, and then have its value be the rating. Then we can join these two RDDs together and compare the results.

```
// prepare predictions for comparison  
val predictionsKeyedByUserProductRDD = predictionsForTestRDD.map{  
  case Rating(user, product, rating) => ((user, product), rating)  
}  
// prepare test for comparison  
val testKeyedByUserProductRDD = testRatingsRDD.map{  
  case Rating(user, product, rating) => ((user, product), rating)  
}  
//Join the test with predictions  
val testAndPredictionsJoinedRDD = testKeyedByUserProductRDD.  
  join(predictionsKeyedByUserProductRDD)
```

Let's look at the comparison for a few products:

```
// print the (user, product),( test rating, predicted rating)
scala> testAndPredictionsJoinedRDD.take(5).mkString("\n")

((660,47),(2.0,2.695739817626425))
((918,495),(3.0,2.731500894684755))
((151,605),(4.0,3.693202454620277))
((249,235),(4.0,3.420363946441323))
((325,616),(4.0,3.4547303609194855))
```

We're not terribly far off. However, it would be nice to know how often our predictor really gets the result wrong.

Let's look for false positives in our predicted ratings. We'll define a false positive as a movie that the user rated a 1 or lower and the predictor rated the movie a 4 or more. We'll then count those to see how badly we're doing.

```
val falsePositives =(testAndPredictionsJoinedRDD.filter{
  case ((user, product), (ratingT, ratingP)) => (ratingT <= 1 && ratingP >=4)
})
falsePositives.take(3)
falsePositives.count
```

Finally, we'll evaluate the model by computing the Mean Absolute Error (MAE) and Mean Squared Error (MSE) between test and predictions. We'll just write our own version of that right here, although most people would use the library function to do this. The ML libraries have a variety of build-in functions to rate the quality of results.

```
//Evaluate the model using Mean Absolute Error (MAE) between test and predictions
val meanAbsoluteError = testAndPredictionsJoinedRDD.map {
  case ((user, product), (testRating, predRating)) =>
    val err = (testRating - predRating)
    Math.abs(err)
}.mean()
println("Mean Absolute Error = " + meanAbsoluteError)

//We can also calculate the Mean Squared Error (MSE) equally easily

val MSE = testAndPredictionsJoinedRDD.map{
  case ((user, product), (actual, predicted)) =>
    math.pow((actual - predicted), 2)
}.reduce(_ + _) / testAndPredictionsJoinedRDD.count
println("Mean Squared Error = " + MSE)
```

Using the fairly small number of iterations we specified above, we got an MAE of around 0.7 and a MSE of around 1. We have to expect every run to be different because the ALS algorithm initializes with a random value.

However, we can ask the question, can we improve this? What if we did more iterations? Added more hidden features? Tried a different lambda? We encourage you to experiment with this algorithm and see if you can reduce the MSE and MAE from our first tries.

This step concludes the lab.

Big Data Science With Spark

Module 8: Spark Streaming Labs and Exercises

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017



Spark Streaming

Lab 01: Reading File Streams

In this lab, we'll demonstrate streaming using files in an HDFS directory. Our streaming job will wait for a file to arrive in the directory and then immediately count the words in it.

Note that a new RDD is created and immediately processed each time a file is discovered.

We'll walk through the process in this lab. For this and all the streaming labs, you will require two terminal windows on your cluster. We'll qualify which instructions to type in which terminal window in each step.

Terminal 1: Making The Input Directory And Starting spark-shell

First, we have to ensure we run spark with enough cores to populate one core for the Spark streaming receiver and another for a Spark DStream.

Connect to your cluster using the secure shell, and type the following to start the spark-shell with two cores:

```
hdfs dfs -mkdir /tmp/streaming-input  
spark-shell --master local[2]
```

Terminal 1: Imports and Creating your Spark Context

We have to import the Spark Streaming module and some definitions. We also have to decide how often we want our micro-batches to run. For this example, we'll poll the input directory every 10 seconds

```
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}

// Create the streaming context
val ssc = new StreamingContext(sc, Seconds(10))
```

Note that the streaming context is created from our Spark Context, which in the spark-shell is in the variable `sc`.

Terminal 1: Wordcount code

Now we simply use the Scala wordcount code we are familiar with. The only difference with previous wordcount examples is that we're reading our text file with `sc.textFileStream` instead of `sc.textFile`.

```
val lines = ssc.textFileStream("/tmp/streaming-input")
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
wordCounts.print()
```

Terminal 1: Start The Stream

Now we simply start our stream using `ssc.start`.

```
ssc.start()
ssc.awaitTermination() // Wait for the computation to terminate
```

Terminal 2: Copy A File Into /tmp/streaming-input

We'll now copy all of Shakespeare into the input directory, followed by the text of some logs about 30 seconds later. ~~~ `bash` `hdfs dfs -cp /data/shakespeare/input/all-shakespeare.txt /tmp/streaming-input` `hdfs dfs -cp /data/logs/log.txt /tmp/streaming-input` ~~~

Terminal 1: See The WordCount Output

You should see the words get counted on Terminal 1 soon after you add each file. It should look like the following:

```
-----  
Time: 1504475990000 ms  
-----
```

```
(hack'd.,1)  
(House,1)  
(nobleman,10)  
(Never,,1)  
(dream'd,15)  
(stuck,,1)  
(perpetual.,2)  
(bombast,2)  
(unluckily,,2)  
(consideration,,3)  
...
```

```
-----  
Time: 1504476000000 ms  
-----
```

```
-----  
Time: 1504476010000 ms  
-----
```

```
(cluster:,1)  
(ERROR mysql,1)  
(angry,1)  
(mysql,1)  
(at,1)  
(ERROR php:,1)  
(are,1)  
(replace,1)  
(with,1)  
(me?,1)  
...
```

You should now kill the job by hitting control-C in Terminal 1.

This step concludes the lab.



SparkSQL in Scala

Lab01: Basic SparkSQL Walkthrough

At this point, we assume that you are running a `spark-shell` on your cluster. You should type the commands after the `scala>` prompt and you should see the outputs shown.

Begin by reading in a dataset. Hint: It's not exactly big data. In fact, it just looks like this:

```
[hadoop@ip-172-31-57-176 sparkclass]$ hdfs dfs -cat /data/spark-resources-data/people.json
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

That said, let's read it in so we have some data to apply SparkSQL to.

```
scala> val df = spark.read.json("/data/spark-resources-data/people.json")
df: org.apache.spark.sql.DataFrame = [age: bigint, name: string]
```

Now that we have people read in as a DataFrame, we can now query it in different ways.

```
// Displays the content of the DataFrame to stdout
df.show()
+----+-----+
| age|   name|
+----+-----+
| null|Michael|
|  30|   Andy|
|  19|  Justin|
+----+-----+
```

```
// Print the schema in a tree format
df.printSchema()
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
+-----+
|   name|
+-----+
|Michael|
|   Andy|
|  Justin|
+-----+

// Select everybody, but increment the age by 1
df.select(df.select($"name", df($"age" + 1)).show()
+-----+-----+
|   name|(age + 1)|
+-----+-----+
|Michael|    null|
|   Andy|    31|
|  Justin|    20|
+-----+-----+
f("name"), df("age" + 1).show()

// Select people older than 21
df.filter(df("age") > 21).show()
+---+-----+
|age|name|
+---+-----+
| 30|Andy|
+---+-----+

// Count people by age
df.groupBy("age").count().show()
+---+-----+
| age|count|
+---+-----+
|  19|    1|
| null|    1|
|  30|    1|
+---+-----+
```

Read A Table From The Hive Warehouse

If you ran the Hive-Tables2 and Hive-LoadingData queries in the Hive exercises, you should have a Hive table called stocks which has a lot of data loaded into 4 partitions

Remember that you if you read that into a database with your name on it (DB), you have to change to that DB using the `use yourname;` command. The following will simply assume you are using the default Hive database.

```
// Existing Spark session is assumed to be in spark
scala> spark.sql("USE default")           // Select my database
res0: org.apache.spark.sql.DataFrame = []

scala> val stocks = spark.sql("SELECT * FROM STOCKS")
stocks: org.apache.spark.sql.DataFrame = [ymd: string, price_open: float ... 7 more fields]

scala> stocks.show(5) // Will only show the first 5 rows
```

ymd	price_open	price_high	price_low	price_close	volume	price_adj_close	exchg	symbol
2015-06-22	127.49	128.06	127.08	127.61	33833500	127.61	NASDAQ	AAPL
2015-06-19	127.71	127.82	126.4	126.6	54181300	126.6	NASDAQ	AAPL
2015-06-18	127.23	128.31	127.22	127.88	35241100	127.88	NASDAQ	AAPL
2015-06-17	127.72	127.88	126.74	127.3	32768500	127.3	NASDAQ	AAPL
2015-06-16	127.03	127.85	126.37	127.6	31404000	127.6	NASDAQ	AAPL

```
only showing top 5 rows

scala> stocks.count()
res4: Long = 40547
```

Reading In A Large Dataset

We'd like to read in an FAA dataset that has all the US airline flights between 2010 and 2016. You could do that in SQL with the following command, but I don't recommend it; it has a lot of columns.

