



Cloudera Custom Training: Hands-On Exercises

General Notes.....	3
Hands-On Exercise: Run a YARN Job	6
Hands-On Exercise: Explore RDDs Using the Spark Shell	12
Hands-On Exercise: Process Data Files with Apache Spark	22
Hands-On Exercise: Use Pair RDDs to Join Two Datasets	26
Hands-On Exercise: Write and Run an Apache Spark Application	31
Hands-On Exercise: Configure an Apache Spark Application.....	36
Hands-On Exercise: View Jobs and Stages in the Spark Application UI.....	41
Hands-On Exercise: Persist an RDD.....	47
Hands-On Exercise: Implement an Iterative Algorithm with Apache Spark	50
Hands-On Exercise: Use Apache Spark SQL for ETL.....	54

Hands-On Exercise: Produce and Consume Apache Kafka Messages	60
Hands-On Exercise: Collect Web Server Logs with Apache Flume	63
Hands-On Exercise: Write an Apache Spark Streaming Application	66
Hands-On Exercise: Process Multiple Batches with Apache Spark Streaming	71
Hands-On Exercise: Process Apache Kafka Messages with Apache Spark Streaming	76
Appendix A: Enabling Jupyter Notebook for PySpark.....	81
Appendix B: Managing Services on the Course Virtual Machine	84

General Notes

Cloudera's training courses use a Virtual Machine running the CentOS Linux distribution. This VM has CDH installed in pseudo-distributed mode. Pseudo-distributed mode is a method of running Hadoop whereby all Hadoop daemons run on the same machine. It is, essentially, a cluster consisting of a single machine. It works just like a larger Hadoop cluster; the only key difference is that the HDFS block replication factor is set to 1, since there is only a single DataNode available.

Points to Note while Working in the Virtual Machine

- The Virtual Machine (VM) is set to log in as the user `training` automatically. If you log out, you can log back in as the user `training` with the password `training`.
- If you need it, the root password is `training`. You may be prompted for this if, for example, you want to change the keyboard layout. In general, you should not need this password since the `training` user has unlimited sudo privileges.
- In some command-line steps in the exercises, you will see lines like this:

```
$ hdfs dfs -put united_states_census_data_2010 \
/user/training/example
```

The dollar sign (\$) at the beginning of each line indicates the Linux shell prompt. The actual prompt will include additional information (for example, `[training@localhost training_materials]$`) but this is omitted from these instructions for brevity.

The backslash (\) at the end of the first line signifies that the command is not completed, and continues on the next line. You can enter the code exactly as shown (on two lines), or you can enter it on a single line. If you do the latter, you should *not* type in the backslash.

- Although most students are comfortable using UNIX text editors like `vi` or `emacs`, some might prefer a graphical text editor. To invoke the graphical editor from the command line, type `gedit` followed by the path of the file you wish to edit.

Appending `&` to the command allows you to type additional commands while the editor is still open. Here is an example of how to edit a file named `myfile.txt`:

```
$ gedit myfile.txt &
```

Points to Note during the Exercises

Directories

- The main directory for the exercises is `~/training_materials/devsh/exercises`. Each directory under that one corresponds to an exercise or set of exercises—this is referred to in the instructions as “the exercise directory.” Any scripts or files required for the exercise (other than data) are in the exercise directory.
- Within each exercise directory you may find the following subdirectories:
 - `solution`—This contains solution code for each exercise.
 - `stubs`—A few of the exercises depend on provided starter files containing skeleton code.
 - Maven project directories—For exercises for which you must write Scala classes, you have been provided with preconfigured Maven project directories. Within these projects are two packages: `stubs`, where you will do your work using starter skeleton classes; and `solution`, containing the solution class.
- Data files used in the exercises are in `~/training_materials/data`. Usually you will upload the files to HDFS before working with them.
- The VM defines a few environment variables that are used in place of longer paths in the instructions. Since each variable is automatically replaced with its corresponding values when you run commands in the terminal, this makes it easier and faster for you to enter a command.
 - The two environment variables for this course are `$DEVSH` and `$DEVDATA`. Under `$DEVSH` you can find `exercises`, `examples`, and `scripts`.
 - You can always use the `echo` command if you would like to see the value of an environment variable:

```
$ echo $DEVSH
```

Step-by-Step Instructions

As the exercises progress, and you gain more familiarity with the tools and environment, we provide fewer step-by-step instructions; as in the real world, we merely give you a requirement and it's up to you to solve the problem! You should feel free to refer to the hints or solutions provided, ask your instructor for assistance, or consult with your fellow students.

Bonus Exercises

There are additional challenges for some of the hands-on exercises. If you finish the main exercise, please attempt the additional steps.

Catch-Up Script

If you are unable to complete an exercise, we have provided a script to catch you up automatically. Each exercise has instructions for running the catch-up script, where applicable.

```
$ $DEVSH/scripts/catchup.sh
```

The script will prompt for which exercise you are starting; it will set up all the required data as if you had completed all the previous exercises.

Warning: If you run the catch up script, you may lose your work. (For example, all data will be deleted from HDFS.)

Hands-On Exercise: Run a YARN Job

Files and Data Used in This Exercise

Exercise directory: `$DEVSH/exercises/yarn`

Data files (HDFS): `/loudacre/kb`

In this exercise, you will submit an application to the YARN cluster, and monitor the application using both the Hue Job Browser and the YARN Web UI.

The application you will run is provided for you. It is a simple Spark application written in Python that counts the occurrence of words in Loudacre's customer service Knowledge Base (which you uploaded in a previous exercise). The focus of this exercise is not on what the application does, but on how YARN distributes tasks in a job across a cluster, and how to monitor an application and view its log files.

IMPORTANT: For this custom course, you *must* run the following command to prepare for this exercise before continuing:

```
$ $DEVSH/scripts/catchup.sh
```

Exploring the YARN Cluster

1. Visit the YARN Resource Manager (RM) UI in Firefox using the provided bookmark, or by going to URL `http://localhost:8088/`.

No jobs are currently running so the current view shows the cluster "at rest."

Who Is Dr. Who?

You may notice that YARN says you are logged in as `dr.who`. This is what is displayed when user authentication is disabled for the cluster, as it is on the training VM. If user authentication were enabled, you would have to log in as a valid user to view the YARN UI, and your actual username would be displayed, together with user metrics such as how many applications you had run, how much system resources your applications used and so on.

- Take note of the values in the Cluster Metrics section, which displays information such as the number of applications running currently, previously run, or waiting to run; the amount of memory used and available; and how many worker nodes are in 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	2 GB	0 B	0	2	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

ID

User

Name

Application Type

Queue

StartTime

FinishTime

State

FinalStatus

Progress

Tracking UI

No data available in table

Showing 0 to 0 of 0 entries

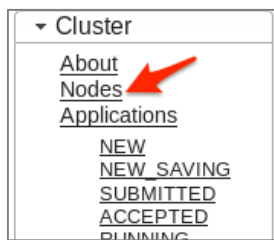
First

Previous

Next

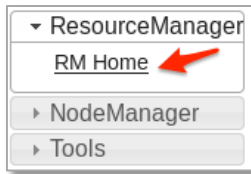
Last

- Click the **Nodes** link in the Cluster menu on the left. The bottom section will display a list of worker nodes in the cluster. The pseudo-distributed cluster used for training has only a single node, which is running on the local machine. In the real world, this list would show multiple worker nodes.



- Click the **Node HTTP Address** to open the Node Manager UI for that node. This displays statistics about the selected node, including amount of available memory, currently running applications (there are none), and so on.

- To return to the Resource Manager, click **ResourceManager** → **RM Home** on the left.



Submitting an Application to the YARN Cluster

- In a terminal window, change to the exercise directory:

```
$ cd $DEVSH/exercises/yarn
```

- Run the example `wordcount.py` program on the YARN cluster to count the frequency of words in the Knowledge Base dataset:

```
$ spark-submit --master yarn-client \
  wordcount.py /loudacre/kb/*
```

The `spark-submit` command is used to submit a Spark program for execution on the cluster. Since Spark is managed by YARN on the course VM, this gives us the opportunity to see how the YARN UI displays information about a running job. For now, focus on learning about the YARN UI.

While the application is running, continue with the next steps. If it completes before you finish the exercise, go to the terminal, press the up arrow until you get to the `spark-submit` command again, and rerun the application.

Viewing the Application in the Hue Job Browser

- Go to Hue in Firefox, and select the Job Browser. (Depending on the width of your browser, you may see the whole label, or just the icon.)



- The Job Browser displays a list of currently running and recently completed applications. (If you don't see the application you just started, wait a few seconds, the page will automatically reload; it can take some time for the application to be accepted and start running.) Review the entry for the current job.

Username

training

Text

Search for text

Succeeded

Running

Failed

Killed

Logs	ID	Name	Application Type	Status	User	Maps	Reduces	Queue	Priority	Duration	Submitted	
	1452185447586_0007	wordcount.py	SPARK	RUNNING	training	100%	100%	root.training	N/A	1m:32s	01/21/16 07:06:56	Kill

This page allows you to click the application ID to see details of the running application, or to kill a running job. (Do not do that now though!)

Viewing the Application in the YARN UI

To get a more detailed view of the cluster, use the YARN UI.

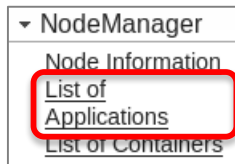
- Reload the YARN RM page in Firefox. You will now see the application you just started in the bottom section of the RM home page.

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
1	0	1	0	2	2 GB	2 GB	1 GB	2	2	1	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	VCores Pending	VCores Reserved	VCores Reserved
0	0	1	0	0	0	0	0 B	0 B	0 B	0	0	0	0	0	0
Show 20 ▾ entries Search: <input type="text"/>															
ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI					
application_1428590029950_0001	training	PythonWordCount	SPARK	root.training	Thu Apr 9 07:53:57 -0700 2015	N/A	RUNNING	UNDEFINED	<div><div></div></div>	ApplicationMaster					
Showing 1 to 1 of 1 entries First Previous 1 Next Last															

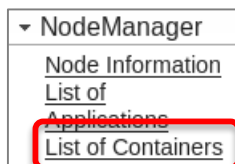
- As you did in the first exercise section, select **Nodes**.
- Select the **Node HTTP Address** to open the Node Manager UI.

Node Labels	Rack	Node State	Node Address	Node HTTP Address	Last health-update
	/default-rack	RUNNING	localhost:59233	localhost:8042	Mon Aug 24 10:41:13 -0700 2015

13. Now that an application is running, you can click **List of Applications** to see the application you submitted.



14. If your application is still running, try clicking on **List of Containers**.



This will display the containers the Resource Manager has allocated on the selected node for the current application. (No containers will show if no applications are running; if you missed it because the application completed, you can run the application again. In the terminal window, use the up arrow key to recall previous commands.)

Show 20 entries		Search:
ContainerId	ContainerState	logs
container_1428590029950_0001_01_000001	RUNNING	logs
container_1428590029950_0001_01_000002	RUNNING	logs
Showing 1 to 2 of 2 entries		First Previous 1 Next Last

Viewing the Application Using the yarn Command

15. Open a second terminal window.

Tip: Resize the terminal window to be as wide as possible to make it easier to read the command output.

