**Hadoop Developer Training – Spark**

**Table of Contents**

[Getting Started 4](#_Toc451866057)

[Lab 1 : Explore RDDs Using the Spark Shell 6](#_Toc451866058)

[Lab 2 : Use RDDs to Transform a Dataset 9](#_Toc451866059)

[Lab 3: Process Data Files with Spark 12](#_Toc451866060)

[Lab 4: Use Pair RDDs to Join Two Datasets 15](#_Toc451866061)

[Lab 5: Write and Run a Spark Application 18](#_Toc451866062)

[Lab 6: Configure a Spark Application 21](#_Toc451866063)

[Lab 7: View Jobs and Stages in the Spark Application UI 24](#_Toc451866064)

[Lab 8: Persist an RDD 28](#_Toc451866065)

[Lab 9: Partition Data Files Using Spark 30](#_Toc451866066)

[Lab 10: Use Spark SQL 32](#_Toc451866067)

[Appendix A: Enabling iPython 35](#_Toc451866068)

# Getting Started

Working with the Virtual Machine

1. The VM is set to automatically log in as the user training. Should you log out at any time, you can log back in as the user training with the password training.

2. Should 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.

3. In some command-­-line steps in the exercises, you will see lines like this:

$ hdfs dfs –put Shakespeare /user/training/shakespeare

The dollar sign ($) at the beginning of each line indicates the Linux shell prompt. The actual prompt will include additional information (e.g., [training@localhost workspace]$ ) 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.

Points to note during the exercises

1. The main directory for the exercises for this course is

$DEV1/exercises (~/dev1/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.

2. Within each exercise directory you may find the following subdirectories:

a. solution – Solution code for each exercise.

b. stubs – A few of the exercises depend on provided starter files containing skeleton code.

c. 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.

3. Data files used in the exercises are in $DEV1DATA (~/DEV1DATA/data). Usually you will upload the files to HDFS before working with them.

4. As the exercises progress, and you gain more familiarity with Hadoop and Spark, 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 solutions provided, ask your instructor for assistance, or consult

with your fellow students!

# Lab 1 : Explore RDDs Using the Spark Shell

Exercise Directory: $DEV1/exercises/spark-shell

Data files (local): $DEV1DATA/frostroad.txt

In this Exercise you will start the Spark Shell and read a text file into a Resilient Distributed Data Set (RDD).

You may choose to do 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.

Start the Python Spark Shell

1. 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, hit Return a few times to clear the screen output.

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

pyspark> sc

Note: 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>

3. Using command completion, you can see all the available SparkContext methods:

type sc. (sc followed by a dot) and then the [TAB] key.

4. You can exit the shell by hitting Ctrl-­-D or by typing exit.

Start and Use the Scala Spark Shell

5. 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, hit Enter a few times to clear the screen output.

6. Spark creates a SparkContext object for you called sc. Make sure the object

exists:

scala> sc

Scala will display information about the sc object such as: res0: org.apache.spark.SparkContext = org.apache.spark.SparkContext@2f0301fa

7. Using command completion, you can see all the available SparkContext methods: type sc. (sc followed by a dot) and then the [TAB] key. Load and view text file (Python or Spark)

8. 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:

$DEV1DATA/frostroad.txt.

9. Define an RDD to be created by reading in the test file on the local file system. Use the first command if you are using Python, and the second one if you are using Scala.

pyspark> mydata = sc.textFile(file:/DEV1DATA/data/frostroad.txt")

scala> val mydata = sc. textFile("file:/DEV1DATA/data/frostroad.txt")

scala> val mydata = sc. textFile("file:/root/dev1/examples/example-data/purplecow.txt")

10. 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>> mydata.count()

scala> mydata.count()

The count operation causes the RDD to be materialized (created and populated). The number of lines (23) should be displayed, e.g.

Out[4]: 23 (Python) or

res0: 23 (Scala)

11. 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> mydata.collect()

scala> mydata.collect()

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

13. You can exit the shell at any time by typing exit.

# Lab 2 : Use RDDs to Transform a Dataset

Exercise Directory: $DEV1/exercises/spark-transform

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

In this Exercise you will practice using RDDs in the Spark Shell. Use Spark to explore the web server logs you captured in Flume earlier. Explore the Loudacre web log files

1.In this section you will be using weblogs you imported into HDFS in the Flume exercise.