```
// This is one way to create a s3flights dataframe.
// val creates3flightsSQL = "CREATE EXTERNAL TABLE s3flights (
  Year INT,
  Quarter INT,
  Month INT,
```

DayOfMonth INT,
DayOfWeek INT,
FlightDate STRING,
UniqueCarrier STRING,
AirlineID INT,
Carrier STRING,
TailNum STRING,
FlightNum INT,
OriginAirportID INT,
OriginAirportSeqID INT,
OriginCityMarketID INT,
Origin STRING,
OriginCityName STRING,
OriginState STRING,
OriginStateFips INT,
OriginStateName STRING,
OriginWac INT,
DestAirportID INT,
DestAirportSeqID INT,
DestCityMarketID INT,
Dest STRING,
DestCityName STRING,
DestState STRING,
DestStateFips INT,
DestStateName STRING,
DestWac INT,
CRSDepTime INT,
DepTime INT,
DepDelay INT,
DepDelayMinutes INT,
DepDel15 INT,
DepartureDelayGroups INT,
DepTimeBlk STRING,
TaxiOut INT,
WheelsOff INT,
WheelsOn INT,
TaxiIn INT,
CRSArrTime INT,
ArrTime INT,
ArrDelay INT,
ArrDelayMinutes INT,
ArrDel15 INT,
ArrivalDelayGroups INT,
ArrTimeBlk STRING,
Cancelled TINYINT,

CancellationCode STRING,
Diverted TINYINT,
CRSElapsedTime INT,
ActualElapsedTime INT,
AirTime INT,
Flights INT,
Distance INT,
 DistanceGroup INT,
CarrierDelay INT,
WeatherDelay INT,
NASDelay INT,
SecurityDelay INT,
LateAircraftDelay INT,
FirstDepTime INT,
TotalAddGTime INT,
LongestAddGTime INT,
DivAirportLandings INT,
DivReachedDest INT,
DivActualElapsedTime INT,
DivArrDelay INT,
DivDistance INT,
Div1Airport STRING,
Div1AirportID INT,
Div1AirportSeqID INT,
Div1WheelsOn INT,
Div1TotalGTime INT,
Div1LongestGTime INT,
Div1WheelsOff INT,
Div1TailNum STRING,
Div2Airport STRING,
Div2AirportID INT,
Div2AirportSeqID INT,
Div2WheelsOn INT,
Div2TotalGTime INT,
Div2LongestGTime INT,
Div2WheelsOff INT,
Div2TailNum STRING,
Div3Airport STRING,
Div3AirportID INT,
Div3AirportSeqID INT,
Div3WheelsOn INT,
Div3TotalGTime INT,
Div3LongestGTime INT,
Div3WheelsOff INT,
Div3TailNum STRING,

```

Div4Airport STRING,
Div4AirportID INT,
Div4AirportSeqID INT,
Div4WheelsOn INT,
Div4TotalGTime INT,
Div4LongestGTime INT,
Div4WheelsOff INT,
Div4TailNum STRING,
Div5Airport STRING,
Div5AirportID INT,
Div5AirportSeqID INT,
Div5WheelsOn INT,
Div5TotalGTime INT,
Div5LongestGTime INT,
Div5WheelsOff INT,
Div5TailNum STRING
)
STORED AS PARQUET LOCATION 's3://think.big.academy.aws/ontime/parquet'
//val s3flights = hiveContext.sql(creates3flightsSQL)

```

The reason I don't recommend this approach is that SQL requires us to specify all the columns and their types. That information is already in the parquet file, so it seems redundant.

We can take advantage of SparkSQL's intelligence by simply loading the parquet file as a dataframe and letting it infer the table schema.

We happen to already have this file in a slightly trimmed down form in HDFS. It's at

```
hdfs:///data/flightdata/parquet-trimmed
```

```

scala> val s3flights = spark.read.parquet("hdfs:///data/flightdata/parquet-trimmed")
s3flights: org.apache.spark.sql.DataFrame = [year: int, quarter: int ... 62 more fields]

```

Now let's select only some of the fields. Further, we're going to filter it to only be the year 2013. Note that the use of a data pipeline allows Spark to lazily evaluate s3flights and only pull those records that are of interest into memory.

Note that because this is a dataframe, our filter tests MUST use === for equality, not == The good news: this runs pretty fast, much faster than the SQL equivalent.

```
scala> val flights = s3flights.select("year", "month", "dayofmonth", "carrier", "tailnum",
    "actualelapsedtime", "origin", "dest", "deptime", "arrdelayminutes").
    filter(s3flights("year") === 2013)
flights: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [year: int, month: int ... 8

scala> flights.show(5)
17/08/29 19:59:40 WARN Utils: Truncated the string representation of a plan since it was too l
+---+---+---+---+---+---+---+---+---+---+---+---+
|year|month|dayofmonth|carrier|tailnum|actualelapsedtime|origin|dest|deptime|arrdelayminutes|
+---+---+---+---+---+---+---+---+---+---+---+---+
|2013|  1|    18|DL|N325US|    184|PHL|MSP|    758|         0|
|2013|  1|    18|DL|N325NB|    172|FLL|LGA|    657|         0|
|2013|  1|    18|DL|N649DL|    190|LGA|ATL|   1657|        24|
|2013|  1|    18|DL|N130DL|    251|SLC|ATL|    953|        43|
|2013|  1|    18|DL|N651DL|    171|BOS|ATL|    711|         0|
+---+---+---+---+---+---+---+---+---+---+---+---+
only showing top 5 rows

scala> flights.printSchema
root
 |-- year: integer (nullable = true)
 |-- month: integer (nullable = true)
 |-- dayofmonth: integer (nullable = true)
 |-- carrier: string (nullable = true)
 |-- tailnum: string (nullable = true)
 |-- actualelapsedtime: integer (nullable = true)
 |-- origin: string (nullable = true)
 |-- dest: string (nullable = true)
 |-- deptime: integer (nullable = true)
 |-- arrdelayminutes: integer (nullable = true)
```

Now let's do some simple operations on this DataFrame. How many flights did each carrier fly in 2013? That's a one-liner:

```
scala> flights.groupBy("carrier").count.show
```

```
+-----+-----+
|carrier|  count|
+-----+-----+
|      UA| 505798|
|      AA| 537891|
|      EV| 748696|
|      B6| 241777|
|      DL| 754670|
|      OO| 626359|
|      F9|  75612|
|      YV| 140922|
|      US| 412373|
|      MQ| 439865|
|      HA|  72286|
|      AS| 154743|
|      FL| 173952|
|      VX|  57133|
|      WN|1130704|
|      9E| 296701|
+-----+-----+
```

Let's now compute average delay by carrier and destination

```
scala> flights.groupBy("carrier", "dest").mean("arrdelayminutes").show
```

```
+-----+-----+
|carrier|dest|avg(arrdelayminutes)|
+-----+-----+
|      DL|STL|      8.348766061594942|
|      DL|MSY|      8.593764258555133|
|      AS|IAH|      8.534246575342467|
|      EV|JAX|     19.50561403508772|
|      EV|LFT|     12.419161676646707|
|      EV|SYR|     18.71390798519302|
|      B6|SRQ|     12.782278481012659|
|      US|ROC|      9.662983425414364|
|      UA|JFK|     14.447821229050279|
|      VX|MCO|      4.666026871401152|
|      VX|LAS|     11.598548621190131|
|      WN|ALB|     14.44125144843569|
|      WN|BWI|     11.236715445573436|
|      OO|TUL|     12.621647058823529|
|      AS|SLC|      4.045955882352941|
|      DL|OAK|      3.1056768558951964|
|      MQ|HSV|     17.594059405940595|
|      US|ORD|     15.945858981533297|
|      WN|MAF|     11.323456790123457|
|      OO|EAU|     13.131884057971014|
+-----+-----+
```

only showing top 20 rows

Now by carrier, destination and origin!

```
scala> val delays = flights.groupBy("carrier", "dest", "origin").mean("arrdelayminutes")
delays: org.apache.spark.sql.DataFrame = [carrier: string, dest: string ... 2 more fields]
```

```
scala> delays.printSchema
```

root

```
-- carrier: string (nullable = true)
-- dest: string (nullable = true)
-- origin: string (nullable = true)
-- avg(arrdelayminutes): double (nullable = true)
```

Note that we've created a new column named avg(arrdelayminutes) Let's sort this to show the worst flights in terms of average delays in 2013

```
scala> delays.sort(desc("avg(arrdelayminutes)")).show(5)
```

```
+-----+-----+-----+-----+
|carrier|dest|origin|avg(arrdelayminutes)|
+-----+-----+-----+-----+
|      EV|EYW|  MIA|          375.0|
|      EV|GPT|  MSY|          315.0|
|      UA|DEN|  MSN|          285.0|
|      EV|RIC|  PIT|          145.0|
|      EV|AEX|  LFT|          138.0|
+-----+-----+-----+-----+
```

only showing top 5 rows

The worst delays were between Key West and Miami, FL Who knew?

Persisting DataFrames

Now let's look at some timing. Let's time the following operation using the `spark.time` method:

```
spark> spark.time(flights.groupBy("carrier", "dest", "origin").
  mean("arrdelayminutes").
  sort(desc("avg(arrdelayminutes)")).show())
```

```
+-----+-----+-----+
|carrier|dest|origin|avg(arrdelayminutes)|
+-----+-----+-----+
|      EV|EYW|  MIA|          375.0|
|      EV|GPT|  MSY|          315.0|
|      UA|DEN|  MSN|          285.0|
|      EV|RIC|  PIT|          145.0|
|      EV|AEX|  LFT|          138.0|
|      UA|SFO|  MSN|          132.0|
|      YV|ITO|  OGG|          132.0|
|      EV|ATL|  JFK|          128.0|
|      UA|MSN|  ORD|          123.5|
|      UA|STL|  EWR|          121.0|
|      EV|XNA|  LGA|          119.0|
|      OO|IAH|  PBI|          117.0|
|      EV|BTV|  BNA|          114.0|
|      F9|COS|  DEN|          111.0|
|      EV|MEM|  BOS|          110.0|
|      EV|GRK|  SAT|          109.0|
|      OO|PBI|  IAH|          107.0|
|      OO|ORD|  LGA|          107.0|
|      UA|CLE|  IAD|          104.0|
|      UA|LAX|  SEA|          103.0|
+-----+-----+-----+
```

only showing top 20 rows

Time taken: 18743 ms

Now persist the flights dataset by typing `flights.persist()` . Then See how long that command take now.