16. View the list of currently running applications.

```
$ yarn application -list
```

If your application is still running, you should see it listed, including the application ID (such as `application_1469799128160_0001`), the application name (`PythonWordCount`), the type (`SPARK`), and so on.

If there are no applications on the list, your application has probably finished running. By default, only current applications are included. Use the `-appStates ALL` option to include all applications in the list:

```
$ yarn application -list -appStates ALL
```

17. Take note of your application's ID (such as `application_1469799128160_0001`), and use it in place of `app-id` in the command below to get a detailed status report on the application.

```
$ yarn application -status app-id
```

This is the end of the exercise

Hands-On Exercise: Explore RDDs Using the Spark Shell

Files and Data Used in This Exercise

Exercise directory: \$DEVSH/exercises/spark-shell

Data files (local): \$DEVDATA/frostroad.txt
 \$DEVDATA/weblogs/*

In this exercise, you will use the Spark shell to work with RDDs.

You will start by viewing and bookmarking the Spark documentation in your browser. Then you will start the Spark shell and read a simple text file into a Resilient Distributed Data Set (RDD). Finally, you will copy the weblogs data set to HDFS and use RDDs to transform the data.

Important: If you did not complete the previous exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Viewing the Spark Documentation

1. Start Firefox in your Virtual Machine and visit the Spark documentation on your local machine using the provided bookmark.
2. From the **Programming Guides** menu, select the **Spark Programming Guide**. Briefly review the guide. You may wish to bookmark the page for later review.
3. From the **API Docs** menu, select either **Scala** or **Python**, depending on your language preference. Bookmark the API page for use during class. Later exercises will refer you to this documentation.

Starting the Spark Shell

You may choose to do the remaining steps in this exercise using either Scala or Python. Follow the instructions below for Python, or skip to the next section for Scala.

Note: Instructions for Python are provided in **blue**, while instructions for Scala are in **red**.

Starting the Python Spark Shell

Follow these instructions if you are using Python to complete this exercise. Otherwise, skip this section and continue with Starting the Scala Spark Shell.

4. In a terminal window, start the `pyspark` shell:

```
$ pyspark
```

You may get several `INFO` and `WARNING` messages, which you can disregard. If you don't see the `In[n]>` prompt after a few seconds, press `Enter` a few times to clear the screen output.

5. Spark creates a `SparkContext` object for you called `sc`. Make sure the object exists:

```
pyspark> sc
```

Note on Shell Prompt

To help you keep track of which shell is being referenced in the instructions, the prompt will be shown here as either `pyspark>` or `scala>`. The actual prompt will vary depending on which version of Python or Scala you are using and what command number you are on.

Pyspark will display information about the `sc` object such as

```
<pyspark.context.SparkContext at 0x2724490>
```

6. Using command completion, you can see all the available `SparkContext` methods: type `sc.` (`sc` followed by a dot) and then the [TAB] key.
7. You can exit the shell by pressing `Ctrl+D` or by typing `exit`. However, stay in the shell for now to complete the remainder of this exercise.

Starting the Scala Spark Shell

Follow these instructions if you are using Scala to complete this exercise. Otherwise, skip this section and continue with Reading and Displaying a Text File. (Don't try to run both a Scala and a Python shell at the same time; doing so will cause errors and slow down your machine.)

8. In a terminal window, start the Scala Spark shell:

```
$ spark-shell
```

You may get several `INFO` and `WARNING` messages, which you can disregard. If you don't see the `scala>` prompt after a few seconds, press `Enter` a few times to clear the screen output.

9. Spark creates a `SparkContext` object for you called `sc`. Make sure the object exists:

```
scala> sc
```

Note on Shell Prompt

To help you keep track of which shell is being referenced in the instructions, the prompt will be shown here as either `pyspark>` or `scala>`. The actual prompt will vary depending on which version of Python or Scala you are using and which command number you are on.

Scala will display information about the `sc` object such as:

```
res0: org.apache.spark.SparkContext =
org.apache.spark.SparkContext@2f0301fa
```

10. Using command completion, you can see all the available SparkContext methods: type `sc.` (`sc` followed by a dot) and then the [TAB] key.
11. You can exit the shell at any time by typing `sys.exit` or pressing `Ctrl+D`. However, stay in the shell for now to complete the remainder of this exercise.

Reading and Displaying a Text File (Python or Spark)

12. Review the simple text file you will be using by viewing (without editing) the file in a text editor in a separate window (not the Spark shell). The file is located at: `$DEVDATA/frostroad.txt`.
13. Define an RDD to be created by reading in the test file on the local filesystem. Use the first command if you are using Python, and the second one if you are using Scala. (You only need to complete the exercises in Python *or* Scala. Do not attempt to run both shells at the same time; it will result in error messages and slow down your machine.)

```
pyspark> myrdd = sc.textFile(\
"file:/home/training/training_materials/\
data/frostroad.txt")
```

```
scala> val myrdd = sc.textFile(
"file:/home/training/training_materials/data/frostroad.txt")
```

- **Note:** In subsequent instructions, both Python and Scala commands will be shown but not noted explicitly; Python shell commands are in blue and preceded with `pyspark>`, and Scala shell commands are in red and preceded with `scala>`.

14. Spark has not yet read the file. It will not do so until you perform an operation on the RDD. Try counting the number of lines in the dataset:

```
pyspark> myrdd.count()
```

```
scala> myrdd.count()
```

The `count` operation causes the RDD to be materialized (created and populated). The number of lines (23) should be displayed, for example:

```
Out[2]: 23 (Python) or  
res1: Long = 23 (Scala)
```

15. Try executing the `collect` operation to display the data in the RDD. Note that this returns and displays the entire dataset. This is convenient for very small RDDs like this one, but be careful using `collect` for more typical large datasets.

```
pyspark> myrdd.collect()
```

```
scala> myrdd.collect()
```

16. Using command completion, you can see all the available transformations and operations you can perform on an RDD. Type `myrdd.` and then the [TAB] key.

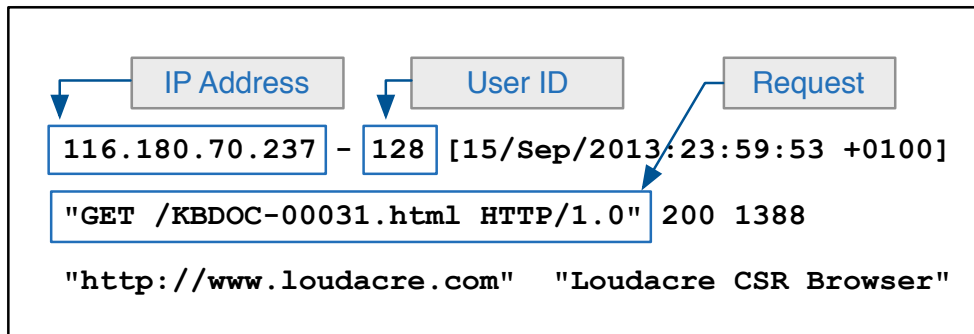
A Tip for PySpark Users: Controlling Log Messages

You may have noticed that by default, PySpark displays many log messages tagged `INFO`. If you find this output distracting, you may temporarily override the default logging level by using the command: `sc.setLogLevel("WARN")`. You can return to the prior level of logging with `sc.setLogLevel("INFO")` or by restarting the PySpark shell. Configuring logging will be covered later in the course.

Exploring the Loudacre Web Log Files

17. In this section you will be using data in

`~/training_materials/data/weblogs`. Review one of the `.log` files in the directory. Note the format of the lines:



18. In the previous steps you used a data file residing on the local Linux filesystem. In the real world, you will almost always be working with distributed data, such as files stored on the HDFS cluster, instead. Copy the dataset from the local filesystem to the `loudacre` HDFS directory. In a separate terminal window (not your Spark shell) execute:

```
$ hdfs dfs -put \
~/training_materials/data/weblogs/ /loudacre/
```

19. In the Spark shell, set a variable for the data files so you do not have to retype the path each time.

```
pyspark> logfiles="/loudacre/weblogs/*"
```

```
scala> val logfiles="/loudacre/weblogs/*"
```

20. Create an RDD from the data file.

```
pyspark> logsRDD = sc.textFile(logfiles)
```

```
scala> val logsRDD = sc.textFile(logfiles)
```

21. Create an RDD containing only those lines that are requests for JPG files.

```
pyspark> jpglogsRDD=\
logsRDD.filter(lambda line: ".jpg" in line)
```

```
scala> val jpglogsRDD=
logsRDD.filter(line => line.contains(".jpg"))
```

22. View the first 10 lines of the data using take:

```
pyspark> jpglogsRDD.take(10)
```

```
scala> jpglogsRDD.take(10)
```

23. Sometimes you do not need to store intermediate objects in a variable, in which case you can combine the steps into a single line of code. For instance, execute this single command to count the number of JPG requests. (The correct number is 64978.)

```
pyspark> sc.textFile(logfiles).filter(lambda line: \
".jpg" in line).count()
```

```
scala> sc.textFile(logfiles).
filter(line => line.contains(".jpg")).count()
```

24. Now try using the map function to define a new RDD. Start with a simple map that returns the length of each line in the log file.

```
pyspark> logsRDD.map(lambda line: len(line)).take(5)
```

```
scala> logsRDD.map(line => line.length).take(5)
```

This prints out an array of five integers corresponding to the first five lines in the file. (The correct result is: 151, 143, 154, 147, 160.)

25. That is not very useful. Instead, try mapping to an array of words for each line:

```
pyspark> logsRDD \
    .map(lambda line: line.split(' ')).take(5)
```

```
scala> logsRDD.map(line => line.split(' ')).take(5)
```

This time Spark prints out five arrays, each containing the words in the corresponding log file line.

26. Now that you know how `map` works, define a new RDD containing just the IP addresses from each line in the log file. (The IP address is the first “word” in each line.)

```
pyspark> ipsRDD = \
    logsRDD.map(lambda line: line.split(' ')[0])
pyspark> ipsRDD.take(5)
```

```
scala> val ipsRDD =
    logsRDD.map(line => line.split(' ')[0])
scala> ipsRDD.take(5)
```

27. Although `take` and `collect` are useful ways to look at data in an RDD, their output is not very readable. Fortunately, though, they return arrays, which you can iterate through:

```
pyspark> for ip in ipsRDD.take(10): print ip
```

```
scala> ipsRDD.take(10).foreach(println)
```

28. Finally, save the list of IP addresses as a text file:

```
pyspark> ipsRDD.saveAsTextFile("/loudacre/iplist")
```

```
scala> ipsRDD.saveAsTextFile("/loudacre/iplist")
```

- **Note:** If you re-run this command, you will not be able to save to the same directory because it already exists. Be sure to delete the directory using either the `hdfs` command (in a separate terminal window) or the Hue file browser first.

29. In a terminal window or the Hue file browser, list the contents of the `/loudacre/iplist` folder. You should see multiple files, including several `part-xxxxxx` files, which are the files containing the output data. “Part” (partition) files are numbered because there may be results from multiple tasks running on the cluster. Review the contents of one of the files to confirm that they were created correctly.

Bonus Exercise

Use RDD transformations to create a dataset consisting of the IP address and corresponding user ID for each request for an HTML file. (Disregard requests for other file types.) The user ID is the third field in each log file line.