2. Set a variable for the data file so you do not have to retype it each time.

pyspark> logfile="/loudacre/weblogs/FlumeData.\*"

scala> var logfile="/loudacre/weblogs/FlumeData.\*"

3. Create an RDD from the data file.

pyspark> logs = sc.textFile(logfile)

scala> var logs = sc.textFile(logfile)

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

pyspark> jpglogs=logs.filter(lambda line: ".jpg" in line)

scala> var jpglogs=logs.filter(line => line.contains(".jpg"))

scala> var jpglogs=mydata.filter(line => line.contains(".jpg"))

5. 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, if all you need is to count the number of JPG requests, you can execute this in a single

command:

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

6. 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> logs.map(lambda line: len(line)).take(5)

scala> logs.map(line => line.length).take(5)

This prints out an array of five integers corresponding to the first five lines in the file.

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

spark> logs.map(lambda line: line.split()).take(5)

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

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

8. 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> ips = logs.map(lambda line: line.split()[0])

pyspark> ips.take(5)

scala> var ips = logs.map(line =>line.split(' ')(0))

scala> ips.take(5)

9. 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 ips.take(10): print ip

scala> ips.take(10).foreach(println)

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

pyspark ips.saveAsTextFile("/loudacre/iplist")

scala> ips.saveAsTextFile("/loudacre/iplist")

11. In a terminal window or the Hue file browser, list the contents of the /loudcare/iplist folder. You should see multiple files, including several part-xxxxx 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; you will learn more about this later.) Review the contents of one of the files to confirm that they were created correctly.

# Lab 3: Process Data Files with Spark

Exercise Directory: $DEV1/exercises/spark-etl

Data files (local):

$DEV1DATA/activations/\*

$DEV1DATA/devicestatus.txt (Bonus)

Stubs:ActivationModels.pyspark

ActivationModels.scalaspark

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.

Review the API Documentation for RDD Operations

Visit the Spark API page you bookmarked previously. Follow the link at the top for the RDD class and review the list of available methods. The Data Review the data in $DEV1DATA/activations. Each XML file contains data for all the devices activated by customers during a specific month.

Copy this data to /loudacre in HDFS. 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>

The Task

Your code should 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

…

1. Start with the ActivationModels stub script in the exercise directory. (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.

2. 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.

3. 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.

4. 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.

5. Save the formatted strings to a text file in the directory

/loudacre/account-models.

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 $DEV1DATA/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 provider’s 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 to

• Load the dataset

• 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 (e.g. Ronin S2.) Split this field by spaces to separate the manufacturer from the model (e.g. manufacturer Ronin, model S2.)

• 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 $DEV1/exercises/spark- etl/bonus.

# Lab 4: Use Pair RDDs to Join Two Datasets

Exercise Directory: /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.

Explore Web Log Files Continue working with the web log files, as in the previous exercise.

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/\*6") would include only filenames ending with the digit 6.

1. Using map-­-reduce, 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 reduce to sum the values for each user ID. Your RDD data will be similar to:

(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])

…

Join Web Log Data with Account Data

In the Sqoop exercise you completed earlier, you imported data files containing Loudacre’s customer account data from MySQL to HDFS. Review that data now (located in /loudacre/accounts). 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 17:12:12,2013-07-1 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.

(userid1,([userid1,2008-11-24 10:04:08,\N,Cheryl,West,4905 Olive Street,San Francisco,CA,…],4))

(userid2,([ userid2,2008-11-2314:05:07,\N,Elizabeth,Kerns,4703 Eva Pearl

treet,Richmond,CA,…],8))

(userid3,([ userid3,2008-11-02 17:12:12,2013-07-1816:42:36,Melissa,Roman,3539 James Martin

Circle,Oakland,CA,…],1))

…

c. Display the user ID, hit count, and first name (3rd value) and last name (4th value) for the first 5 elements, e.g.:

userid1 4 Cheryl West userid2 8 Elizabeth Kerns userid3 1 Melissa Roman

…

Bonus Exercises

If you have more time, attempt the following challenges:

1. Challenge 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 challenge

2. Challenge 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. Challenge 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, e.g.