```
spark> spark.time(flights.groupBy("carrier", "dest", "origin").
  mean("arrdelayminutes").
  sort(desc("avg(arrdelayminutes)")).show())
```

```
+-----+-----+-----+-----+
|carrier|dest|origin|avg(arrdelayminutes)|
+-----+-----+-----+-----+
|      EV|EYW|  MIA|          375.0|
|      EV|GPT|  MSY|          315.0|
|      UA|DEN|  MSN|          285.0|
|      EV|RIC|  PIT|          145.0|
|      EV|AEX|  LFT|          138.0|
|      UA|SFO|  MSN|          132.0|
|      YV|ITO|  OGG|          132.0|
|      EV|ATL|  JFK|          128.0|
|      UA|MSN|  ORD|          123.5|
|      UA|STL|  EWR|          121.0|
|      EV|XNA|  LGA|          119.0|
|      OO|IAH|  PBI|          117.0|
|      EV|BTV|  BNA|          114.0|
|      F9|COS|  DEN|          111.0|
|      EV|MEM|  BOS|          110.0|
|      EV|GRK|  SAT|          109.0|
|      OO|ORD|  LGA|          107.0|
|      OO|PBI|  IAH|          107.0|
|      UA|CLE|  IAD|          104.0|
|      UA|LAX|  SEA|          103.0|
+-----+-----+-----+-----+
```

only showing top 20 rows

Time taken: 16630 ms

Persistence doesn't seem to help much, does it?

Actually it helps more than you think. Remember that Scala uses lazy evaluation for its results. When we told Spark to persist the `flights` DataFrame, it evaluated that lazily; it has no idea whether that DataFrame has been computed yet or not -- it just remembered that it should cache it when it next computes it. Therefore, the second run of the command was the first one that cached `flights`.

If this is the case, we should see substantially better performance if we run the command one more time


```
spark> spark.time(flights.groupBy("carrier", "dest", "origin").
  mean("arrdelayminutes").
  sort(desc("avg(arrdelayminutes)")).show())
```

```
+-----+-----+-----+-----+
|carrier|dest|origin|avg(arrdelayminutes)|
+-----+-----+-----+-----+
|      EV|EYW|  MIA|          375.0|
|      EV|GPT|  MSY|          315.0|
|      UA|DEN|  MSN|          285.0|
|      EV|RIC|  PIT|          145.0|
|      EV|AEX|  LFT|          138.0|
|      UA|SFO|  MSN|          132.0|
|      YV|ITO|  OGG|          132.0|
|      EV|ATL|  JFK|          128.0|
|      UA|MSN|  ORD|          123.5|
|      UA|STL|  EWR|          121.0|
|      EV|XNA|  LGA|          119.0|
|      OO|IAH|  PBI|          117.0|
|      EV|BTV|  BNA|          114.0|
|      F9|COS|  DEN|          111.0|
|      EV|MEM|  BOS|          110.0|
|      EV|GRK|  SAT|          109.0|
|      OO|ORD|  LGA|          107.0|
|      OO|PBI|  IAH|          107.0|
|      UA|CLE|  IAD|          104.0|
|      UA|LAX|  SEA|          103.0|
+-----+-----+-----+-----+
```

only showing top 20 rows

Time taken: 3614 ms

Sure enough, that did the trick. The third time we ran that command, it ran in roughly a fifth of the time taken for the first run.

This step concludes this lab.



Spark Streaming

Lab 01: Reading Network Streams

In this lab, we'll demonstrate streaming using a network connection. Our streaming job will wait for someone to connect to a network socket and will process the data that arrives on that socket.

Note that a new RDD is created and immediately processed each time a file is discovered.

We'll walk through the process in this lab. For this and all the streaming labs, you will require two terminal windows on your cluster. We'll qualify which instructions to type in which terminal window in each step.

Terminal 1: Starting spark-shell

First, we have to ensure we run spark with enough cores to populate one core for the Spark streaming receiver and another for a Spark DStream.

Connect to your cluster using the secure shell, and type the following to start the spark-shell with two cores:

```
spark-shell --master local[2]
```

Terminal 1: Set Up For Network Streaming

The only difference between this setup and the one in Lab 01 is that we are going to use

`ssc.socketTextStream` instead of `ssc.textFileStream` to read our data. `socketTextStream` requires us to specify what socket we want to listen to, so we've chosen 999. We still will use a 10 second micro-batch interval.

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._ // not necessary since Spark 1.3

// Create a local StreamingContext with two working threads
// and a batch interval of 10 second.
// The master requires 2 cores to prevent from a starvation scenario.

val ssc = new StreamingContext(sc, Seconds(10))
val lines = ssc.socketTextStream("localhost", 9999)
```

Terminal 1: Wordcount code

Here, everything is pretty much the same as with our file-based streaming wordcount program.

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

// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

// Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.print()
```

Terminal 1: Start The Stream

Now we simply start our stream using `ssc.start`.

```
ssc.start()
ssc.awaitTermination() // Wait for the computation to terminate
```

Terminal 2: Send Some Data To Network Port 9999

We're going to use the network cat program `nc` to provide input to our stream. We do that by the following command

```
nc -lk 9999
```

The terminal will now wait for you to type words. You should type some input for the streaming program. I suggest something such as:

```
Do you like green eggs and ham?  
I do not like them, Sam-I-Am.  
I do not like green eggs and ham.  
Would you like them here or there?  
I would not like them here or there.  
I would not like them anywhere.  
I do not like green eggs and ham.  
I do not like them, Sam-I-Am.
```

Wait for about 30 seconds after you've typed that, and then copy and paste it again.

```
Do you like green eggs and ham?  
I do not like them, Sam-I-Am.  
I do not like green eggs and ham.  
Would you like them here or there?  
I would not like them here or there.  
I would not like them anywhere.  
I do not like green eggs and ham.  
I do not like them, Sam-I-Am.
```

Terminal 1: See The WordCount Output

You should see the words get counted on Terminal 1 soon after you add each file. It should look like the following:

```
-----  
Time: 1504477320000 ms  
-----
```

```
(Sam-I-Am.,2)  
(here,2)  
(them,,2)  
(not,6)  
(or,2)  
(green,3)  
(anywhere.,1)  
(would,2)  
(Would,1)  
(like,8)  
...
```

```
-----  
Time: 1504477330000 ms  
-----
```

```
17/09/03 22:22:13 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.  
17/09/03 22:22:13 WARN BlockManager: Block input-0-1504477333600 replicated to only 0 peer(s) :  
-----
```

```
Time: 1504477340000 ms  
-----
```

```
(Sam-I-Am.,2)  
(here,2)  
(them,,2)  
(not,6)  
(or,2)  
(green,3)  
(anywhere.,1)  
(would,2)  
(Would,1)  
(like,8)  
...
```

You should now kill the job by hitting control-C in Terminal 1 and Terminal 2. You will likely see many errors on Terminal 1 as the streaming network job shuts down.

This step concludes the lab.



Spark Structured Streaming

Lab 03: Reading Network Streams

In this lab, we'll demonstrate streaming using a network connection using the Structured Streaming API. Our streaming job will wait for someone to connect to a network socket and will process the data that arrives on that socket.

Unlike in our prior labs, in this case, we'll be using Streaming Dataframes.

We'll walk through the process in this lab. For this and all the streaming labs, you will require two terminal windows on your cluster. We'll qualify which instructions to type in which terminal window in each step.

Terminal 1: Starting spark-shell

First, we have to ensure we run spark with enough cores to populate one core for the Spark streaming receiver and another for a Spark DStream.

Connect to your cluster using the secure shell, and type the following to start the spark-shell with two cores:

```
spark-shell --master local[2]
```

Terminal 1: Set Up For Network Streaming

Here, we're going to use the more modern Spark 2.0 instantiation of a network stream, using

```
spark.readStream.format
```

 with a set of options.

```
// Create DataFrame representing the stream of input lines from connection to localhost:9999
val lines = spark.readStream.
  format("socket").
  option("host", "localhost").
  option("port", 9999).load()
```

Terminal 1: Wordcount code

Because we get to read the data as a DataFrame, we can use DataFrame operations such as groupBy.

```
// Split the lines into words
val words = lines.as[String].flatMap(_.split(" "))

// Generate running word count
val wordCounts = words.groupBy("value").count()
```

Terminal 2: Set Up Network Port 9999

We're going to use the network cat program `nc` to provide input to our stream. We do that by the following command

```
nc -lk 9999
```

Terminal 1: Start Streaming

Now we will set up our query of the network along with some parameters about how to process the output. We are going to use "complete" mode, which allows us to count all the words we have seen since the stream started.

```
val query = wordCounts.writeStream.
  outputMode("complete").
  format("console")
query.start().awaitTermination()
```

Terminal 2: Type Words

The terminal will now wait for you to type words. You should type some input for the streaming program. I

suggest something such as:

```
Now is the time for the winter of our discontent.  
Tomorrow and tomorrow and tomorrow  
Creeps forth in its petty pace from day to day  
To the last syllable of recorded time.
```

Terminal 1: See The WordCount Output

You should see the words get counted on Terminal 1 soon after you add each file. It should look like the following:

```
-----  
Batch: 0  
-----  
+-----+-----+  
|      value|count|  
+-----+-----+  
|    winter|    1|  
|      for|    1|  
|       is|    1|  
|     Now|    1|  
|     our|    1|  
|discontent.|    1|  
|      the|    2|  
|       of|    1|  
|     time|    1|  
+-----+-----+  
  
-----  
Batch: 1  
-----  
+-----+-----+  
|      value|count|  
+-----+-----+  
|  Tomorrow|    1|  
| tomorrow|    2|  
|    winter|    1|  
|      for|    1|  
|       is|    1|  
|     Now|    1|  
|     our|    1|
```


discontent.	1
the	2
and	2
of	1
time	1

+-----+-----+

Batch: 2

value count
-----+-----
Tomorrow 1
time. 1
tomorrow 2
winter 1
forth 1
for 1
in 1
petty 1
is 1
syllable 1
its 1
Now 1
our 1
discontent. 1
the 3
creeps 1
and 2
of 2
time 1
recorded 1

+-----+-----+

only showing top 20 rows

You should now kill the job by hitting control-C in Terminal 1 and Terminal 2. You will likely see many errors on Terminal 1 as the streaming network job shuts down.

This step concludes the lab.

Big Data Science With Spark

Module 9: Spark In Python Labs and Exercises

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017

Spark Basics Walkthrough (01-spark-basics-lab.{scala,md})

This file gets you started in Spark in Scala.

Turn down the logging level

By default Spark versions before 2.0, many of Spark's logging parameters are set to `INFO` leading to extremely verbose messages printed to the console. In fact, if you don't turn these down, the prompt will soon scroll off the screen!

On Amazon EMR

If you are running a version of Spark before 2.0, edit `/usr/lib/spark/conf/log4j.properties` on the cluster with your text editor of choice (`emacs` , `vi` , `nano`) to selectively change `INFO` entries to `WARN` . To change them all easily:

```
sed -i.bak 's/INFO/WARN/' /usr/lib/spark/conf/log4j.properties
```

On Vagrant

If you are running a version of Spark before 2.0, edit `/vagrant/latest-spark/conf/log4j.properties` on the cluster with your text editor of choice (`emacs` , `vi` , `nano`) to selectively change `INFO` entries to `WARN` . To change them all easily:

```
sed -i.bak 's/INFO/WARN/' /vagrant/latest-spark/conf/log4j.properties
```

Launch Spark's interactive environment

Begin by starting the Pyspark shell from the Linux Shell command line

```
pyspark --master yarn-client          // start in client mode
```

At the `>>>` prompt, examine the Spark Context you were handed when you started Spark by typing `sc`. Throughout this lab, we'll show what you type immediately after the `>>>` prompt followed by what pyspark returns.