Display the data in the form *ipaddress/userid*, such as:

```
165.32.101.206/8
100.219.90.44/102
182.4.148.56/173
246.241.6.175/45395
175.223.172.207/4115
...
```

This is the end of the exercise

Hands-On Exercise: Process Data Files with Apache Spark

Files and Data Used in This Exercise:

Exercise directory:	<code>\$DEVSH/exercises/spark-etl</code>
Data files (local):	<code>\$DEVDATA/activations/*</code> <code>\$DEVDATA/devicestatus.txt (Bonus)</code>
Stubs:	<code>ActivationModels.pyspark</code> <code>ActivationModels.scalaspark</code>

In this exercise, you will parse a set of activation records in XML format to extract the account numbers and model names.

One of the common uses for Spark is doing data Extract/Transform/Load operations. Sometimes data is stored in line-oriented records, like the web logs in the previous exercise, but sometimes the data is in a multi-line format that must be processed as a whole file. In this exercise, you will practice working with file-based instead of line-based formats.

Important: If you did not run the course catch-up script for a previous exercise, do so now and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Reviewing the API Documentation for RDD Operations

1. Visit the Spark API page, which you might have bookmarked; if not, you can find it from the **Spark Doc** bookmark. Follow the link for the `RDD` class and review the list of available operations. (In the Scala API, the link will be near the top of the main window; in Python scroll down to the Core Classes area.)

Reviewing the Data

2. Review the data on the local Linux filesystem in the directory `$DEVDATA/activations`. Each XML file contains data for all the devices activated by customers during a specific month.

Sample input data:

```
<activations>
  <activation timestamp="1225499258" type="phone">
    <account-number>316</account-number>
    <device-id>
      d61b6971-33e1-42f0-bb15-aa2ae3cd8680
    </device-id>
    <phone-number>5108307062</phone-number>
    <model>iFruit 1</model>
  </activation>
  ...
</activations>
```

3. Copy the entire `activations` directory to `/loudacre` in HDFS.

```
$ hdfs dfs -put $DEVDATA/activations /loudacre/
```

Processing the Files

Follow the steps below to write code to go through a set of activation XML files and extract the account number and device model for each activation, and save the list to a file as `account_number:model`.

The output will look something like:

```
1234:iFruit 1
987:Sorrento F00L
4566:iFruit 1
...
```

4. Start with the `ActivationModels` stub script in the exercise directory: `$DEVSH/exercises/spark-etl`. (A stub is provided for Scala and Python; use whichever language you prefer.) Note that for convenience you have been provided with functions to parse the XML, as that is not the focus of this exercise. Copy the stub code into the Spark shell of your choice.
5. Use `wholeTextFiles` to create an RDD from the activations dataset. The resulting RDD will consist of tuples, in which the first value is the name of the file, and the second value is the contents of the file (XML) as a string.
6. Each XML file can contain many activation records; use `flatMap` to map the contents of each file to a collection of XML records by calling the provided `getActivations` function. `getActivations` takes an XML string, parses it, and returns a collection of XML records; `flatMap` maps each record to a separate RDD element.
7. Map each activation record to a string in the format `account-number:model`. Use the provided `getAccount` and `getModel` functions to find the values from the activation record.
8. Save the formatted strings to a text file in the directory `/loudacre/account-models`.

Bonus Exercise

If you have more time, attempt the following extra bonus exercise:

Another common part of the ETL process is data scrubbing. In this bonus exercise, you will process data in order to get it into a standardized format for later processing.

Review the contents of the file `$DEVDATA/devicestatus.txt`. This file contains data collected from mobile devices on Loudacre's network, including device ID, current status, location, and so on. Because Loudacre previously acquired other mobile providers' networks, the data from different subnetworks has a different format. Note that the records in this file have different field delimiters: some use commas, some use pipes (`|`), and so on. Your task is the following:

- Upload the `devicestatus.txt` file to HDFS.
- Determine which delimiter to use (hint: the character at position 19 is the first use of the delimiter).
- Filter out any records which do not parse correctly (hint: each record should have exactly 14 values).
- Extract the date (first field), model (second field), device ID (third field), and latitude and longitude (13th and 14th fields respectively).
- The second field contains the device manufacturer and model name (such as `Ronin S2`). Split this field by spaces to separate the manufacturer from the model (for example, manufacturer `Ronin`, model `S2`). Keep just the manufacturer name.
- Save the extracted data to comma-delimited text files in the `/loudacre/devicestatus_etl` directory on HDFS.
- Confirm that the data in the file(s) was saved correctly.

The solutions to the bonus exercise are in `$DEVSH/exercises/spark-etl/solution/bonus`.

This is the end of the exercise

Hands-On Exercise: Use Pair RDDs to Join Two Datasets

Files and Data Used in This Exercise:

Exercise directory: \$DEVSH/exercises/spark-pairs

Data files (HDFS): /loudacre/weblogs/*
 /loudacre/accounts/*

In this exercise, you will continue exploring the Loudacre web server log files, as well as the Loudacre user account data, using key-value pair RDDs.

IMPORTANT: For this custom course, you *must* run the following command to prepare for this exercise before continuing:

```
$ $DEVSH/scripts/catchup.sh
```

Exploring Web Log Files

Continue working with the web log files, as in earlier exercises.

Tip: In this exercise, you will be reducing and joining large datasets, which can take a lot of time. You may wish to perform the exercises below using a smaller dataset, consisting of only a few of the web log files, rather than all of them. Remember that you can specify a wildcard; `textFile("/loudacre/weblogs/*2.log")` would include only filenames ending with `2.log`.

1. Using map-reduce logic, count the number of requests from each user.
 - a. Use `map` to create a pair RDD with the user ID as the key and the integer 1 as the value. (The user ID is the third field in each line.) Your data will look something like this:

(userid, 1)
(userid, 1)
(userid, 1)
...

- b. Use `reduceByKey` to sum the values for each user ID. Your RDD data will be similar to this:

(userid, 5)
(userid, 7)
(userid, 2)
...

2. Use `countByKey` to determine how many users visited the site for each frequency. That is, how many users visited once, twice, three times, and so on.

- a. Use `map` to reverse the key and value, like this:

(5, userid)
(7, userid)
(2, userid)
...

- b. Use the `countByKey` action to return a map of *frequency:user-count* pairs.

3. Create an RDD where the user ID is the key, and the value is the list of all the IP addresses that user has connected from. (IP address is the first field in each request line.)

- Hint: Map to (userid, ipaddress) and then use `groupByKey`.

(userid, 20.1.34.55)
(userid, 245.33.1.1)
(userid, 65.50.196.141)
...



(userid, [20.1.34.55, 74.125.239.98])
(userid, [75.175.32.10, 245.33.1.1, 66.79.233.99])
(userid, [65.50.196.141])
...

Joining Web Log Data with Account Data

Review the data located in `/loudacre/accounts` containing Loudacre's customer account data (previously imported from MySQL to HDFS using Sqoop). The first field in each line is the user ID, which corresponds to the user ID in the web server logs. The other fields include account details such as creation date, first and last name, and so on.

4. Join the accounts data with the weblog data to produce a dataset keyed by user ID which contains the user account information and the number of website hits for that user.
 - a. Create an RDD, based on the accounts data, consisting of key/value-array pairs: (userid, [values...])

(userid1, [userid1, 2008-11-24 10:04:08, \N, Cheryl, West, 4905 Olive Street, San Francisco, CA, ...])
(userid2, [userid2, 2008-11-23 14:05:07, \N, Elizabeth, Kerns, 4703 Eva Pearl Street, Richmond, CA, ...])
(userid3, [userid3, 2008-11-02 17:12:12, 2013-07-18 16:42:36, Melissa, Roman, 3539 James Martin Circle, Oakland, CA, ...])
...

- b. Join the pair RDD with the set of user-id/hit-count pairs calculated in the first step.

<code>(userid1, ([userid1, 2008-11-24 10:04:08, \N, Cheryl, West, 4905 Olive Street, San Francisco, CA, ...], 4))</code>
<code>(userid2, ([userid2, 2008-11-23 14:05:07, \N, Elizabeth, Kerns, 4703 Eva Pearl Street, Richmond, CA, ...], 8))</code>
<code>(userid3, ([userid3, 2008-11-02 17:12:12, 2013-07-18 16:42:36, Melissa, Roman, 3539 James Martin Circle, Oakland, CA, ...], 1))</code>
...

- c. Display the user ID, hit count, and first name (4th value) and last name (5th value) for the first 5 elements. The output should look similar to this:

```
userid1 6 Rick Hopper
userid2 8 Lucio Arnold
userid3 2 Brittany Parrott
...
```

Managed Memory Leak Error Message

When executing a join operation in Scala, you may see an error message such as this:

```
ERROR Executor: Managed memory leak detected
This message is a Spark bug and can be disregarded.
```

Bonus Exercises

If you have more time, attempt the following extra bonus exercises:

1. Use `keyBy` to create an RDD of account data with the postal code (9th field in the CSV file) as the key.

Tip: Assign this new RDD to a variable for use in the next bonus exercise.

2. Create a pair RDD with postal code as the key and a list of names (Last Name, First Name) in that postal code as the value.

- Hint: First name and last name are the 4th and 5th fields respectively.
 - Optional: Try using the `mapValues` operation.
3. Sort the data by postal code, then for the first five postal codes, display the code and list the names in that postal zone. For example:

```
--- 85003
Jenkins,Thad
Rick,Edward
Lindsay,Ivy
...
--- 85004
Morris,Eric
Reiser,Hazel
Gregg,Alicia
Preston,Elizabeth
...
```

This is the end of the exercise

Hands-On Exercise: Write and Run an Apache Spark Application

Files and Data Used in This Exercise:

Exercise directory: \$DEVSH/exercises/spark-application

Data files (HDFS): /loudacre/weblogs

Scala project:

\$DEVSH/exercises/spark-application/countjpgs_project

Scala classes: stubs.CountJPGs
 solution.CountJPGs

Python stub: CountJPGs.py

Python solution:

\$DEVSH/exercises/spark-application/python-
solution/CountJPGs.py

In this exercise, you will write your own Spark application instead of using the interactive Spark shell application.

Write a simple program that counts the number of JPG requests in a web log file. The name of the file should be passed to the program as an argument.

This is the same task as in the “Explore RDDs Using the Spark Shell” exercise. The logic is the same, but this time you will need to set up the `SparkContext` object yourself.

Depending on which programming language you are using, follow the appropriate set of instructions below to write a Spark program.

Before running your program, be sure to exit from the Spark shell.

Important: This exercise depends on a previous exercise: “Explore RDDs Using the Spark Shell.” If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Writing a Spark Application in Python

Editing Python Files

You may use any text editor you wish. If you don't have an editor preference, you may wish to use gedit, which includes language-specific support for Python.