--- 85003

Jenkins,Thad Rick,Edward Lindsay,Ivy

…

--- 85004

Morris,Eric Reiser,Hazel Gregg,Alicia Preston,Elizabeth

…

# Lab 5: Write and Run a Spark Application

Exercise Directory: $DEV1/exercises/spark-application

Data files (HDFS): /loudacre/weblogs

Scala Project: countjpgs

Scala Classes: stubs.CountJPGs solution.CountJPGs

Python Stub: CountJPGs.py

Python Solution: $DEV1/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 in to the program as an argument.

This is the same task you did earlier in the “Use RDDs to Transform a Dataset” 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.

Write a Spark application in Python 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. A simple stub file to get started has been provided in the exercise project:

$DEV1/exercises/spark-application/CountJPGs.py. This stub imports the required Spark class and sets up your main code block.

2. Set up a SparkContext using the following code:

sc = SparkContext()

3. 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 “Getting Started with RDDs” exercise for the code to do this.

4. Run the program, passing the name of the log file to process, e.g.:

$ spark-submit CountJPGs.py /loudacre/weblogs/\*

Write a Spark application in Scala

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: $DEV1/exercises/spark- application/countjpgs.

1. Before starting this exercise, start the Archiva service; this service provides a local Maven repository that has been pre-­-cached with the libraries you will need for these exercises:

$ sudo service archiva start

2. Edit the Scala class defined in CountJPGs.scala in src/main/scala/stubs/.

3. Set up a SparkContext using the following code:

val sc = new 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 “Use RDDs to Explore and Transform a Dataset” exercise for the code to do this.

5. From the countjpgs project directory, build your project using the following command:

$ mvn package

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

$ spark-submit \

--class stubs.CountJPGs \

target/countjpgs-1.0.jar /loudacre/weblogs/\*

Submit 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.

1. 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/\*

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

3. Take note of the your application’s ID (e.g. application\_12345…). (You may wish to copy it.) In a terminal window, enter

$ yarn logs -applicationId <your-app-id>

# Lab 6: Configure a Spark Application

Exercise Directory: $DEV1/exercises/spark-application

Data files (HDFS)

/loudacre/weblogs/\*

Properties files (local) spark.conf log4j.properties

In this exercise you will practice setting various Spark configuration options. You will work with the CountJPGs program you wrote in the prior exercise.

Set configuration options at the command line

1. Rerun the CountJPGs Python or Scala program you wrote in the previous exercise, this time specifying an application name. For example:

$ spark-submit --master yarn-client \

--name 'Count JPGs' \

CountJPGs.py /loudacre/weblogs/\*

$ 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: View the Spark Application UI. 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.

Set configuration options in a configuration file

4. Change directories to your exercise working directory. (If you are working in Scala, that is the countjpgs project directory.)

5. Using a text editor, create a file in the 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

6. 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/\*

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

Set logging levels

8. Copy the template file /etc/spark/conf/log4j.properties.template to log4j.properties in your exercise working directory.

9. Edit log4j.properties . The first line currently reads:

log4j.rootCategory=INFO, console Replace INFO with DEBUG:

log4j.rootCategory=DEBUG, console

10. Rerun your Spark application. Because the current directory is on the Java classpath, you log4.properties file will set the logging level to DEBUG.

11. Notice that the output now contains both the INFO messages it did before and DEBUG messages, e.g.:

15/03/19 11:40:45 INFO MemoryStore: ensureFreeSpace(154293) called with curMem=0, maxMem=311387750

15/03/19 11:40:45 INFO MemoryStore: Block broadcast\_0 stored as values to memory (estimated size 150.7 KB, free 296.8 MB)

15/03/19 11:40:45 DEBUG BlockManager: Put block broadcast\_0 locally took 79 ms

15/03/19 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 generates unnecessarily distracting output.

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

13. You can also set the log level for the Spark Shell by placing the log4j.properties file in your working directory before starting the shell.Try starting the shell from the directory in which you placed the file and note that only WARN messages now appear.

Note: Throughout the rest of the exercises, you may change these settings depending on whether you find the extra logging messages helpful or distracting.

# Lab 7: View Jobs and Stages in the Spark Application UI

Exercise Directory: $DEV1/exercises/spark-stages

Data files (HDFS):

/loudacre/weblogs/\*

/loudacre/accounts/\*

Test Scripts: SparkStages.pyspark SparkStages.scalaspark

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.