```
>>> sc
<SparkContext master=yarn appName=PySparkShell>
>>> sc.version
u'2.2.0'
>>> sc.appName
u'PySparkShell'
>>>
```

Play with an RDD

Now create a simple, small RDD by hand:

```
smallprimes = sc.parallelize(Array(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
```

Try some basic operations on the RDD. Note that a function takes no arguments, no parentheses are needed after the function name.

```
>>> smallprimes = sc.parallelize([2, 3, 5, 7, 11, 13, 17, 19, 23,
... 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97])
>>> smallprimes.count()
25
>>> smallprimes.min()
2
>>> smallprimes.max()
97
>>> smallprimes.sum()
1060
>>> smallprimes.collect()
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
>>>
```

Read data from HDFS

A more realistic scenario is to create an RDD from data read from disk. Spark can natively access HDFS and

S3 in addition to the local file system. Try reading in the stock quote data from HDFS:

```
>>> rdd = sc.textFile("hdfs:///data/stocks-flat/input")
>>> rdd.first()
u'NASDAQ,AAIT,2015-06-22,35.299999,35.299999,35.299999,35.299999,300,35.299999'
>>> rdd.count()
2131092
>>>
```

Simple Transformations and Actions

Let's first filter the data set so we only see AAPL records:

```
>>> aapl = rdd.filter(lambda line: "AAPL" in line)
>>> aapl.count()
8706
>>>
```

First Program In Pyspark (02-Pyspark-First-Program)

We saw in our walkthrough that we count the number of stock quotes for AAPL using code that looks like this:

```
rdd = sc.textFile("hdfs:///data/stocks-flat/input")
aapl = rdd.filter(lambda line: "AAPL" in line)
aapl.count()
```

Try using that pattern to count the number of lines in the file /data/shakespeare/input that contain the word "king". You'll want to use the toLowerCase function before the contains function to ensure all the text is lower case.

One solution looks like this:

```
rdd = sc.textFile("hdfs:///data/shakespeare/input")
kings = rdd.filter(lambda line: "king" in line.lower())
kings.count()
```

You should have gotten 4773 lines that reference a king in all of Shakespeare. That's almost 3% of the 175,376 lines in all of Shakespeare's plays and poems.



A TERADATA COMPANY

Spark in Scala

Spark Transformations Program

In this lab, you should type the following into your spark-shell:

```
fib = sc.parallelize([1, 2, 3, 5, 8, 13, 21, 34])
```

Then, using the REPL, use both named functions and anonymous functions to do the following:

- Compute all the squares
- Return those squares that are divisible by 3

You'll want to use the `.map` and `.filter` transformations on `fib` to invoke your functions

The answer is

```
/>>> ## Compute all the squares and sum all the values provided
... fib = sc.parallelize([1, 2, 3, 5, 8, 13, 21, 34])
>>>
>>> def square(x): return x * x
...
>>> def divisible(x): return (x % 3 == 0)
...
>>> ## First using named functions
... fib.map(square).filter(divisible).collect()
[9, 441]
>>>
>>> ## Now use functional literals
... fib.map(lambda x: x*x).filter(lambda y: y % 3 == 0).collect()
[9, 441]
>>>
```


Pyspark Functions, Anonymous and Otherwise

Simple operations in python

Try typing the following lines at the spark-shell:

```
>>> 2 + 2                                # Should be Int = 2
4
>>> 2.0 + 2                             # Should be Double = 2.0
4.0
>>> "This is a string"                  # 'This is a string'
'This is a string'
```

Function Definitions In Pyspark

In this little exercise, we'll define a simple function in the most straightforward, obvious way and then show how Scala's ability to infer context and types allows us to be more succinct.

Let's start by defining a simple add function and testing it.

```
def add(x, y):
    return x + y

add(42,13)

# Shorter version all on one line
def add(x, y): return x + y
```

You should see 55 as your result.

We don't have a lot of other fancy definition forms for Python as we did for Scala.

Anonymous Functions or Function Literals

First, we'll define a named greeting function named `greeting` that just prepends a cheery "Hello ".

```
# a named function greeting
def greeting(x): return "Hello " + x

greeting("Joe")
'Hello Joe'
>>>
```

Now we'll do the same with an anonymous function, which we'll assign to the variable `greeting`. We get the same cheery greeting.

```
# an anonymous function whose definition is assigned to variable greeting
greeting = lambda x: "Hello " + x
greeting("Joe")
'Hello Joe'
>>>
```

Now let's create a list of names and apply our anonymous function stored in `greeting` to that list. Unfortunately, we can't use `map` to do this because `map` isn't defined for local lists. Instead, we'll use a Python comprehension.

```
names = ["Joe", "Mary", "Barbara"]
# comprehension for a local list;
# local lists don't have the map method defined.
[greeting(x) for x in names]
['Hello Joe', 'Hello Mary', 'Hello Barbara']
>>>
```

Now do this in Pyspark

If we parallelize our list, we can then use the `map` function on our list.

```
names = sc.parallelize(["Joe", "Mary", "Barbara"])
names.map(greeting).collect()
['Hello Joe', 'Hello Mary', 'Hello Barbara']
>>>
```

```
# Now let's get rid of the name greeting
```

```
names.map(lambda x: "Hello " + x).collect()
['Hello Joe', 'Hello Mary', 'Hello Barbara']
>>>
```

Here are a collection of other function definitions to try out to get a feel for how anonymous functions can shorten our code and make it more readable at the same time.

```

maximize = lambda a, b: a if (a > b) else b
maximize(5, 3)

# Define doubler as an immutable variable whose value is a function (note we aren't using def)
doubler = lambda x: x * 2 # This assigns a function literal to doubler
doubler(4)

# We can also use this for our even number tester
nums = sc.parallelize([1, 2, 3, 4, 5])
nums.map(lambda x: x % 2 == 0).collect()

# These literal definitions are incredibly valuable in Spark
# because many of the Spark operations
# take functional arguments. Most of these will never be assigned
# to a variable. For example
# the following finds list entries that have an "f"
# in them using an anonymous function/function literal

mylist = sc.parallelize(["foo", "bar", "fight"])
mylist.filter(lambda x: "f" in x).collect()

# We could have written that as follows,
# but it's longer and less clear

mylist = sc.parallelize(["foo", "bar", "fight"])
def ffilter(s):
    return "f" in s # define a named function to search for f

mylist.filter(ffilter).collect() # will generate the same result as previous

```

A Full PySpark Example

Load error messages from a log into memory, then interactively search for various patterns.

The file `log.txt` has the following 5 lines:

```
ERROR      php: dying for unknown reasons
WARN       dave, are you angry at me?
ERROR      did mysql just barf?
WARN       xylons approaching
ERROR      mysql cluster: replace with spark cluster
```

Our objective is to count all the error messages (not warnings) that have reference *mysql* or *php*.

```
lines = sc.textFile("hdfs:///data/logs/log.txt")

# transformed RDDs
errors = lines.filter(lambda line: line.startswith("ERROR"))
fields = errors.map(lambda message: message.split("\t"))
messages = fields.map(lambda r: r[1])
messages.cache()

# actions
messages.filter(lambda m: "mysql" in m).count()
messages.filter(lambda m: "php" in m).count()
```

You should see 2 messages that have "mysql" in them, and one message that has "php".

Now Wordcount

The periods in this dataflow pipeline are at the ends of lines so that we can execute this code in interactive pyspark without modification.

Type this into pyspark and see how it works for you.

```
text_file = sc.textFile("hdfs:///data/shakespeare/input")
word_counts = text_file \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b)

# Let's sort the results by count

sorted_counts = word_counts.sortBy(lambda x: x[1], ascending=False)
sorted_counts.saveAsTextFile("hdfs:///tmp/shakespeare-wc-python")
```

You can examine the results using `hdfs dfs`. We've omitted the permissions information so that the lines are visible in our printed output.

```
hdfs dfs -ls /tmp/shakespeare-wc-python
Found 3 items
/tmp/shakespeare-wc-python/_SUCCESS
/tmp/shakespeare-wc-python/part-00000
/tmp/shakespeare-wc-python/part-00001
[vagrant@edge ~]$ hdfs dfs -cat /tmp/shakespeare-wc-python/part-00000 | head -5
hdfs dfs -cat /tmp/shakespeare-wc-python/part-00000 | head -5
(u'the', 25815)
(u'I', 20402)
(u'and', 19249)
(u'to', 17222)
(u'of', 16526)
```

Compute Pi

Estimate Pi in Pyspark

This program generates 100,000 x and y variables between 0 and 1. It then counts the ratio of those that fall within a unit circle over the number of total samples; that value should approximate pi/4.

Try various values of NUM_SAMPLES to see how the computed value and runtimes vary.