1. If you are using Python, follow these instructions; otherwise, skip this section and continue to Writing a Spark Application in Scala below.
2. A simple stub file to get started has been provided in the exercise project: `$DEVSH/exercises/spark-application/CountJPGs.py`. This stub imports the required Spark class and sets up your main code block. Open the stub file in an editor.
3. Create a `SparkContext` object using the following code:

```
sc = SparkContext()
```

4. In the body of the program, load the file passed in to the program, count the number of JPG requests, and display the count. You may wish to refer back to the “Explore RDDs Using the Spark Shell” exercise for the code to do this.
5. At the end of the application, be sure to stop the Spark context:

```
sc.stop()
```

6. Change to the exercise working directory, then run the program, passing the name of the log file to process, for example:


```
$ cd $DEVSH/exercises/spark-application/
$ spark-submit CountJPGs.py /loudacre/weblogs/*
```

7. Once the program completes, you might need to scroll up to see your program output. (The correct number of JPG requests is 64978.)
8. Skip the section below on writing a Spark application in Scala and continue with Submitting a Spark Application to the Cluster.

Writing a Spark Application in Scala

Editing Scala Files

You may use any text editor you wish. If you don't have an editor preference, you may wish to use gedit, which includes language-specific support for Scala. If you prefer to work in an IDE, Eclipse is included and configured for the Scala projects in the course. However, teaching use of Eclipse is beyond the scope of this course.

A Maven project to get started has been provided: `$DEVSH/exercises/spark-application/countjpgs_project`.

9. Edit the Scala class defined in `CountJPGs.scala` in `src/main/scala/stubs/`.
10. Create a `SparkContext` object using the following code:

```
val sc = new SparkContext()
```

11. In the body of the program, load the file passed to the program, count the number of JPG requests, and display the count. You may wish to refer back to the "Explore RDDs Using the Spark Shell" exercise for the code to do this.
12. At the end of the application, be sure to stop the Spark context:

```
sc.stop
```

13. Change to the project directory, then build your project using the following command:

```
$ cd \
  $DEVSH/exercises/spark-application/countjpgs_project
$ mvn package
```

14. If the build is successful, Maven will generate a JAR file called `countjpgs-1.0.jar` in `countjpgs-project/target`. Run the program using the following command:

```
$ spark-submit \
  --class stubs.CountJPGs \
  target/countjpgs-1.0.jar /loudacre/weblogs/*
```

15. Once the program completes, you might need to scroll up to see your program output. (The correct number of JPG requests is 64978.)

Submitting a Spark Application to the Cluster

In the previous section, you ran a Python or Scala Spark application using `spark-submit`. By default, `spark-submit` runs the application locally. In this section, run the application on the YARN cluster instead.

16. Re-run the program, specifying the cluster master in order to run it on the cluster. Use one of the commands below depending on whether your application is in Python or Scala.

To run Python:

```
$ spark-submit \
  --master yarn-client \
  CountJPGs.py /loudacre/weblogs/*
```

To run Scala:

```
$ spark-submit \
  --class stubs.CountJPGs \
  --master yarn-client \
  target/countjpgs-1.0.jar /loudacre/weblogs/
```

17. After starting the application, open Firefox and visit the YARN Resource Manager UI using the provided bookmark (or going to URL <http://localhost:8088/>). While the application is running, it appears in the list of applications something like this:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1428074921193_0030	training	solution.CountJPGs	SPARK	root.training	Mon Apr 6 07:47:18 -0700 2015	N/A	RUNNING	UNDEFINED	<div></div>	ApplicationMaster

After the application has completed, it will appear in the list like this:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking
application_1428074921193_0030	training	solution.CountJPGs	SPARK	root.training	Mon Apr 6 07:47:18 -0700 2015	Mon Apr 6 07:48:09 -0700 2015	FINISHED	SUCCEEDED	<div></div>	History

This is the end of the exercise

Hands-On Exercise: Configure an Apache Spark Application

Files and Data Used in This Exercise:

Exercise directory: \$DEVSH/exercises/spark-application

Data files (HDFS): /loudacre/weblogs

Scala project:

\$DEVSH/exercises/spark-application/countjpgs_project

Scala classes: stubs.CountJPGs
 solution.CountJPGs

Python stub: CountJPGs.py

Python solution:

\$DEVSH/exercises/spark-application/python-
solution/CountJPGs.py

In this exercise, you will practice setting various Spark configuration options.

You will work with the CountJPGs program you wrote in the prior exercise.

Important: If you did not run the course catch-up script for “Use Pair RDDs to Join Two Datasets” or an exercise after that one, do so now and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Setting Configuration Options at the Command Line

1. Change to the correct directory (if necessary) and re-run the CountJPGs Python or Scala program you wrote in the previous exercise, this time specifying an application name. For example:

```
$ cd $DEVSH/exercises/spark-application/
$ spark-submit --master yarn-client \
  --name 'Count JPGs' \
  CountJPGs.py /loudacre/weblogs/*
```

```
$ cd \
$DEVSH/exercises/spark-application/countjpgs_project
$ spark-submit --class stubs.CountJPGs \
  --master yarn-client \
  --name 'Count JPGs' \
  target/countjpgs-1.0.jar /loudacre/weblogs/*
```

2. Visit the Resource Manager UI again and note the application name listed is the one specified in the command line.
3. *Optional:* From the RM application list, follow the ApplicationMaster link (if the application is still running) or the History link to visit the Spark Application UI. View the **Environment** tab. Take note of the `spark.*` properties such as `master`, `appName`, and `driver` properties.

Setting Configuration Options in a Properties File

4. Using a text editor, create a file in the current working directory called `myspark.conf`, containing settings for the properties shown below:

```
spark.app.name      My Spark App
spark.master        yarn-client
spark.executor.memory 400M
```

5. Re-run your application, this time using the properties file instead of using the script options to configure Spark properties:

```
$ spark-submit --properties-file myspark.conf \
  CountJPGs.py /loudacre/weblogs/*
```

```
$ spark-submit --properties-file myspark.conf \
  --class stubs.CountJPGs \
  target/countjpgs-1.0.jar /loudacre/weblogs/*
```

- While the application is running, view the YARN UI and confirm that the Spark application name is correctly displayed as “My Spark App.”

ID	User	Name	Application Type	Queue	StartTime
application_1433857140912_0001	training	My Spark App	SPARK	root.training	Wed Jun 10 08:35:13 -0700 2015

Setting Logging Levels

- Copy the template file
`/usr/lib/spark/conf/log4j.properties.template` to
`/usr/lib/spark/conf/log4j.properties`. You will need to use
superuser privileges to do this, so use the `sudo` command:

```
$ sudo cp \
  /usr/lib/spark/conf/log4j.properties.template \
  /usr/lib/spark/conf/log4j.properties
```

- Load the new `log4j.properties` file into an editor. Again, you will need to use `sudo`. To edit the file with `gedit`, for instance, do this:

```
$ sudo gedit /usr/lib/spark/conf/log4j.properties
```

gedit Warnings

While using gedit with `sudo`, you may see `Gtk-WARNING` messages indicating permission issues or non-existent files. These can be disregarded.

9. The first line currently reads:

```
log4j.rootCategory=INFO, console
```

Replace `INFO` with `DEBUG`:

```
log4j.rootCategory=DEBUG, console
```

10. Save the file, and then close the editor.
11. Rerun your Spark application. Notice that the output now contains both `INFO` and `DEBUG` messages, like this:

```
16/03/19 11:40:45 INFO MemoryStore: ensureFreeSpace(154293) called
with curMem=0, maxMem=311387750
16/03/19 11:40:45 INFO MemoryStore: Block broadcast_0 stored as
values to memory (estimated size 150.7 KB, free 296.8 MB)
16/03/19 11:40:45 DEBUG BlockManager: Put block broadcast_0 locally
took 79 ms
16/03/19 11:40:45 DEBUG BlockManager: Put for block broadcast_0
without replication took 79 ms
```

Debug logging can be useful when debugging, testing, or optimizing your code, but in most cases it generates unnecessarily distracting output.

12. Edit the `log4j.properties` file again to replace `DEBUG` with `WARN` and try again. This time notice that no `INFO` or `DEBUG` messages are displayed, only `WARN` messages.

Note: Throughout the rest of the exercises, you may change these settings depending on whether you find the extra logging messages helpful or distracting. You can also override the current setting temporarily by calling `sc.setLogLevel` with your preferred setting. For example, in either Scala or Python, call:

```
> sc.setLogLevel("INFO")
```

This is the end of the exercise

Hands-On Exercise: View Jobs and Stages in the Spark Application UI

Files and Data Used in This Exercise:

Exercise directory: `$DEVSH/exercises/spark-stages`

Data files (HDFS): `/loudacre/weblogs/*`
 `/loudacre/accounts/*`

In this exercise, you will use the Spark Application UI to view the execution stages for a job.

In a previous exercise, you wrote a script in the Spark shell to join data from the accounts dataset with the weblogs dataset, in order to determine the total number of web hits for every account. Now you will explore the stages and tasks involved in that job.

Important: Important: If you did not run the course catch-up script for “Use Pair RDDs to Join Two Datasets” or an exercise after that one, do so now and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Exploring Partitioning of File-Based RDDs

1. Start (or restart, if necessary) the Spark shell. Although you would typically run a Spark application on a cluster, your course VM cluster has only a single worker node that can support only a single executor. To simulate a more realistic multi-node cluster, run in local mode with three threads:

```
$ pyspark --master 'local[3]'
```

```
$ spark-shell --master 'local[3]'
```

2. Review the accounts dataset (/loudacre/accounts/) using Hue or the command line. Take note of the number of files.
3. Create an RDD based on a *single file* in the dataset, such as /loudacre/accounts/part-m-00000, and then call `toDebugString` on the RDD, which displays the number of partitions in parentheses () before the RDD file and ID. How many partitions are in the resulting RDD?

```
pyspark> accounts=sc. \
    textFile("/loudacre/accounts/part-m-00000")
pyspark> print accounts.toDebugString()
```

```
scala> var accounts=sc.
    textFile("/loudacre/accounts/part-m-00000")
scala> accounts.toDebugString
```

4. Repeat this process, but specify a minimum of three of partitions: `sc.textFile(filename, 3)`. Does the RDD correctly have three partitions?
5. Finally, set the `accounts` variable to a new RDD based on *all the files* in the accounts dataset. How does the number of files in the dataset compare to the number of partitions in the RDD?
6. *Optional:* Use `foreachPartition` to print the first record of each partition.