Explore 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 2 threads:

$ pyspark --master local[2]

$ spark-shell --master local[2]

2. Review the accounts dataset (/loudacre/accounts/) using Hue or command line. Take note of the number of files.

3. Create an RDD based on a single file in the dataset, e.g.

/loudacre/accounts/part-m-00000 and then call toDebugString on the RDD, which displays the number of partitions in parentheses () before the RDD 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, create an 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. Bonus: use foreachPartition to print out the first record of each partition. Set up the job

7. Create an RDD of accounts, keyed by ID and with 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> var 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, e.g. textFile("/loudacre/weblogs/\*6").

pyspark> userreqs = sc.textFile("/loudacre/weblogs/\*6").map(lambda line: line.split()) \

.map(lambda words: (words[2],1)) \

.reduceByKey(lambda v1,v2: v1+v2)

scala> var userreqs = sc. textFile("/loudacre/weblogs/\*6"). 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 based on first name, last name and total hits:

pyspark> accounthits = accountsByID.join(userreqs).values()

scala> var 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

a. How many stages are in this job?

b. Which stages are dependent on which?

c. How many tasks will each stage consist of?

Run the job and review it in the Spark Application UI

11. In your browser, visiting 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. 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 on 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 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? The number of tasks in the first stages correspond to the number of partitions, which for this example corresponds to the number of files processed.

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.

2. Click on 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.)

3. 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.

4. 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?

Leave the Spark Shell running for the next exercise.

# Lab 8: Persist an RDD

Exercise Directory: $DEV1/exercises/spark-persist

Data files (HDFS):

/loudacre/weblogs/\*

/loudacre/accounts/\*

Job Setup Script:

$DEV1/spark-stages/SparkStages.pyspark

$DEV1/spark-stages/SparkStages.scalaspark

In this Exercise you will explore the performance effect of caching (that is persisting to memory) an RDD.

1. Make sure the Spark Shell is still running from the last exercise. If it isn’t, restart it (in local mode with 2 threads) and paste in the job setup code from the solution file or the previous exercise.

2. This time to start the job you are going to perform a slightly different action than last time: count the number of user accounts with a total hit count greater than five:

pyspark> accounthits.filter(lambda (firstlast,hitcount): hitcount > 5).count()

scala> accounthits.filter(pair => pair.\_2 > 5).count()

3. Cache (persist to memory) the RDD 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 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 on the RDD ID to see details about partitions and persistence.

8. Click on 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 StorageLevel.DISK\_ONLY and compare the storage report in the Spark Application Web UI. (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.)

# Lab 9: Partition Data Files Using Spark

Files and Data Used in this Exercise

Exercise Directory: $DEV1/exercises/spark-partfile

Data files:

/loudacre/devicestatus\_etl/\* (HDFS)

$DEV1DATA/status-regions.txt (local)

Stubs: partition-status.pyspark / .scalaspark

Output directory (HDFS): /loudacre/devstatusByRegion

In this exercise you will use Spark to create a dataset for device status data, partitioned by region.

This exercise brings together what you have learned about using Spark for data processing with the earlier chapter on Data File Partitioning.

The Task

In the previous exercise, you calculated the five k-­-means data points for the locations in a data set of device status records.

Now you will use those five locations to define five regions: A, B, C, D and E. For each region, the location from the previous exercise is the center point.

• Save the five data points you calculated above, or use the provided data file:

$DEV1DATA/status-regions.txt, which is in the format:

region,latitude,longitude region,latitude,longitude

…

•Start with the stub files in the exercise directory, which provides code to read the region location datafile a dictionary with region as key, and (latitude,longitude) pairs as value. It also provides a function to parse a line of the device status data, and return the region whose center point is closest to the location for the status data.

•Read the device status data and write out the same data such that the files are in partitioned directories; that is, all status for devices in the A region is written in a directory called region=A, and so on.

• In Hive or Impala, define a table for the device status data that is partitionedby region.

• If you are using Impala, manually add the five partitions to the table using ADD PARTITION. If you are using Hive, you can use the MSCK REPAIR TABLE command to automatically add the partitioned directories to the table.

# Lab 10: Use Spark SQL

Exercise Directory: $DEV1/exercises/spark-sql

MySQL table: loudacre.webpage

Output directory (HDFS): /loudacre/webpage\_files

In this exercise you will use Spark SQL to load data from MySQL, process it, and store it to HDFS.