```
import sys
from random import random
from operator import add
NUM_SAMPLES = 10000
def f(_):
    x = random() * 2 - 1
    y = random() * 2 - 1
    return 1 if x ** 2 + y ** 2 <= 1 else 0

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

Pi is roughly 3.136000
>>>
```

It's not a great estimate, but you can improve it by increasing the number of samples.

This step concludes the lab.



A TERADATA COMPANY

SparkSQL in Python

Lab05: Basic Pyspark SparkSQL Walkthrough

At this point, we assume that you are running a `spark-shell` on your cluster. You should type the commands after the `python>` prompt and you should see the outputs shown.

Begin by reading in a dataset. Hint: It's not exactly big data. In fact, it just looks like this:

```
[hadoop@ip-172-31-57-176 sparkclass]$ hdfs dfs -cat /data/spark-resources-data/people.json
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

That said, let's read it in so we have some data to apply SparkSQL to.

```
df = spark.read.json("/data/spark-resources-data/people.json")
```

Now that we have people read in as a DataFrame, we can now query it in different ways.

```
// Displays the content of the DataFrame to stdout
df.show()
+----+-----+
| age | name |
+----+-----+
| null | Michael |
| 30 | Andy |
| 19 | Justin |
+----+-----+
```



```
// Print the schema in a tree format
df.printSchema()
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
+-----+
|   name|
+-----+
|Michael|
|   Andy|
|  Justin|
+-----+

// Select everybody, but increment the age by 1
df.select(df.name, df.age + 1).show()
+-----+-----+
|   name|(age + 1)|
+-----+-----+
|Michael|      null|
|   Andy|       31|
|  Justin|       20|
+-----+-----+

// Select people older than 21
df.filter(df("age") > 21).show()
+---+-----+
|age|name|
+---+-----+
| 30|Andy|
+---+-----+

// Count people by age
df.groupBy("age").count().show()
+---+-----+
|age|count|
+---+-----+
| 19|     1|
|null|     1|
| 30|     1|
+---+-----+
```

Read A Table From The Hive Warehouse

If you ran the Hive-Tables2 and Hive-LoadingData queries in the Hive exercises, you should have a Hive table called stocks which has a lot of data loaded into 4 partitions

Remember that you if you read that into a database with your name on it (DB), you have to change to that DB using the `use yourname;` command. The following will simply assume you are using the default Hive database.

```
>>> spark.sql("USE default")          ## Select my database
DataFrame[]
>>>
>>> stocks = spark.sql("SELECT * FROM STOCKS")
>>> stocks.show(5)  ## Will only show the first 20 rows
```

ymd	price_open	price_high	price_low	price_close	volume	price_adj_close	exchg	symbol
2015-06-22	167.649994	168.339996	167.199997	167.729996	2210700	167.729996	NYSE	IBM
2015-06-19	167.619995	168.419998	166.770004	166.990005	6971200	166.990005	NYSE	IBM
2015-06-18	167.050003	168.720001	167.050003	168.25	3329100	168.25	NYSE	IBM
2015-06-17	167	167.850006	166.100006	167.169998	2861100	167.169998	NYSE	IBM
2015-06-16	166.330002	167.399994	165.910004	166.839996	3246900	166.839996	NYSE	IBM

```
only showing top 5 rows

>>> stocks.count() ## How many rows? Should be 40,547
2131092
>>>
```

Reading In A Large Dataset

We'd like to read in an FAA dataset that has all the US airline flights between 2010 and 2016. You could do that in SQL with the following command, but I don't recommend it; it has a lot of columns.

```
// This is one way to create a s3flights dataframe.
// val creates3flightsSQL = "CREATE EXTERNAL TABLE s3flights (
  Year INT,
  Quarter INT,
  Month INT,
  DayOfMonth INT,
  DayOfWeek INT,
```

FlightDate STRING,
UniqueCarrier STRING,
AirlineID INT,
Carrier STRING,
TailNum STRING,
FlightNum INT,
OriginAirportID INT,
OriginAirportSeqID INT,
OriginCityMarketID INT,
Origin STRING,
OriginCityName STRING,
OriginState STRING,
OriginStateFips INT,
OriginStateName STRING,
OriginWac INT,
DestAirportID INT,
DestAirportSeqID INT,
DestCityMarketID INT,
Dest STRING,
DestCityName STRING,
DestState STRING,
DestStateFips INT,
DestStateName STRING,
DestWac INT,
CRSDepTime INT,
DepTime INT,
DepDelay INT,
DepDelayMinutes INT,
DepDel15 INT,
DepartureDelayGroups INT,
DepTimeBlk STRING,
TaxiOut INT,
WheelsOff INT,
WheelsOn INT,
TaxiIn INT,
CRSArrTime INT,
ArrTime INT,
ArrDelay INT,
ArrDelayMinutes INT,
ArrDel15 INT,
ArrivalDelayGroups INT,
ArrTimeBlk STRING,
Cancelled TINYINT,
CancellationCode STRING,
Diverted TINYINT,

CRSElapsedTime INT,
ActualElapsedTime INT,
AirTime INT,
Flights INT,
Distance INT,
 DistanceGroup INT,
CarrierDelay INT,
WeatherDelay INT,
NASDelay INT,
SecurityDelay INT,
LateAircraftDelay INT,
FirstDepTime INT,
TotalAddGTime INT,
LongestAddGTime INT,
DivAirportLandings INT,
DivReachedDest INT,
DivActualElapsedTime INT,
DivArrDelay INT,
DivDistance INT,
Div1Airport STRING,
Div1AirportID INT,
Div1AirportSeqID INT,
Div1WheelsOn INT,
Div1TotalGTime INT,
Div1LongestGTime INT,
Div1WheelsOff INT,
Div1TailNum STRING,
Div2Airport STRING,
Div2AirportID INT,
Div2AirportSeqID INT,
Div2WheelsOn INT,
Div2TotalGTime INT,
Div2LongestGTime INT,
Div2WheelsOff INT,
Div2TailNum STRING,
Div3Airport STRING,
Div3AirportID INT,
Div3AirportSeqID INT,
Div3WheelsOn INT,
Div3TotalGTime INT,
Div3LongestGTime INT,
Div3WheelsOff INT,
Div3TailNum STRING,
Div4Airport STRING,
Div4AirportID INT,

```

    Div4AirportSeqID INT,
    Div4WheelsOn INT,
    Div4TotalGTime INT,
    Div4LongestGTime INT,
    Div4WheelsOff INT,
    Div4TailNum STRING,
    Div5Airport STRING,
    Div5AirportID INT,
    Div5AirportSeqID INT,
    Div5WheelsOn INT,
    Div5TotalGTime INT,
    Div5LongestGTime INT,
    Div5WheelsOff INT,
    Div5TailNum STRING
)
STORED AS PARQUET LOCATION 's3://think.big.academy.aws/ontime/parquet'
//val s3flights = hiveContext.sql(creates3flightsSQL)

```

The reason I don't recommend this approach is that SQL requires us to specify all the columns and their types. That information is already in the parquet file, so it seems redundant.

We can take advantage of SparkSQL's intelligence by simply loading the parquet file as a dataframe and letting it infer the table schema.

We happen to already have this file in a slightly trimmed down form in HDFS. It's at

```
hdfs:///data/flightdata/parquet-trimmed .
```

```
s3flights = spark.read.parquet("hdfs:///data/flightdata/parquet-trimmed")
```

Now let's select only some of the fields. Further, we're going to filter it to only be the year 2013. Note that the use of a data pipeline allows Spark to lazily evaluate s3flights and only pull those records that are of interest into memory.

The good news: this runs pretty fast, much faster than the Hive equivalent.

```
p>>> flights = s3flights.select("year", "month", "dayofmonth", \
... "carrier", "tailnum", "actualelapsedtime", \
... "origin", "dest", "deptime", "arrdelayminutes"). \
... filter(s3flights.year == 2013)
>>> flights.show(5)
```

year	month	dayofmonth	carrier	tailnum	actualelapsedtime	origin	dest	deptime	arrdelayminutes
2013	1	18	DL	N325US	184	PHL	MSP	758	0
2013	1	18	DL	N325NB	172	FLL	LGA	657	0
2013	1	18	DL	N649DL	190	LGA	ATL	1657	24
2013	1	18	DL	N130DL	251	SLC	ATL	953	43
2013	1	18	DL	N651DL	171	BOS	ATL	711	0

only showing top 5 rows

```
>>> flights.printSchema()
root
 |-- year: integer (nullable = true)
 |-- month: integer (nullable = true)
 |-- dayofmonth: integer (nullable = true)
 |-- carrier: string (nullable = true)
 |-- tailnum: string (nullable = true)
 |-- actualelapsedtime: integer (nullable = true)
 |-- origin: string (nullable = true)
 |-- dest: string (nullable = true)
 |-- deptime: integer (nullable = true)
 |-- arrdelayminutes: integer (nullable = true)
```

Now let's do some simple operations on this DataFrame. How many flights did each carrier fly in 2013? That's a one-liner:

```
>>> flights.groupBy("carrier").count().show()
```

```
+-----+-----+
|carrier|  count|
+-----+-----+
|      UA| 505798|
|      AA| 537891|
|      EV| 748696|
|      B6| 241777|
|      DL| 754670|
|      OO| 626359|
|      F9|  75612|
|      YV| 140922|
|      US| 412373|
|      MQ| 439865|
|      HA|  72286|
|      AS| 154743|
|      FL| 173952|
|      VX|  57133|
|      WN|1130704|
|      9E| 296701|
+-----+-----+
```

Let's now compute average delay by carrier and destination

```
>>> flights.groupBy("carrier", "dest").mean("arrdelayminutes").show()
```

```
+-----+-----+-----+
|carrier|dest|avg(arrdelayminutes)|
+-----+-----+-----+
|      DL|STL|      8.348766061594942|
|      DL|MSY|      8.593764258555133|
|      AS|IAH|      8.534246575342467|
|      EV|JAX|     19.50561403508772|
|      EV|LFT|     12.419161676646707|
|      EV|SYR|     18.71390798519302|
|      B6|SRQ|     12.782278481012659|
|      US|ROC|      9.662983425414364|
|      UA|JFK|     14.447821229050279|
|      VX|MCO|      4.666026871401152|
|      VX|LAS|     11.598548621190131|
|      WN|ALB|     14.44125144843569|
|      WN|BWI|     11.236715445573436|
|      OO|TUL|     12.621647058823529|
|      AS|SLC|      4.045955882352941|
|      DL|OAK|      3.1056768558951964|
|      MQ|HSV|     17.594059405940595|
|      US|ORD|     15.945858981533297|
|      WN|MAF|     11.323456790123457|
|      OO|EAU|     13.131884057971014|
+-----+-----+-----+
```

only showing top 20 rows

Now by carrier, destination and origin!

```
>>> delays = flights.groupBy("carrier", "dest", "origin").mean("arrdelayminutes")
```

```
>>> delays.printSchema()
```

root

```
|-- carrier: string (nullable = true)
|-- dest: string (nullable = true)
|-- origin: string (nullable = true)
|-- avg(arrdelayminutes): double (nullable = true)
```

Note that we've created a new column named avg(arrdelayminutes) Let's sort this to show the worst flights in terms of average delays in 2013


```
>>> delays.sort(desc("avg(arrdelayminutes)").show(5)
```

```
+-----+-----+-----+
|carrier|dest|origin|avg(arrdelayminutes)|
+-----+-----+-----+
|      EV|EYW|  MIA|          375.0|
|      EV|GPT|  MSY|          315.0|
|      UA|DEN|  MSN|          285.0|
|      EV|RIC|  PIT|          145.0|
|      EV|AEX|  LFT|          138.0|
+-----+-----+-----+
```

only showing top 5 rows

The worst delays were between Key West and Miami, FL Who knew?

This step concludes this lab.