Setting up the Job

7. Create an RDD of accounts, keyed by ID and with the string `first_name, last_name` for the value:

```
pyspark> accountsByID = accounts \
    .map(lambda s: s.split(',')) \
    .map(lambda values: \
        (values[0],values[4] + ',' + values[3]))
```

```
scala> val accountsByID = accounts.
    map(line => line.split(',')).
    map(values => (values(0),values(4)+',' +values(3)))
```

8. Construct a `userReqs` RDD with the total number of web hits for each user ID:

Tip: In this exercise, you will be reducing and joining large datasets, which can take a lot of time running on a single machine, as you are using in the course. Therefore, rather than use all the web log files in the dataset, specify a subset of web log files using a wildcard; for example, select only filenames ending in 2 by specifying `textFile("/loudacre/weblogs/*2.log")`.

```
pyspark> userReqs = sc \
    .textFile("/loudacre/weblogs/*2.log") \
    .map(lambda line: line.split()) \
    .map(lambda words: (words[2],1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```

```
scala> val userReqs = sc.
    textFile("/loudacre/weblogs/*2.log").
    map(line => line.split(' ')).
    map(words => (words(2),1)).
    reduceByKey((v1,v2) => v1 + v2)
```

9. Then join the two RDDs by user ID, and construct a new RDD with first name, last name, and total hits:

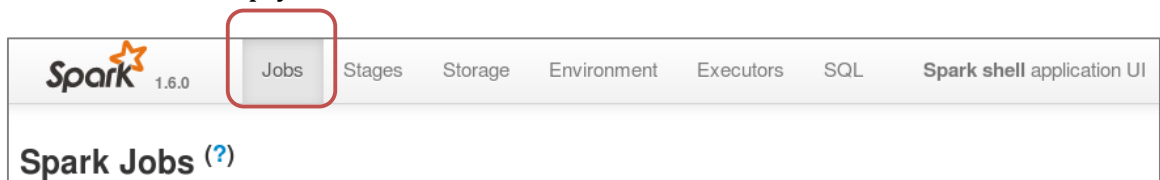
```
pyspark> accountHits = accountsByID.join(userReqs)\
    .values()
```

```
scala> val accountHits =
    accountsByID.join(userReqs).map(pair => pair._2)
```

10. Print the results of `accountHits.toDebugString` and review the output. Based on this, see if you can determine
- How many stages are in this job?
 - Which stages are dependent on which?
 - How many tasks will each stage consist of?

Running the Job and Reviewing the Job in the Spark Application UI

11. In your browser, visit the Spark Application UI by using the provided toolbar bookmark, or visiting URL `http://localhost:4040/`.
12. In the Spark UI, make sure the **Jobs** tab is selected. No jobs are yet running so the list will be empty.



13. Return to the shell and start the job by executing an action (`saveAsTextFile`):

```
pyspark> accountHits.\
    saveAsTextFile("/loudacre/userreqs")
```

```
scala> accountHits.  
      saveAsTextFile("/loudacre/userreqs")
```

14. Reload the Spark UI Jobs page in your browser. Your job will appear in the Active Jobs list until it completes, and then it will display in the Completed Jobs List.
15. Click the job description (which is the last action in the job) to see the stages. As the job progresses you may want to refresh the page a few times.

Things to note:

- a. How many stages are in the job? Does it match the number you expected from the RDD's `toDebugString` output?
 - b. The stages are numbered, but the numbers do not relate to the order of execution. Note the times the stages were submitted to determine the order. Does the order match what you expected based on RDD dependency?
 - c. How many tasks are in each stage?
 - d. The Shuffle Read and Shuffle Write columns indicate how much data was copied between tasks. This is useful to know because copying too much data across the network can cause performance issues.
16. Click the stages to view details about that stage. Things to note:
 - a. The Summary Metrics area shows you how much time was spend on various steps. This can help you narrow down performance problems.
 - b. The Tasks area lists each task. The Locality Level column indicates whether the process ran on the same node where the partition was physically stored or not. Remember that Spark will attempt to always run tasks where the data is, but may not always be able to, if the node is busy.

- c. In a real-world cluster, the executor column in the Task area would display the different worker nodes that ran the tasks. (In this single-node cluster, all tasks run on the same host: `localhost`.)
17. When the job is complete, return to the **Jobs** tab to see the final statistics for the number of tasks executed and the time the job took.
18. *Optional:* Try re-running the last action. (You will need to either delete the `saveAsTextFile` output directory in HDFS, or specify a different directory name.) You will probably find that the job completes much faster, and that several stages (and the tasks in them) show as “skipped.”

Bonus question: Which tasks were skipped and why?

This is the end of the exercise

Hands-On Exercise: Persist an RDD

Files and Data Used in This Exercise:

Exercise directory: `$DEVSH/exercises/spark-persist`

Data files (HDFS): `/loudacre/weblogs/*`
`/loudacre/accounts/*`

Stubs: `SparkPersist.pyspark`
`SparkPersist.scalaspark`

In this exercise, you will practice how to persist RDDs.

Important: If you did not run the course catch-up script for “Use Pair RDDs to Join Two Datasets” or an exercise after that one, do so now and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

1. Copy the code in the `SparkPersist` stub script in the exercise directory (`$DEVSH/exercises/spark-persist`) into the Spark shell. (A stub is provided for Scala and Python; use whichever language you prefer.)
2. The stub code is very similar to the job setup code in the “View Jobs and Stages in the Spark Application UI” exercise. It sets up an RDD called `accountHits` that joins account data with web log data. However, this time you will start the job by performing a slightly different action than in that exercise: count the number of user accounts with a total hit count greater than five. Enter the code below into the shell:

```
pyspark> accountHits\  
  .filter(lambda (firstlast,hitcount): hitcount > 5)\  
  .count()
```

```
scala> accountHits.filter(pair => pair._2 > 5).count()
```

3. Persist the RDD to memory by calling `accountHits.persist()`.
4. In your browser, view the Spark Application UI and select the **Storage** tab. At this point, you have marked your RDD to be persisted, but you have not yet performed an action that would cause it to be materialized and persisted, so you will not yet see any persisted RDDs.
5. In the Spark shell, execute the count again.
6. View the RDD's `toDebugString`. Notice that the output indicates the persistence level selected.
7. Reload the **Storage** tab in your browser, and this time note that the RDD you persisted is shown. Click the RDD Name to see details about partitions and persistence.
8. Click the **Executors** tab and take note of the amount of memory used and available for your one worker node.

Note that the classroom environment has a single worker node with a small amount of memory allocated, so you may see that not all of the dataset is actually cached in memory. In the real world, for good performance a cluster will have more nodes, each with more memory, so that more of your active data can be cached.

9. *Optional:* Set the RDD's persistence level to `DISK_ONLY` and compare the storage report in the Spark Application Web UI.
 - Hint: Set the RDD's persistence level to `StorageLevel.DISK_ONLY`. You will need to import the class first or use the fully qualified name of the class `StorageLevel` when invoking `persist()`.
 - Hint: Because you have already persisted the RDD at a different level, you will need to `unpersist()` first before you can set a new level.

This is the end of the exercise

Hands-On Exercise: Implement an Iterative Algorithm with Apache Spark

Files and Data Used in This Exercise:

Exercise directory: `$DEVSH/exercises/spark-iterative`

Data files (HDFS): `/loudacre/devicestatus_etl/*`

Stubs: `KMeansCoords.pyspark`
 `KMeansCoords.scalaspark`

In this exercise, you will practice implementing iterative algorithms in Spark by calculating k-means for a set of points.

Reviewing the Data

1. If you completed the bonus section of the “Process Data Files with Apache Spark” exercise, you used Spark to extract the date, maker, device ID, latitude and longitude from the `devicestatus.txt` data file, and store the results in the HDFS directory `/loudacre/devicestatus_etl`.

If you did not complete that bonus exercise, upload the solution file from the local filesystem to HDFS now. (If you have run the course catch-up script, this is already done for you.)

```
$ hdfs dfs -put $DEVDATA/static_data/devicestatus_etl \
  /loudacre/
```

2. Examine the data in the dataset. Note that the latitude and longitude are the 4th and 5th fields, respectively, as shown in the sample data below:

```
2014-03-15:10:10:20,Sorrento,8cc3b47e-bd01-4482-b500-
28f2342679af,33.6894754264,-117.543308253
2014-03-15:10:10:20,MeeToo,ef8c7564-0a1a-4650-a655-
c8bbd5f8f943,37.4321088904,-121.485029632
```

Calculating k-means for Device Location

3. Start with the provided `KMeansCoords` stub file, which contains the following convenience functions used in calculating k-means:
 - `closestPoint`: given a (latitude/longitude) point and an array of current center points, returns the index in the array of the center closest to the given point
 - `addPoints`: given two points, return a point which is the sum of the two points—that is, $(x1+x2, y1+y2)$
 - `distanceSquared`: given two points, returns the squared distance of the two—this is a common calculation required in graph analysis

Note that the stub code sets the variable `K` equal to 5—this is the number of means to calculate.

4. The stub code also sets the variable `convergeDist`. This will be used to decide when the k-means calculation is done—when the amount the locations of the means changes between iterations is less than `convergeDist`. A “perfect” solution would be 0; this number represents a “good enough” solution. For this exercise, use a value of 0.1.
5. Parse the input file—which is delimited by commas—into (latitude, longitude) pairs (the 4th and 5th fields in each line). Only include known locations—that is, filter out (0, 0) locations. Be sure to persist the resulting RDD because you will access it each time through the iteration.
6. Create a K-length array called `kPoints` by taking a random sample of K location points from the RDD as starting means (center points). For example, in Python:

```
data.takeSample(False, K, 42)
```

Or in Scala:

```
data.takeSample(false, K, 42)
```

7. Iteratively calculate a new set of K means until the total distance between the means calculated for this iteration and the last is smaller than `convergeDist`.

For each iteration:

- a. For each coordinate point, use the provided `closestPoint` function to map that point to the index in the `kPoints` array of the location closest to that point. The resulting RDD should be keyed by the index, and the value should be the pair: `(point, 1)`. (The value 1 will later be used to count the number of points closest to a given mean.) For example:

(1, ((37.43210, -121.48502), 1))
(4, ((33.11310, -111.33201), 1))
(0, ((39.36351, -119.40003), 1))
(1, ((40.00019, -116.44829), 1))
...

- b. Reduce the result: for each center in the `kPoints` array, sum the latitudes and longitudes, respectively, of all the points closest to that center, and also find the number of closest points. For example:

(0, ((2638919.87653, -8895032.182481), 74693))
(1, ((3654635.24961, -12197518.55688), 101268))
(2, ((1863384.99784, -5839621.052003), 48620))
(3, ((4887181.82600, -14674125.94873), 126114))
(4, ((2866039.85637, -9608816.13682), 81162))

- c. The reduced RDD should have (at most) K members. Map each to a new center point by calculating the average latitude and longitude for each set of closest points: that is, map $(\text{index}, (\text{totalX}, \text{totalY}), n)$ to $(\text{index}, (\text{totalX}/n, \text{totalY}/n))$.
 - d. Collect these new points into a local map or array keyed by index.
 - e. Use the provided `distanceSquared` method to calculate how much the centers “moved” between the current iteration and the last. That is, for each center in `kPoints`, calculate the distance between that point and the corresponding new point, and sum those distances. That is the delta between iterations; when the delta is less than `convergeDist`, stop iterating.
 - f. Copy the new center points to the `kPoints` array in preparation for the next iteration.
8. When all iterations are complete, display the final K center points.

This is the end of the exercise

Hands-On Exercise: Use Apache Spark SQL for ETL

Files and Data Used in This Exercise

Exercise directory: \$DEVSH/exercises/spark-sql

MySQL database: loudacre

MySQL table: webpage

Output path (HDFS): /loudacre/webpage_files

In this exercise, you will use Spark SQL to load structured data from a Parquet file, process it, and store it to a new file.

The data you will work with in this exercise is from the `webpage` table in the `loudacre` database in MySQL. Although Spark SQL does allow you to directly access tables in a database using JDBC, doing so is not generally a best practice, because in a distributed environment it may lead to an unintentional Denial of Service attack on the database. So in this exercise, you will use Sqoop to import the data to HDFS first. You will use Parquet file format rather than a text file because this preserves the data's schema for use by Spark SQL.

Importing Data from MySQL Using Sqoop

1. In a terminal window, use Sqoop to import the `webpage` table from MySQL. Use Parquet file format.