Review the Data in MySQL

Review the data currently in the MySQL loudacre.mysql table.

1. List the columns and types in the table:

$ mysql -utraining -ptraining loudacre -e"describe webpage"

2. View the first few rows from the table:

$ mysql -utraining -ptraining loudacre -e"select \* from webpage limit 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 Spark SQL to extract the data in the column, split the string, and create a new dataset in HDFS containing each web page number, and its associated files in separate rows.

Load the Data from MySQL

3. If necessary, start the Spark Shell.

4. Import the SQLContext class definition, and define a SQL context:

scala> import org.apache.spark.sql.SQLContext

scala> val sqlCtx = new SQLContext(sc)

pyspark> from pyspark.sql import SQLContext

pyspark> sqlCtx = SQLContext(sc)

5. Create a new DataFrame based on the webpage table from the database:

scala> val webpages=sqlCtx.load("jdbc", Map("url"-> "jdbc:mysql://localhost/loudacre?user=training&password =training","dbtable" -> "webpage"))

pyspark>webpages=sqlCtx.load(source="jdbc",url="jdbc:mysql://localhost/loudacre?user=training&pass word=training", dbtable="webpage")

6. Examine the schema of the new DataFrame by calling webpages.printSchema().

7. Create a new DataFrame by selecting the web\_page\_num and associated files columns from the existing DataFrame:

scala> val assocfiles = webpages.select(webpages("web\_page\_num"),webpages("asso ciated\_files"))

python> assocfiles = webpages.select(webpages.web\_page\_num,webpages.associated\_files)

8. In order to manipulate the data using Spark, convert the DataFrame into a to a Pair RDD using the map method. The input into the map method is a Row object. They key is the web\_page\_num value (the first value in the Row), and the value is the associated\_files string (the second value in the Row).

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

index:

scala> val afilesrdd = assocfiles.map(row => (row.getInt(0),row.getString(1)))

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

pyspark> afilesrdd = assocfiles.map(lambda row: \ (row.web\_page\_num,row.associated\_files))

9. Now that you have an RDD, you can use the familiar flatMapValues transformation to split and extract the filenames in the associated\_files column:

scala> val afilesrdd2 =\afilesrdd.flatMapValues(filestring =>filestring.split(','))

pyspark> afilesrdd2 = afilesrdd.flatMapValues(lambda filestring:filestring.split(','))

10. Create a new DataFrame from the RDD:

scala> val afiledf = sqlCtx.createDataFrame(afilesrdd2)

pyspark> afiledf = sqlCtx.createDataFrame(afilesrdd2)

11. Call printSchema on the new DataFrame. Note that Spark SQL gave the columns generic names: \_1 and \_2.

12. Create a new DataFrame by renaming the columns to reflect the data they hold. In Scala, you can use the toDF shortcut method to create a new DataFrame based on an existing one with the columns renamed:

scala> val finaldf = afiledf.toDF("web\_page\_num","associated\_file")

In Python, use the withColumnRenamed method to rename the two columns:

pyspark> finaldf = afiledf. withColumnRenamed('\_1','web\_page\_num'). \

withColumnRenamed('\_2','associated\_file')

13. Call printSchema to confirm that the new DataFrame has the correct column names.

14. Your final DataFrame contains the processed data, so save it in Parquet format (the default) in /loudacre/webpage\_files. (The code is the same in Scala and Python)

> finaldf.save("/loudacre/webpage\_files")

View the Output

15. Using Hue or the HDFS command line, list the files that were saved by Spark SQL.

16. Execute the following DDL command in Impala to create a table to access the new Parquet dataset:

CREATE EXTERNAL TABLE webpage\_files LIKE PARQUET

'/loudacre/webpage\_files/part-r-00001.parquet' STORED AS PARQUET LOCATION '/loudacre/webpage\_files'

17. Try executing a simple query to confirm the table is set up correctly.

# Appendix A: Enabling iPython

Notebook

iPython Notebook is installed on the VM for this course. To use it instead of the command-­-line version of iPython, 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. (Must be a new terminal so it loads your edited.bashrc file).

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

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

7. Enter some spark code such as the following and use the play button to execute your spark code.

8. Notice the output displayed.