SparkSQL in Python

Lab06: SparkSQL Schemas

This lab will demonstrate how Spark SQL can use Hive tables. If you have already created the `employees` and `stocks` Hive tables as part of the Spark Hive Fundamentals module, you may skip directly to the section labeled **Querying Hive Tables from Spark SQL**. However, we recommend you read through this next section anyway because interacting with Hive is a powerful feature of Spark. Further, Spark's intelligence simplifies some aspects of table manipulation over Hive's SQL-based approach.

Creating Hive Tables

The Hive exercises create two tables for later use:

1. `employees` is a very simple table of employees that uses complex data types such as maps and structs
2. `stocks` is a Hive table of roughly 2 million stock prices partitioned by stock exchange and stock symbol.

We'll use two different techniques to create these tables

Employees

We'll create `employees` from a text file by using Spark SQL statements to create the schema just as we would in Hive. Unlike in the Hive example, we'll save some time by creating this table as an *external* table (i.e., one where we simply provide the location of the file that backs the table).

We'll build out the SQL string first and then invoke the `sql` method using our `spark` session variable.

```
emptable = """CREATE EXTERNAL TABLE IF NOT EXISTS employees
  (name string,
   salary float,
   subordinates array<string>,
   deductions map<string, float>,
   address struct<street:string, city:string, state:string, zip:int>)
row format delimited
lines terminated by '\n'
stored as textfile location '/data/employees/input/""""

spark.sql(emptable)
spark.sql("select * from employees").show()
```

You should see the results:

name	salary	subordinates	deductions	address
John Doe	100000.0	[Mary Smith, Todd...	Map(Federal Taxes...	[1 Michigan Ave.,...
Mary Smith	80000.0	[Bill King]	Map(Federal Taxes...	[100 Ontario St.,...
Todd Jones	70000.0	[]	Map(Federal Taxes...	[200 Chicago Ave....
Bill King	60000.0	[]	Map(Federal Taxes...	[300 Obscure Dr.,...
Boss Man	200000.0	[John Doe, Fred F...	Map(Federal Taxes...	[1 Pretentious Dr...
Fred Finance	150000.0	[Stacy Accountant]	Map(Federal Taxes...	[2 Pretentious Dr...
Stacy Accountant	60000.0	[]	Map(Federal Taxes...	[300 Main St.,Nap...

Stocks

To keep things simple, we're not going to expect that all our partitions for the stocks table have already been created. Instead, we're going to read in a flat input file and have Spark create a partitioned table in Hive from that file.

Here are the HDFS input files we are going to read:

```
/data/stocks-flat/input/NASDAQ_daily_prices_A.csv
/data/stocks-flat/input/NASDAQ_daily_prices_I.csv
/data/stocks-flat/input/NYSE_daily_prices_G.csv
/data/stocks-flat/input/NYSE_daily_prices_I.csv
```

Read these in using Spark's native .csv reader, which was added to the distribution in Spark 2.0.0. We'll then add column names as arguments to the `toDF` function

```
stocks = spark.read.format("csv"). \
load("/data/stocks-flat/input/"). \
toDF("exchg", "symbol", \
"ymd", "price_open", \
"price_high", \
"price_low", \
"price_close", \
"volume", \
"price_adj_close")
```

Now we'll write this table into Hive. In the process, we'll specify that it should be partitioned by `exchg` and `symbol`. We'll then ask Spark to describe the table for us.

```
spark.sql("drop table stocks") ## delete if already exists
stocks.write.partitionBy("exchg", "symbol").saveAsTable("stocks")
spark.sql("describe stocks").show(50)
```

You should see the following description showing that the table has been properly partitioned.

col_name	data_type	comment
ymd	string	null
price_open	string	null
price_high	string	null
price_low	string	null
price_close	string	null
volume	string	null
priceadjclose	string	null
exchg	string	null
symbol	string	null
# Partition Information		
# col_name	data_type	comment
exchg	string	null
symbol	string	null

SQL Queries

With the tables all set up, we can now do normal SQL queries on our tables. So let's get to it.

First, let's find those employees who live in Zip Code 60500.

```
spark.sql("SELECT name FROM employees WHERE address.zip = 60500").show()
```

You should see

name
Boss Man
Fred Finance

That was a piece of cake. Let's now transition to `stocks`, which is a bit more of a Big Data dataset at more than 2 million rows. Actually, let's count how many rows there actually are.

```
stks = spark.read.table("stocks")
stks.count()
```

The value should be 2,131,092.

One of the nice things about the SQL interface is that types get converted on the fly based on context. If we look at the description of the stock table above, every column was a string type. We're now going to do some numeric comparisons.

Up until this point, we've invoked `sql` as a method on our Spark Session. However, SQL is used so commonly that you can leave off the spark session reference.

```
sql("""SELECT ymd, price_open, price_close FROM stocks
WHERE symbol = 'AAPL' AND exchg = 'NASDAQ' LIMIT 20""").show()
```

Output should be

ymd	price_open	price_close
2015-06-22	127.489998	127.610001
2015-06-19	127.709999	126.599998
2015-06-18	127.230003	127.879997
2015-06-17	127.720001	127.300003
2015-06-16	127.029999	127.599998
2015-06-15	126.099998	126.919998
2015-06-12	128.190002	127.169998
2015-06-11	129.179993	128.589996
2015-06-10	127.919998	128.880005
2015-06-09	126.699997	127.419998
2015-06-08	128.899994	127.800003
2015-06-05	129.5	128.649994
2015-06-04	129.580002	129.360001
2015-06-03	130.660004	130.119995
2015-06-02	129.860001	129.960007
2015-06-01	130.279999	130.539993
2015-05-29	131.229996	130.279999
2015-05-28	131.860001	131.779999
2015-05-27	130.339996	132.039993
2015-05-26	132.600006	129.619995

Now let's see

```
sql("SELECT year(s.ymd), avg(s.price_close) FROM stocks s WHERE s.symbol = 'AAPL' AND s.exchg
```

You should now see

year(CAST(ymd AS DATE))	avg(CAST(price_close AS DOUBLE))
1990	37.56175417786561
2003	18.54476169444443
2007	128.2739047848606
2015	124.3063555169492
2013	472.63488080952385
1997	17.966775490118593
1988	41.53902472332016
1994	34.08054893650793
2014	295.4023412182538
2004	35.52694387301588
1982	19.142774332015808
1996	24.919478582677176
1989	41.65872438095236
1998	30.564851150793665
1985	20.195169592885374
2012	576.0497200880001
2009	146.81412911904766
1995	40.54017670238094
1980	30.442332307692308
2001	20.219431697580646

Big Data Science With Spark

Module 10: Spark In R Labs and Exercises

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017



Spark In R

Lab 01: Connecting To RStudio Server

The goal of this lab is to ensure you can connect and use the RStudio Integrated Development Environment.

RStudio usually runs on your server on a designated node. In our case, you'll find it on port 8787. To get started with R on RStudio Server, go to

IPADDRESS:8787

Where IPADDRESS is the address of the edge node on your cluster.

You should see a login screen that looks like this:



Sign in to RStudio

Username:

hadoop

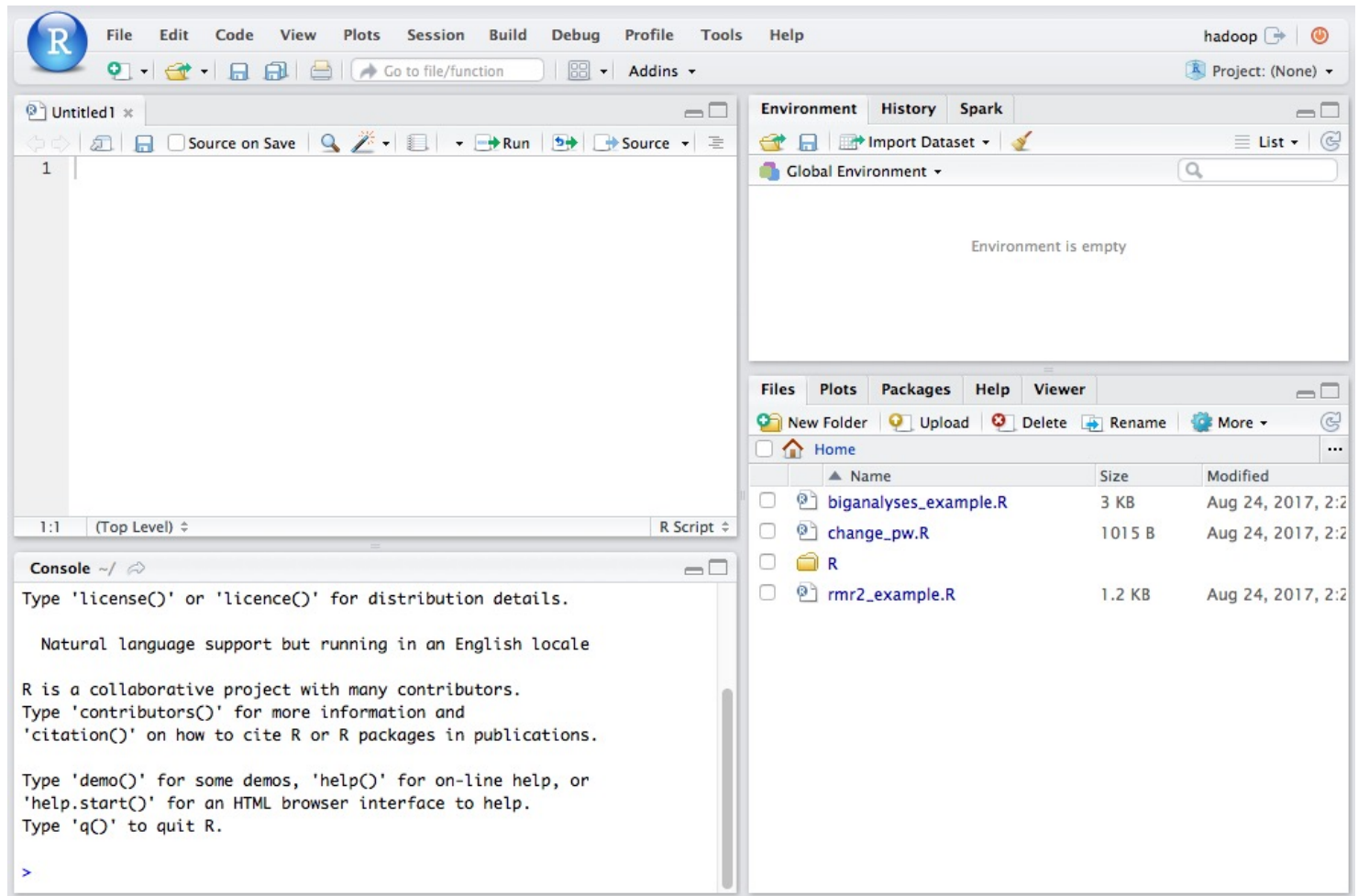
Password:

.....