```
$ sqoop import \  
  --connect jdbc:mysql://localhost/loudacre \  
  --username training --password training \  
  --table webpage \  
  --target-dir /loudacre/webpage \  
  --as-parquetfile
```

2. Using Hue or the `hdfs` command line utility, list the data files imported to the `/loudacre/webpage` directory.
3. *Optional:* Download one of the generated Parquet files from HDFS to a local directory. Use `parquet-tools head` and `parquet-tools schema` to review the schema and some sample data. Take note of the structure of the data; you will use this data in the next exercise sections.

Creating a DataFrame from a Table

4. If necessary, start the Spark shell.
5. The Spark shell predefines a SQL context object as `sqlContext`. What type is the SQL context? In either Python or Scala, view the `sqlContext` object:

```
> sqlContext
```

6. Create a DataFrame based on the `webpage` table:

```
pyspark> webpageDF = sqlContext \
    .read.load("/loudacre/webpage")
```

```
scala> val webpageDF = sqlContext.
    read.load("/loudacre/webpage")
```

7. Examine the schema of the new DataFrame by calling `webpageDF.printSchema()`.
8. View the first few records in the table by calling `webpageDF.show(5)`.

Note that the data in the `associated_files` column is a comma-delimited string. Loudacre would like to make this data available in an Impala table, but in order to perform required analysis, the `associated_files` data must be extracted and normalized. Your goal in the next section is to use the DataFrames API to extract the data in the column, split the string, and create a

new data file in HDFS containing each page ID, and its associated files in separate rows.

Querying a DataFrame

9. Create a new DataFrame by selecting the `web_page_num` and `associated_files` columns from the existing DataFrame:

```
python> assocFilesDF = \
    webpageDF.select(webpageDF.web_page_num, \
    webpageDF.associated_files)
```

```
scala> val assocFilesDF =
    webpageDF.select($"web_page_num", $"associated_files")
```

10. View the schema and the first few rows of the returned DataFrame to confirm that it was created correctly.
11. In order to manipulate the data using core Spark, convert the DataFrame into a pair RDD using the `map` method. The input into the `map` method is a `Row` object. The key is the `web_page_num` value, and the value is the `associated_files` string.

In Python, you can dynamically reference the column value of the `Row` by name:

```
pyspark> aFilesRDD = assocFilesDF.map(lambda row: \
    (row.web_page_num, row.associated_files))
```

In Scala, use the correct `get` method for the type of value with the column index:

```
scala> val aFilesRDD = assocFilesDF.
    map(row => (row.getAs[Short]("web_page_num"),
    row.getAs[String]("associated_files")))
```


12. Now that you have an RDD, you can use the familiar `flatMapValues` transformation to split and extract the filenames in the `associated_files` column:

```
pyspark> aFilesRDD2 = aFilesRDD \
    .flatMapValues( \
        lambda filestring:filestring.split(',') )
```

```
scala> val aFilesRDD2 =
    aFilesRDD.flatMapValues(filestring =>
        filestring.split(','))
```

13. Import the `Row` class and convert the pair RDD to a `Row` RDD. (Note: this step is only necessary in Scala.)

```
scala> import org.apache.spark.sql.Row
scala> val aFilesRowRDD = aFilesRDD2.map(pair =>
    Row(pair._1,pair._2))
```

14. Convert the RDD back to a `DataFrame`, using the original `DataFrame`'s schema:

```
pyspark> aFileDF = sqlContext. \
    createDataFrame(aFilesRDD2,assocFilesDF.schema)
```

```
scala> val aFileDF = sqlContext.
    createDataFrame(aFilesRowRDD,assocFilesDF.schema)
```

15. Call `printSchema` on the new `DataFrame`. Note that Spark SQL gave the columns the same names they had originally: `web_page_num` and `associated_files`. The second column name is no longer accurate, because the data in the column reflects only a *single* associated file.

- 16.** Create a new DataFrame with the `associated_files` column renamed to `associated_file`:

```
pyspark> finalDF = aFileDF. \
    withColumnRenamed('associated_files', \
        'associated_file')
```

```
scala> val finalDF = aFileDF.
    withColumnRenamed("associated_files",
        "associated_file")
```

- 17.** Call `finalDF.printSchema()` to confirm that the new DataFrame has the correct column names.
- 18.** Call `show(5)` on the new DataFrame to confirm that the final data is correct.
- 19.** Your final DataFrame contains the processed data, so save it in Parquet format (the default) in directory `/loudacre/webpage_files`.

```
pyspark> finalDF.write. \
    mode("overwrite"). \
    save("/loudacre/webpage_files")
```

```
scala> finalDF.write.
    mode("overwrite").
    save("/loudacre/webpage_files")
```

- 20.** Using Hue or the HDFS command line tool, list the Parquet files that were saved by Spark SQL.
- 21. Optional:** Use `parquet-tools schema` and `parquet-tools head` to review the schema and some sample data of the generated files.

22. *Optional:* In the Spark web UI, try viewing the **SQL** tab. How many queries were completed as part of this exercise? How many jobs?

This is the end of the exercise

Hands-On Exercise: Produce and Consume Apache Kafka Messages

Files and Data Used in This Exercise

Exercise directory: \$DEVSH/exercises/kafka

In this exercise, you will use Kafka's command line tool to create a Kafka topic. You will also use the command line producer and consumer clients to publish and read messages.

Creating a Kafka Topic

1. Open a new terminal window and create a Kafka topic named `weblogs` that will contain messages representing lines in Loudacre's web server logs. Since your exercise environment is a single-node cluster running on a virtual machine, use a replication factor of 1 and a single partition.

```
$ kafka-topics --create \  
  --zookeeper localhost:2181 \  
  --replication-factor 1 \  
  --partitions 1 \  
  --topic weblogs
```

You will see the message: Created topic "weblogs".

2. Display all Kafka topics to confirm that the new topic you just created is listed:

```
$ kafka-topics --list \  
  --zookeeper localhost:2181
```

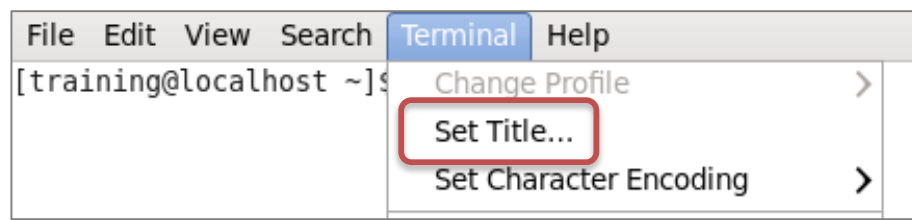
Producing and Consuming Messages

You will now use Kafka command line utilities to start producers and consumers for the topic created earlier.

3. Start a Kafka producer for the `weblogs` topic:

```
$ kafka-console-producer \  
  --broker-list localhost:9092 \  
  --topic weblogs
```

Tip: This exercise involves using multiple terminal windows. To avoid confusion, set a different title for each one by selecting **Set Title...** on the **Terminal** menu:



Set the title for this window to “Kafka Producer.”

4. Publish a test message to the `weblogs` topic by typing the message text and then pressing `Enter`. For example:

```
test weblog entry 1
```

5. Open a new terminal window and adjust it to fit on the window beneath the producer window. Set the title for this window to “Kafka Consumer.”
6. In the new terminal window, start a Kafka consumer that will read from the beginning of the `weblogs` topic:

```
$ kafka-console-consumer \  
  --zookeeper localhost:2181 \  
  --topic weblogs \  
  --from-beginning
```

You should see the status message you sent using the producer displayed on the consumer's console, such as:

```
test weblog entry 1
```

7. Press `Ctrl+C` to stop the weblogs consumer, and restart it, but this time omit the `--from-beginning` option to this command. You should see that no messages are displayed.
8. Switch back to the producer window and type another test message into the terminal, followed by the `Enter` key:

```
test weblog entry 2
```

9. Return to the consumer window and verify that it now displays the alert message you published from the producer in the previous step.

Cleaning Up

10. Press `Ctrl+C` in the consumer terminal window to end its process.
11. Press `Ctrl+C` in the producer terminal window to end its process.

This is the end of the exercise

Hands-On Exercise: Collect Web Server Logs with Apache Flume

Files and Data Used in This Exercise

Exercise directory: \$DEVSH/exercises/flume

Data files (local): \$DEVDATA/weblogs/*

In this exercise, you will run a Flume agent to ingest web log data from a local directory to HDFS.

Apache web server logs are generally stored in files on the local machines running the server. In this exercise, you will simulate an Apache server by placing provided web log files into a local spool directory, and then use Flume to collect the data.

Both the local and HDFS directories must exist before using the spooling directory source.

Creating an HDFS Directory for Flume-Ingested Data

1. Create a directory in HDFS called `/loudacre/weblogs_flume` to hold the data files Flume ingests:

```
$ hdfs dfs -mkdir -p /loudacre/weblogs_flume
```

Creating a Local Directory for Web Server Log Output

2. Create the spool directory into which the web log simulator will store data files for Flume to ingest. On the local Linux filesystem create `/flume/weblogs_spooldir`:

```
$ sudo mkdir -p /flume/weblogs_spooldir
```

3. Give all users the permissions to write to the `/flume/weblogs_spooldir` directory:

```
$ sudo chmod a+w -R /flume
```

Configuring Flume

A Flume agent configuration file has been provided for you:

`$DEVSH/exercises/flume/spooldir.conf`.

Review the configuration file. *You do not need to edit this file.* Take note in particular of the following:

- The *source* is a spooling directory source that pulls from the local `/flume/weblogs_spooldir` directory.
- The *sink* is an HDFS sink that writes files to the HDFS `/loudacre/weblogs_flume` directory.
- The *channel* is a memory channel.

Running the Flume Agent

Next, start the Flume agent and copy the files to the spooling directory.

4. Change directories to the exercise directory.