☐ Stay signed in

Sign In

Log in with user name *hadoop*, password *hadoop*. You should then see the RStudio main screen



You'll want to open the script for the next lab in RStudio. Do the following to do this:

1. Select **Open File** from the **File** menu.
2. In the Open File window, click ... to the right of the **Home** bar.
3. Type the path `/mnt/sparkclass/exercises/Spark-In-R` into the box labelled *Path to folder*.
4. Select the file name `02-SparkR-API.R`.

You should now have the code for the next lab displayed in the top left-hand window.

This step concludes the lab.



Spark In R

Lab 02: Coding To The SparkR API

The goal of this lab is to introduce you to the SparkR API for R.

Setup

If you haven't done so already, connect to RStudio Server on your cluster. You can find the process for doing so in Lab 01.

You can find the code for this walkthrough in `02-SparkR-API.R`.

Initializing R For Spark

For SparkR to run properly under RStudio, it requires several environment variables to be properly set. Specifically, SparkR needs to know:

- `SPARK_HOME`: Where Spark is installed.
- `HADOOP_CONF_DIR`: Where the Hadoop configuration files lives.
- `YARN_CONF_DIR`: The configuration directory for YARN.

We'll set those up before we initialize Spark using the following code:

```

if (nchar(Sys.getenv("SPARK_HOME")) < 1) {
  Sys.setenv(SPARK_HOME = "/usr/lib/spark") # or wherever your Spark install lives
}
if (nchar(Sys.getenv("HADOOP_CONF_DIR")) < 1) {
  Sys.setenv(HADOOP_CONF_DIR = "/etc/hadoop/conf") # or wherever your Hadoop lives
}
if (nchar(Sys.getenv("YARN_CONF_DIR")) < 1) {
  Sys.setenv(YARN_CONF_DIR = "/etc/hadoop/conf") # or wherever your YARN config lives
}

```

Now, we'll load the `magrittr` package for data pipelining and then initialize Spark. We'll request 3GB of memory for the driver program in our initialization. This typically takes about 10-15 seconds to complete because it must spin up the Spark executors.

For those R programmers who typically load the `dplyr` library on startup, you don't want to do that when working with Spark because some of the `dplyr` function names conflict with the SparkR API functions, which can result in very confusing errors.

```

library(magrittr)
# please note -- do not load dplyr or you will have function name conflicts

library(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK_HOME"), "R", "lib")))
sparkR.session(master = "yarn", # execution type
  sparkConfig = list(spark.driver.memory = "3g")) # configure driver and executors

```

Getting Started With SparkR DataFrames

R natively supports a `data.frame` type, which is implemented as a list of columns. SparkR supports a different dataframe type called a `DataFrame`, which is implemented within the Spark Cluster as an RDD of Rows. You must use different functions for the two different types! You also must be careful to spell the dataframe you want appropriately.

Here's an example of the built-in data.frame `faithful` being converted to a `DataFrame`.

```

head(faithful)
##   eruptions waiting
## 1      3.600      79
## 2      1.800      54
## 3      3.333      74
## 4      2.283      62
## 5      4.533      85
## 6      2.883      55

str(faithful)
## 'data.frame':   272 obs. of  2 variables:
## $ eruptions: num  3.6 1.8 3.33 2.28 4.53 ...
## $ waiting : num  79 54 74 62 85 55 88 85 51 85 ...

df <- as.DataFrame(faithful) # note this is as.DataFrame, not as.data.frame
# Displays the first part of the SparkDataFrame
head(df)
## you see the same numbers but it takes longer.
##   eruptions waiting
##1      3.600      79
##2      1.800      54
##3      3.333      74

```

Throughout this lab, we'll largely focus on SparkR DataFrames, not data.frames.

Simple DataFrame Operations

We'll walk through the `people.json` data set just as we did in Scala and Python. However, the syntax for R is quite different because R allows the dot character in variable names. As a result, we will use the `magrittr` pipe operator `%>%` when we want to chain the result of one DataFrame operation into the input of another.

So let's read in `/data/spark-resources-data/people.json` as a DataFrame and perform simple DataFrame operations on it.

```

df <- read.df("/data/spark-resources-data/people.json", "json")
# Show the content of the DataFrame
showDF(df)
##   age   name
## 1  NA Michael
## 2  30    Andy

```

```
## 3 19 Justin

printSchema(df)
## root
## |-- age: long (nullable = true)
## |-- name: string (nullable = true)

names(df)
## [1] "age" "name"

df %>% select("name") %>% showDF()
## +-----+
## |  name|
## +-----+
## |Michael|
## |  Andy|
## | Justin|
## +-----+

## If you don't like pipelines, you can also express this as:
showDF(select(df, "name"))
## +-----+
## |  name|
## +-----+
## |Michael|
## |  Andy|
## | Justin|
## +-----+
##
## However, this gets messy for more complicated pipelines

df %>% select(df$name, df$age + 1) %>% showDF()
## +-----+-----+
## |  name|(age + 1.0)|
## +-----+-----+
## |Michael|      null|
## |  Andy|      31.0|
## | Justin|      20.0|
## +-----+-----+

df %>% filter(df$age > 21) %>% showDF()
## +---+-----+
## |age|name|
## +---+-----+
## | 30|Andy|
```

```
## +---+---+
df %>% groupBy("age") %>% count() %>% showDF()
## +---+---+
## | age|count|
## +---+---+
## | 19|    1|
## |null|    1|
## | 30|    1|
## +---+---+

## Let's save this as a table
createOrReplaceTempView(df, "people")
sql("show tables") %>% showDF()
## +-----+-----+
## |tableName|isTemporary|
## +-----+-----+
## |  people|        false|
## +-----+-----+

teenagers <- sql("select name from people where age >= 13 and age <= 19")
showDF(teenagers)
## +-----+
## |  name|
## +-----+
## |Justin|
## +-----+
```

Here's a quick rundown of what's different when we do DataFrame operations in R instead of Scala or Python:

- The pipeline operator is `%>%`, not `.``.
- We show DataFrames using the R `showDF` function instead of `show`.
- We select columns in R using the `$` operator.
- We don't explicitly reference the Spark session (i.e., we use `read.DF` or `loadDF`, not `spark.read` or `spark.load`). Similarly, we use `sql` to invoke a SQL statement, not `spark.sql` as in the other languages.

Working With The Stocks Table

As with Scala and Python, we'll want to be able to use Hive tables in our analyses.

You probably have already created a `stocks` table in prior labs. However, if you haven't, don't worry; we can

recreate it here from the original .csv file quite easily using `read.df`. While versions of Spark earlier than 2.0 didn't include a .csv file reader, it's now a built-in format in the `read.df` function (and in the `spark.read` functions in Scala and Python).

```
stocks <- read.df("/data/stocks-flat/input", "csv")
names(stocks) <- c("exchg", "symbol", "ymd", "price_open",
  "price_high", "price_low", "price_close",
  "volume", "price_adj_close")
createOrReplaceTempView(stocks, "stocks")
```

Please note that we didn't specify a schema, so Spark is going to read all those fields as strings.

Check to make sure you read the DataFrame correctly. We'll look at the first few rows and then count the total number of rows and those rows that reference AAPL.

```
first10 <- sql("select * from stocks limit 10")
showDF(first10)
```

exchg	symbol	ymd	price_open	price_high	price_low	price_close	volume	price_adj_close
NASDAQ	AAIT	2015-06-22	35.299999	35.299999	35.299999	35.299999	300	35.299999
NASDAQ	AAIT	2015-06-19	35.259998	35.259998	35.259998	35.259998	400	35.259998
NASDAQ	AAIT	2015-06-18	34.52	34.830002	34.52	34.830002	300	34.830002
NASDAQ	AAIT	2015-06-17	34.650002	34.650002	34.650002	34.650002	200	34.650002
NASDAQ	AAIT	2015-06-16	34.799999	34.799999	34.709999	34.77	700	34.77
NASDAQ	AAIT	2015-06-15	35.009998	35.009998	35.009998	35.009998	0	35.009998
NASDAQ	AAIT	2015-06-12	34.639999	35.009998	34.639999	35.009998	300	35.009998
NASDAQ	AAIT	2015-06-11	34.68	34.68	34.68	34.68	200	34.68
NASDAQ	AAIT	2015-06-10	34.689999	34.689999	34.689999	34.689999	0	34.689999
NASDAQ	AAIT	2015-06-09	34.189999	34.689999	34.189999	34.689999	3500	34.689999

```
count(stocks)
## Answer should be 2131092

stocks %>% filter(stocks$symbol == "AAPL") %>% count()
## Answer should be 8706
```

Finally, we'll stop our R Spark Session. While this isn't strictly required because our Spark session times out within a few minutes if you don't use it, explicitly stopping the Spark session prevents complaints from the R interpreter.

```
sparkR.session.stop()
```

This step concludes the lab.



Spark In R

Lab 03: Coding Using Sparklyr

The goal of this lab is to introduce you to the Sparklyr package.

Sparklyr was developed by the team that built the RStudio IDE and is distributed by RStudio.com. It provides a Spark backend to the `dplyr` package but uses the traditional dplyr primitives for coding. Filters and aggregates Spark DataSets and brings them into R for analysis and visualization. Allows use Spark's of distributed machine learning library from R. Allows extensions that call the full Spark API and provide interfaces to Spark packages. Still relatively early in its maturity

Setup

If you haven't done so already, connect to RStudio Server on your cluster. You can find the process for doing so in Lab 01.

You can find the code for this walkthrough in `03-Sparklyr.R`.

Initializing R For Spark

For SparkR to run properly under RStudio, it requires several environment variables to be properly set. Specifically, SparkR needs to know:

- `SPARK_HOME`: Where Spark is installed.
- `HADOOP_CONF_DIR`: Where the Hadoop configuration files lives.
- `YARN_CONF_DIR`: The configuration directory for YARN.

We'll set those up before we initialize Spark using the following code:

```

if (nchar(Sys.getenv("SPARK_HOME")) < 1) {
  Sys.setenv(SPARK_HOME = "/usr/lib/spark") # or wherever your Spark install lives
}
if (nchar(Sys.getenv("HADOOP_CONF_DIR")) < 1) {
  Sys.setenv(HADOOP_CONF_DIR = "/etc/hadoop/conf") # or wherever your Hadoop lives
}
if (nchar(Sys.getenv("YARN_CONF_DIR")) < 1) {
  Sys.setenv(YARN_CONF_DIR = "/etc/hadoop/conf") # or wherever your YARN config lives
}

```

Now, we'll load the `magrittr` package for data pipelining and then initialize Spark. We'll request 3GB of memory for the driver program in our initialization. This typically takes about 10-15 seconds to complete because it must spin up the Spark executors.

For those R programmers who typically load the `dplyr` library on startup, you don't want to do that when working with Spark because some of the `dplyr` function names conflict with the SparkR API functions, which can result in very confusing errors.

```

library(magrittr)
# please note -- do not load dplyr or you will have function name conflicts

library(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK_HOME"), "R", "lib")))
sparkR.session(master = "yarn", # execution type
  sparkConfig = list(spark.driver.memory = "3g")) # configure driver and executors

```

Getting Started With Sparklyr

First, we'll move some dataframes over to the cluster. Sparklyr has a `copy_to` function to take R data.frames and turn them into Spark tables.

```

# copy some built-in sample data to the Spark cluster
iris_tbl <- copy_to(sc, iris)

```

For our second table, we'll read in a fairly good-sized dataset from the FAA listing all the flights in the USA since 2010. Fortunately, it's stored in *parquet* format, which includes its own schema information. We'll read that into a table `flights` using the sparklyr version of `spark.read.parquet` which uses underscores instead of periods:

```
srcflights <- spark_read_parquet(sc, "flights", "/data/flightdata/parquet-trimmed")
```

We'll then trim that down using a dplyr data pipeline to only the years 2015 and 2016. We'll also eliminate a lot of the columns that we don't care about.

```
recentflights <- srcflights %>% filter(year >= 2015)
flights <- recentflights %>%
  select("year", "month", "dayofmonth", "flightdate", "carrier", "tailnum",
         "deptime", "depdelay", "depdelayminutes", "origin",
         "dest", "arrdelay", "arrdelayminutes",
         "distance", "airtime", "cancelled")
flights_tbl <- flights %>% mutate(date = as.Date(flightdate))
```

Doing Analysis With Sparklyr

The best thing about Sparklyr is that once you have everything set up, you simply can work with Spark dataframes as if they were R data.frames using `dplyr` for your grouping and aggregation.

As an example, now that we've read in our `flights_tbl`, we can easily filter it using `dplyr`. Let's see the first few flights that were delayed by 2 minutes:

```
# filter by departure delay and print the first few records
> flights_tbl %>% filter(depdelay == 2)
# Source:   lazy query [?? x 17]
# Database: spark_connection
   year month dayofmonth flightdate carrier tailnum deptime depdelay depdelayminutes origin
   <int> <int>    <int>    <chr>    <chr>   <chr>   <int>   <int>          <int> <chr>
1  2016     6         8 2016-06-08    AA  N795AA    832     2             2   JFK
2  2016     6         7 2016-06-07    AA  N783AA   1132     2             2   LAX
3  2016     6         6 2016-06-06    AA  N390AA   1112     2             2   DFW
4  2016     6         4 2016-06-04    AA  N391AA    907     2             2   DFW
5  2016     6         1 2016-06-01    AA  N799AA   2132     2             2   LAX
6  2016     6         5 2016-06-05    AA  N792AA   1532     2             2   LAX
7  2016     6        24 2016-06-24    AA  N797AA   2232     2             2   JFK
8  2016     6         1 2016-06-01    AA  N3LVAA   1707     2             2   ORD
9  2016     6         5 2016-06-05    AA  N3DUAA   1707     2             2   ORD
10 2016     6        25 2016-06-25    AA  N3DYAA   1827     2             2   ORD
# ... with more rows, and 5 more variables: arrdelayminutes <int>, distance <int>, airtime <int>
#   date <chr>
```

Now let's do something less trivial. Let's examine if there's a relationship between distance and flight delays. What we'll do is group our data by the airplane tail number, compute the mean distance that airplane flew all year and the mean arrival delay. We'll only look at airplanes that flew at least 20 flights and we'll only look at distances less than 2000 miles.

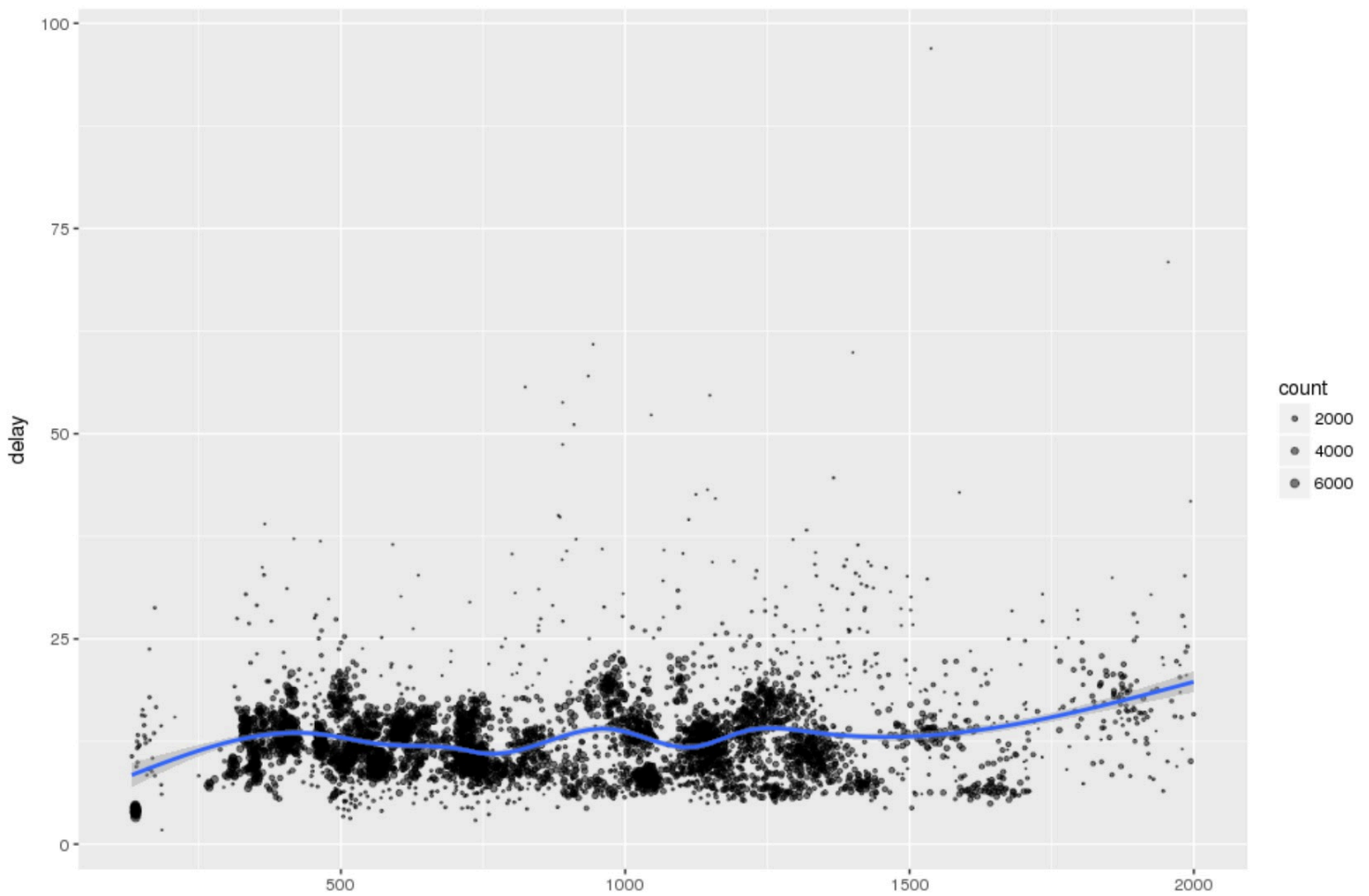
Finally, we'll take that result and plot it using ggplot2 and apply a smoothing regression to it.

Here's the code.

```
# plot data on flight delays
delay <- flights_tbl %>%
  group_by(tailnum) %>%
  summarise(count = n(), dist = mean(distance), delay = mean(arrdelayminutes)) %>%
  filter(count > 20, dist < 2000, !is.na(delay)) %>%
  collect()

library(ggplot2)
ggplot(delay, aes(dist, delay)) +
  geom_point(aes(size = count), alpha = 1/2) +
  geom_smooth() +
  scale_size_area(max_size = 2)
```

And it produces this fairly attractive plot:



Machine Learning With Sparklyr

Sparklyr also includes some nice interfaces to Spark's machine learning library SparkML. This example is fairly trivial, but it takes a standard dataset about cars called `mtcars` and applies a linear regression to it using `ml_linear_regression` that attempts to relate the miles per gallon `mpg` to the vehicle weight `wt` and the number of cylinders `cyl`.

```

# Machine learning example using Spark MLlib

# copy the mtcars data into Spark
mtcars_tbl <- copy_to(sc, mtcars)

# transform the data set, and then partition into 'training', 'test'
partitions <- mtcars_tbl %>%
  filter(hp >= 100) %>%
  mutate(cyl8 = cyl == 8) %>%
  sdf_partition(training = 0.5, test = 0.5, seed = 1099)

# fit a linear regression model to the training dataset
fit <- partitions$training %>%
  ml_linear_regression(response = "mpg", features = c("wt", "cyl"))
fit

summary(fit)

# get the 10th row in test data
car <- tbl_df(partitions$test) %>% slice(10)
# predict the mpg
predicted_mpg <- car$cyl * fit$coefficients["cyl"] + car$wt * fit$coefficients["wt"] + fit$coe-

# print the original and the predicted
sprintf("original mpg = %s, predicted mpg = %s", car$mpg, predicted_mpg)

```

Caveats

Sparklyr is rather slow compared to the SparkR API. However, the fact that it allows data scientists and analysts to exploit frameworks they already know (specifically `dplyr`) into the Spark world means it bears watching.

This step concludes the lab.