```
$ cd $DEVSH/exercises/flume
```

5. Start the Flume agent using the configuration you just reviewed:

```
$ flume-ng agent --conf /etc/flume-ng/conf \
  --conf-file spooldir.conf \
  --name agent1 -Dflume.root.logger=INFO,console
```


6. Wait a few moments for the Flume agent to start up. You will see a message like:
Component type: SOURCE, name: webserver-log-source
started

Simulating Apache Web Server Output

7. Open a separate terminal window. Run the script to place the web log files in the `/flume/weblogs_spooldir` directory:

```
$ $DEVSH/exercises/flume/copy-move-weblogs.sh \  
/flume/weblogs_spooldir
```

This script will create a temporary copy of the web log files and move them to the `spooldir` directory.

8. Return to the terminal that is running the Flume agent and watch the logging output. The output will give information about the files Flume is putting into HDFS.
9. Once the Flume agent has finished, enter `Ctrl+C` to terminate the process.
10. Using the command line or Hue File Browser, list the files that were added by the Flume agent in the HDFS directory `/loudacre/weblogs_flume`.

Note that the files that were imported are tagged with a Unix timestamp corresponding to the time the file was imported, such as `FlumeData.1427214989392`.

This is the end of the exercise

Hands-On Exercise: Write an Apache Spark Streaming Application

Files and Directories Used in This Exercise:

Exercise directory: `$DEVSH/exercises/spark-streaming`

Python stub: `stubs-python/StreamingLogs.py`

Python solution: `solution-python/StreamingLogs.py`

Scala project:

Project directory: `streaminglogs_project`

Stub class: `stubs.StreamingLogs`

Solution class: `solution.StreamingLogs`

Test data (local): `$DEVDATA/weblogs/*`

Test script: `streamtest.py`

In this exercise, you will write a Spark Streaming application to count Knowledge Base article requests.

This exercise has two parts. First, you will review the Spark Streaming documentation. Then you will write and test a Spark Streaming application to read streaming web server log data and count the number of requests for Knowledge Base articles.

Reviewing the Spark Streaming Documentation

1. View the Spark Streaming API by opening the Spark API documentation for either Scala or Python and then:

For Scala:

- Scroll down and select the `org.apache.spark.streaming` package in the package pane on the left.

- Follow the links at the top of the package page to view the `DStream` and `PairDStreamFunctions` classes— these will show you the methods available on a `DStream` of regular RDDs and pair RDDs respectively.

For Python:

- Go to the `pyspark.streaming` module.
 - Scroll down to the `pyspark.streaming.DStream` class and review the available methods.
2. You may also wish to view the *Spark Streaming Programming Guide* (select **Programming Guides > Spark Streaming** on the Spark documentation main page).

Simulating Streaming Web Logs

To simulate a streaming data source, you will use the provided `streamtest.py` Python script, which waits for a connection on the host and port specified and, once it receives a connection, sends the contents of the file(s) specified to the client (which will be your Spark Streaming application). You can specify the speed (in lines per second) at which the data should be sent.

3. Change to the exercise directory.

```
$ cd $DEVSH/exercises/spark-streaming
```

4. Stream the Loudacre web log files at a rate of 20 lines per second using the provided test script.

```
$ python streamtest.py localhost 1234 20 \
  $DEVDATA/weblogs/*
```

This script will exit after the client disconnects, so you will need to restart the script whenever you restart your Spark application.

Writing a Spark Streaming Application

5. To help you get started writing a Spark Streaming application, stub files have been provided for you.

For Python, start with the stub file `StreamingLogs.py` in the `$DEVSH/exercises/spark-streaming/stubs-python` directory, which imports the necessary classes for the application.

For Scala, a Maven project directory called `streaminglogs_project` has been provided in the exercise directory (`$DEVSH/exercises/spark-streaming`). To complete the exercise, start with the stub code in `src/main/scala/stubs/StreamingLogs.scala`, which imports the necessary classes for the application.

6. Define a Streaming context with a one-second batch duration.
7. Create a DStream by reading the data from the host and port provided as input parameters.
8. Filter the DStream to only include lines containing the string `KBDOC`.
9. To confirm that your application is correctly receiving the streaming web log data, display the first five records in the filtered DStream for each one-second batch. (In Scala, use the DStream `print` function; in Python, use `pprint`.)
10. For each RDD in the filtered DStream, display the number of items—that is, the number of requests for KB articles.

Tip: Python does not allow calling `print` within a lambda function, so create a named defined function to print.

11. Save the filtered logs to text files in HDFS. Use the base directory name `/loudacre/streamlog/kblogs`.
12. Finally, start the Streaming context, and then call `awaitTermination()`.

Testing the Application

13. In a new terminal window, change to the correct directory for the language you are using for your application.

For Python, change to the exercise directory:

```
$ cd $DEVSH/exercises/spark-streaming
```

For Scala, change to the project directory for the exercise:

```
$ cd \
$DEVSH/exercises/spark-streaming/streaminglogs_project
```

14. If you are using Scala, build your application JAR file using the `mvn package` command.
15. Use `spark-submit` to run your application locally and be sure to specify two threads; at least two threads or nodes are required to run a streaming application, while the VM cluster has only one. The `StreamingLogs` application takes two parameters: the host name and the port number to connect the DStream to. Specify the same host and port at which the test script you started earlier is listening.

```
$ spark-submit --master 'local[2]' \
  stubs-python/StreamingLogs.py localhost 1234
```

Note: Use `solution-python/StreamingLogs.py` to run the solution application instead.

```
$ spark-submit --master 'local[2]' \
  --class stubs.StreamingLogs \
  target/streamlog-1.0.jar localhost 1234
```

Note: Use `--class solution.StreamingLogs` to run the solution class instead.

16. After a few moments, the application should connect to the test script's simulated stream of web server log output. Confirm that for every batch of data received (every second), the application displays the first few Knowledge Base requests and the count of requests in the batch. Review the HDFS files the application saved in `/loudacre/streamlog`.
17. Return to the terminal window in which you started the `streamtest.py` test script earlier. Stop the test script by typing `Ctrl+C`. You do not need to wait until all the web log data has been sent.

Warning: Stopping Your Application

You *must* stop the test script before stopping your Spark Streaming application. If you attempt to stop the application while the test script is still running, you may find that the application appears to hang while it takes several minutes to complete. (It will make repeated attempts to reconnect with the data source, which the test script does not support.)

18. After the test script has stopped, stop your application by typing `Ctrl+C` in the terminal window the application is running in.

This is the end of the exercise

Hands-On Exercise: Process Multiple Batches with Apache Spark Streaming

Files and Data Used in This Exercise

Exercise directory: `$DEVSH/exercises/spark-streaming-multi`

Python stub: `stubs-python/StreamingLogsMB.py`

Python solution: `solution-python/StreamingLogsMB.py`

Scala project:

Project directory: `streaminglogsMB_project`

Stub class: `stubs.StreamingLogsMB`

Solution class: `solution.StreamingLogsMB`

Data (local): `$DEVDATA/weblogs/*`

In this exercise, you will write a Spark Streaming application to count web page requests over time.

Simulating Streaming Web Logs

To simulate a streaming data source, you will use the provided `streamtest.py` Python script, which waits for a connection on the host and port specified and, once it receives a connection, sends the contents of the file(s) specified to the client (which will be your Spark Streaming application). You can specify the speed (in lines per second) at which the data should be sent.

1. Change to the exercise directory.

```
$ cd $DEVSH/exercises/spark-streaming-multi
```

2. Stream the Loudacre Web log files at a rate of 20 lines per second using the provided test script.

```
$ python streamtest.py localhost 1234 20 \
  $DEVDATA/weblogs/*
```

This script exits after the client disconnects, so you will need to restart the script when you restart your Spark application.

Displaying the Total Request Count

3. A stub file for this exercise has been provided for you in the exercise directory. The stub code creates a Streaming context for you, and creates a DStream called `logs` based on web log request messages received on a network socket.

For Python, start with the stub file `StreamingLogsMB.py` in the `stubs-python` directory.

For Scala, a Maven project directory called `streaminglogsMB_project` has been provided in the exercise directory. To complete the exercise, start with the stub code in `src/main/scala/stubs/StreamingLogsMB.scala`.

4. Enable checkpointing to a directory called `logcheckpt`.
5. Count the number of page requests over a window of five seconds. Print out the updated five-second total every two seconds.

- Hint: Use the `countByWindow` function.

Building and Running Your Application

6. In a different terminal window than the one in which you started the `streamtest.py` script, change to the correct directory for the language you are using for your application.

For Python, change to the exercise directory:

```
$ cd $DEVSH/exercises/spark-streaming-multi
```

For Scala, change to the project directory for the exercise:

```
$ cd \
$DEVSH/exercises/spark-streaming-multi/streaminglogsMB_project
```

7. If you are using Scala, build your application JAR file using the `mvn package` command.
8. Use `spark-submit` to run your application locally and be sure to specify two threads; at least two threads or nodes are required to running a streaming application, while the VM cluster has only one. Your application takes two parameters: the host name and the port number to connect the DStream to. Specify the same host and port at which the test script you started earlier is listening.

```
$ spark-submit --master 'local[2]' \
  stubs-python/StreamingLogsMB.py localhost 1234
```

- **Note:** Use `solution-python/StreamingLogsMB.py` to run the solution application instead.

```
$ spark-submit --master 'local[2]' \
  --class stubs.StreamingLogsMB \
  target/streamlogmb-1.0.jar localhost 1234
```

- **Note:** Use `--class solution.StreamingLogsMB` to run the solution class instead.
9. After a few moments, the application should connect to the test script's simulated stream of web server log output. Confirm that for every batch of data received (every second), the application displays the first few Knowledge Base requests and the count of requests in the batch. Review the files.

10. Return to the terminal window in which you started the `streamtest.py` test script earlier. Stop the test script by typing `Ctrl+C`. You do not need to wait until all the web log data has been sent.

Warning: Stopping Your Application

You *must* stop the test script before stopping your Spark Streaming application. If you attempt to stop the application while the test script is still running, you may find that the application appears to hang while it takes several minutes to complete. (It will make repeated attempts to reconnect with the data source, which the test script does not support.)

11. After the test script has stopped, stop your application by typing `Ctrl+C` in the terminal window the application is running in.

Bonus Exercise

Extend the application you wrote above to also count the total number of page requests by user from the start of the application, and then display the top ten users with the highest number of requests.

Follow the steps below to implement a solution for this bonus exercise:

1. Use map-reduce to count the number of times each user made a page request in each batch (a hit-count).
 - Hint: Remember that the User ID is the 3rd field in each line.
2. Define a function called `updateCount` that takes an array (in Python) or sequence (in Scala) of hit-counts and an existing hit-count for a user. The function should return the sum of the new hit-counts plus the existing count.
 - Hint: An example of an `updateCount` function is in the course material and the code can be found in `$DEVSH/examples/spark/spark-streaming`.
3. Use `updateStateByKey` with your `updateCount` function to create a new DStream of users and their hit-counts over time.

4. Use `transform` to call the `sortByKey` transformation to sort by hit-count.

- Hint: You will have to swap the key (user ID) with the value (hit-count) to sort.

Note: The solution files for this exercise include code for this bonus exercise.

This is the end of the exercise

Hands-On Exercise: Process Apache Kafka Messages with Apache Spark Streaming

Files and Data Used in This Exercise

Exercise directory: `$DEVSH/exercises/spark-streaming-kafka`

Python stub: `stubs-python/StreamingLogsKafka.py`

Python solution: `stubs-solution/StreamingLogsKafka.py`

Scala project:

Project directory: `streaminglogskafka_project`

Stub class: `stubs.StreamingLogsKafka`

Solution class: `solution.StreamingLogsKafka`

Data (local): `$DEVDATA/weblogs/*`

In this exercise, you will write an Apache Spark Streaming application to handle web logs received as messages on a Kafka topic.

In a prior exercise, you started a Flume agent that collects web log files from a local spool directory and passes them to a Kafka sink. In this exercise, you will use the same Flume agent to produce data and publish it to Kafka. The Kafka topic will be the data source for your Spark Streaming application.

IMPORTANT: For this custom course, you *must* run the following command to prepare for this exercise before continuing:

```
$ $DEVSH/scripts/catchup.sh
```

Consuming Messages from a Kafka Direct DStream

1. For Python, start with the stub file `StreamingLogsKafka.py` in the `stubs-python` directory, which imports the necessary classes for the application.

For Scala, a Maven project directory called `streaminglogskafka_project` has been provided in the exercise directory. To complete the exercise, start with the stub code in `src/main/scala/stubs/StreamingLogsKafka.scala`, which imports the necessary classes for the application.

2. Create a DStream using `KafkaUtils.createDirectStream`.
 - The broker list consists of a single broker: `localhost:9092`.
 - The topic list consists of a single topic: the argument to the main function passed in by the user when the application is submitted.

Refer to the course materials for the details of creating a Kafka stream.

3. Kafka messages are in (key, value) form, but for this application, the key is null and only the value is needed. (The value is the web log line.) Map the DStream to remove the key and use only the value.
4. To verify that the DStream is correctly receiving messages, display the first 10 elements in each batch.
5. For each RDD in the DStream, display the number of items—that is, the number of requests.

Tip: Python does not allow calling `print` within a lambda function, so define a named function to print.

6. Save the filtered logs to text files in HDFS. Use the base directory name `/loudacre/streamlog/kafkalog`.

Building and Running Your Application

7. Change to the correct directory for the language you are using for your application.

For Python, change to the exercise directory:

```
$ cd $DEVSH/exercises/spark-streaming-kafka
```

For Scala, change to the project directory for the exercise:

```
$ cd \
$DEVSH/exercises/spark-streaming-kafka/streaminglogskafka_project
```

8. If you are using Scala, you will need to build your application JAR file using the `mvn package` command.
9. Use `spark-submit` to run your application locally and be sure to specify two threads; at least two threads or nodes are required to run a streaming application, while the VM cluster has only one. Your application takes one parameter: the name of the Kafka topic from which the DStream will read messages, `weblogs`.

```
$ spark-submit --master 'local[2]' \
  stubs-python/StreamingLogsKafka.py weblogs
```

- **Note:** Use `solution-python/StreamingLogsKafka.py` to run the solution application instead.

```
$ spark-submit --master 'local[2]' \
  --class stubs.StreamingLogsKafka \
  target/streamlogkafka-1.0.jar weblogs
```

- **Note:** Use `--class solution.StreamingLogsKafka` to run the solution class instead.

Producing Messages for Spark Streaming

10. In a separate terminal window, start a Flume agent using the configuration file from the Flume/Kafka exercise:

```
$ flume-ng agent --conf /etc/flume-ng/conf \
--conf-file \
$DEVSH/exercises/flafka/spooldir_kafka.conf \
--name agent1 -Dflume.root.logger=INFO,console
```

11. Wait a few moments for the Flume agent to start up. You will see a message like:
Component type: SINK, name: kafka-sink started

12. In a separate new terminal window, run the script to place the web log files in the `/flume/weblogs_spooldir` directory.

If you completed the Flume exercises or ran `catchup.sh` previously, the script will prompt whether you want to clear out the `spooldir` directory. Be sure to enter `y` when prompted.

```
$ $DEVSH/exercises/flafka/copy-move-weblogs.sh \
/flume/weblogs_spooldir
```

- **Note:** You can rerun the `copy-move-weblogs.sh` script to send the web log data to Spark Streaming again if needed to test your application.

13. Return to the terminal window where your Spark application is running to verify the count output. Also review the contents of the saved files in HDFS directories `/loudacre/streamlog/kafkalogs-<time-stamp>`. These directories hold `part` files containing the page requests.

Cleaning Up

- 14.** Stop the Flume agent by pressing `Ctrl+C`. You do not need to wait until all the web log data has been sent.
- 15.** Stop the Spark application by pressing `Ctrl+C`. (You may see several error messages resulting from the interruption of the job in Spark; you may disregard these.)

This is the end of the exercise

Appendix A: Enabling Jupyter Notebook for PySpark

Jupyter (iPython Notebook) is installed on the VM for this course. To use it instead of the command-line version of PySpark, follow these steps:

1. Open the following file for editing: `/home/training/.bashrc`
2. Uncomment out the following line (remove the leading #).

```
# export PYSPARK_DRIVER_PYTHON_OPTS='notebook .....jax'
```

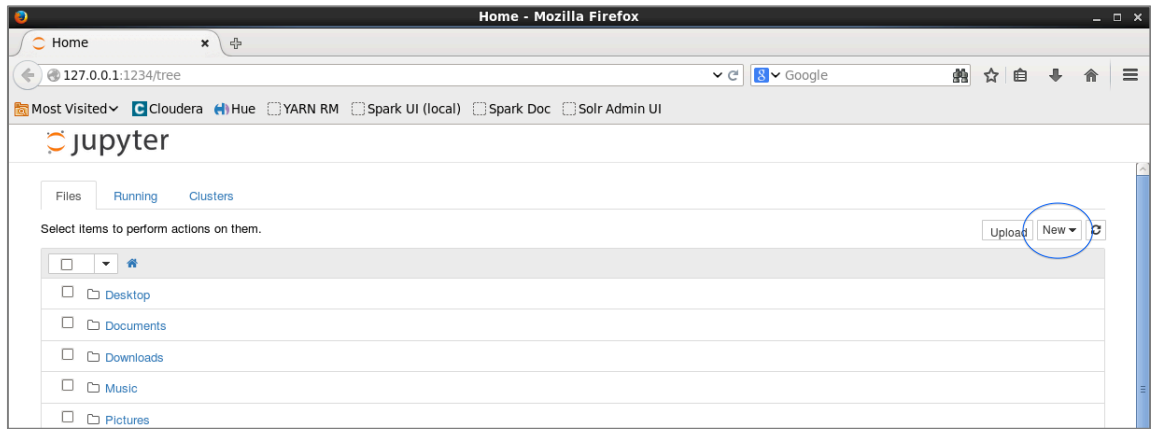
3. Save the file.
4. Open a new terminal window. (It must be a new terminal so it reloads your edited `.bashrc` file.)
5. Confirm the changes with the following Linux command:

```
$ env | grep PYSPARK
```

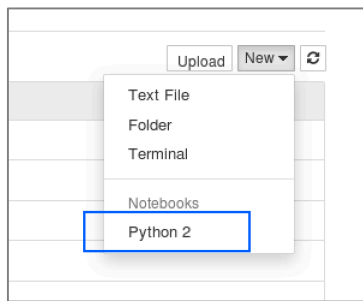
The output should be as follows. If you do not see this output, the `.bashrc` file was not edited or saved properly.

```
PYSPARK_DRIVER_PYTHON=ipython
PYSPARK_DRIVER_PYTHON_OPTS=notebook --ip 127.0.0.1
--port 3333 --no-mathjax
```

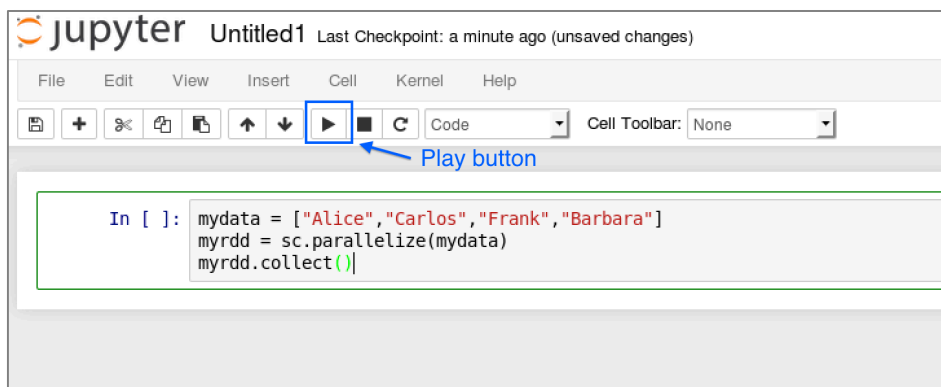
6. Enter `pyspark` in the terminal. This will cause a browser window to open, and you should see the following web page:



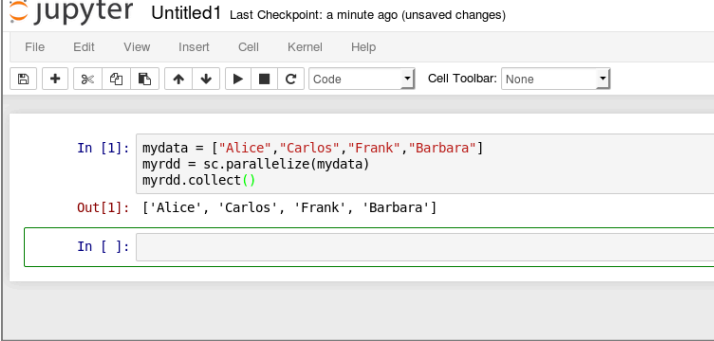
7. On the right hand side of the page select **Python 2** from the **New** menu.



8. Enter some Spark code such as the following and use the play button to execute your Spark code.



9. Notice the output displayed.



The image shows a Jupyter Notebook window titled "Untitled1" with a status bar indicating "Last Checkpoint: a minute ago (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar with icons for file operations, code execution, and cell management. The main area contains a code cell with the following Python code:

```
In [1]: mydata = ["Alice", "Carlos", "Frank", "Barbara"]
        myrdd = sc.parallelize(mydata)
        myrdd.collect()
```

Below the code cell, the output is displayed:

```
Out[1]: ['Alice', 'Carlos', 'Frank', 'Barbara']
```

At the bottom, there is an empty input cell labeled "In []:".

Appendix B: Managing Services on the Course Virtual Machine

On the course VM, Hadoop is installed in *pseudo-distributed mode*. This means all the Hadoop services (daemons) that would normally be running on different machines in a cluster are running on a single machine. This machine is playing the roles of client nodes, master nodes, and worker nodes when you perform the exercises.

There are many services that could be running on the VM at any given time, such as the NameNode, DataNode, YARN ResourceManager, and Hue.

You can discover what services are available on the VM by listing the content of the `/etc/init.d/` directory on the Linux filesystem.

If, for example, the NameNode daemon stopped working, you would need to know the name of the service in order to check its status, start it, stop it, and so on. This is the command to find the name of that service:

```
$ ls -cl /etc/init.d/*hdfs*
```

From the list that the `ls` command produces, you can see the specific name of the NameNode service: `hadoop-hdfs-namenode`.

To manage the service, you need superuser privileges. You could switch to the `root` user (password `training`) or use the `sudo` command ("superuser do").

Use the `service` command with `sudo` to manage the service.

To list the available options you can specify, use `sudo service` with a service name and no additional options:

```
$ sudo service hadoop-hdfs-namenode
```

To check the status of a service, use the `status` option:

```
$ sudo service hadoop-hdfs-namenode status
```

To start a service that is not running, use the `start` option:

```
$ sudo service hadoop-hdfs-namenode start
```

To restart a service that is already running, use the `restart` option:

```
$ sudo service hadoop-hdfs-namenode restart
```

Below is a list of services installed on the VM that are pertinent to this course:

General Apache Hadoop and YARN Services

- `hadoop-hdfs-datanode`
- `hadoop-hdfs-namenode`
- `hadoop-yarn-nodemanager`
- `hadoop-yarn-resourcemanager`

Hadoop MapReduce Services

- `hadoop-mapreduce-historyserver`

Apache Spark Services

- `spark-history-server`

Apache Hive/Apache Impala Services

- `impala-catalog`
- `impala-state-store`
- `impala-server`
- `hive-metastore`
- `hive-server2`

Other Important Services

- hue
- kafka-server
- mysqld
- zookeeper-server