BUAN 6346 BIG DATA

DR. JERRY F. PEREZ

Assignment 3

07/12/2020

Jeffrey Chang / jlc190004

Contents

Lab-Chapter-10-Spark-Docs ..... - 3 -

View the Spark Documentation ..... - 3 -

Lab-Chapter-10-Explore-RDDs-Using-the-Spark-Shell ..... - 5 -

Start the Python Spark Shell ..... - 5 -

Start the Use Scala Spark Shell ..... - 7 -

Load and view text file (Python and Scala) ..... - 8 -

Lab-Chapter-10-Use-RDDs-to-Transform-a-Dataset ..... - 13 -

Using Python ..... - 13 -

Lab-Chapter-11-Process-Data-Files-with-Spark ..... - 19 -

Review the API Documentation for RDD Operations ..... - 19 -

The Data ..... - 19 -

The Task ..... - 21 -

Optional Homework ..... - 23 -

Lab-Chapter-12-Use-Pair-RDDs-to-Join-Two-Datasets ..... - 25 -

Explore Web Log Files ..... - 25 -

Join Web Log Data with Account Data ..... - 27 -

Optional Homework ..... - 28 -

Lab-Chapter-13- Write-and-Run-a-Spark-Application ..... - 30 -

Write a Spark application in Python ..... - 30 -

Submit a Spark application to the cluster ..... - 31 -

Lab-Chapter-13-Configure-a-Spark-Application ..... - 33 -

Set configuration options at the command line ..... - 33 -

Set configuration options in a configuration file ..... - 34 -

Set logging levels ..... - 35 -

Lab-Chapter-14-View-Jobs & Stages-in-the-Spark-Application-UI ..... - 38 -

Explore Partitioning of file-based RDDs ..... - 38 -

Set up the job ..... - 41 -

Run the job and review it in the Spark Application UI ..... - 42 -

Lab-Chapter-15-Presist an RDD ..... - 45 -

Optional ..... - 47 -

Lab-Chapter-16-Implement an Iterative Algorithm with Spark ..... - 50 -

Review the Data ..... - 50 -

Calculate k-means for device location ..... - 50 -

Lab-Chapter-16-Optional Homework: Partition-Data-Files-Using-Spark ..... - 55 -

The Task ..... - 55 -

Lab-Chapter-17-Use-Spark-SQL-for-ETL ..... - 60 -

Review the Data in MySQL ..... - 60 -

Load the Data from MySQL ..... - 61 -

View the Output ..... - 62 -

Lab-Chapter-10-Spark-Docs

View the Spark Documentation

The purpose of the Chapter is to look over the Spark documentation on the website, which had a bookmark in Virtual Machine's Firefox.
URL file:/usr/lib/spark/docs/_site/index.html

From the Programming Guides menu, select the Spark Programming Guide. I bookmark the page for later review. Moreover, you can choose the programing language you use; I prefer Python and briefly review the guide.

From the API Docs menu, I select Python API and bookmark the API page for use for Big Data class. PySpark is the Python API for Spark.

The example for the Scala API documents web page is showing below.

Lab-Chapter-10-Explore-RDDs-Using-the-Spark-Shell

In this exercise, we will start the Spark Shell and read a text file into a Resilient Distributed Data Set (RDD). I prefer to use Python to do this exercise, but also show the step by using the 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. Using [$pyspark]
2. Spark creates a SparkContext object for you called sc. Make sure the object exists, Pyspark will display information about the sc object.
   My version is *<pyspark.context.SparkContext at 0x7facad6e2e10>.*

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 the Use Scala Spark Shell

1. In a terminal window, start the Scala Spark Shell [$ spark-shell]
2. Spark creates a SparkContext object for you called sc. Make sure the object exists, Scala will display information about the sc.
   < s0: org.apache.spark.SparkContext = org.apache.spark.SparkContext@10340285 >
3. 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 and Scala)

Reviewing the simple text file, I use the Gedit, simple text editor, to view the file in a separate window, and the file is located at */home/training_materials/data/frostroad.t*xt

Using Scala

Define an RDD to be created by reading in the test file on the local file system.

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

Spark will not read the file until you execute an operation on the RDD. Using the command [*scala> mydata.count()*] count the number of lines in the dataset. The count operation causes the RDD to be materialized (created and populated). The number of lines (23) should be displayed: res1: Long = 23.

Executing the collect operation, [*scala> mydata.collect()*], to display the data in the RDD. This returns and presents the entire dataset. This is convenient for very small RDDs, but be careful using collect for more typical large datasets.

Type mydata, and then the [TAB] key, this will show all the available transformations and operations you can perform on an RDD. After all, I use the command [*exit*] to exit the shell.

Using Python

Define an RDD to be created by reading in the test file on the local file system.

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

Using the command [*pyspark> mydata.count()*] count the number of lines in the dataset.

Executing the collect operation, [*pyspark> mydata.collect()*], to display the data in the RDD.

Type mydata, and then the [TAB] key, this will show all the available transformations and operations you can perform on an RDD. After all, I use the command [exit] to exit the shell.

Lab-Chapter-10-Use-RDDs-to-Transform-a-Dataset

This Chapter uses Spark to explore previously collected web server logs in Flume. I will practice using RDDs in the Spark Shell with Python and Scala.

Using Hue File Browser, review one of the files in the HDFS /loudacre/weblogs directory, e.g., FlumeData.1592358967497.

I change to VMware 15.5 and the Net-ID I will show on the left-top*

Using Python

I do not have to retype it each time by setting a variable for the data file and create an RDD from the data file.

- pyspark> logfile="/loudacre/weblogs/FlumeData.*"
- pyspark> logs = sc.textFile(logfile)

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

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

View the first 10 lines of the data using *take*:

- pyspark > jpglogs.take(10)

Sometimes can combine the steps into a single line of code without intermediate store objects in a variable. For instance, I count the number of JPG and execute this in a single command. The number of counts is 17251.

pyspark> sc.textFile(logfile).filter(lambda line: ".jpg" in line).count()

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.{151, 143, 154, 147, 160}

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

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

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

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)

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

Last, save the list of IP addresses as a text file:

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

I am using the Hue file browser to check if the work above is successful. *iplist* file is creat under */loudacre*.

List the contents of the /loudacre/iplist folder. We can 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.)

Review the contents of one of the files, part-00000, to confirm that they were created correctly.

Lab-Chapter-11-Process-Data-Files-with-Spark

In this section, I'll parse a set of XML-formatted activation records to extract account numbers and model names. Spark is widely used for ETL operations (Extract / Transform / Load). Often data is stored in line-oriented records, such as the weblogs in the previous exercise, but often the data is in a multi-line format that must be processed as a whole file. In this exercise, instead of line based formats, I will practice working with file-based formats.

Review the API Documentation for RDD Operations

I reviewed different methods available for the RDD class, like *aggregate(), cache(), …etc.*
Website: file:///usr/lib/spark/docs/_site/api/python/pyspark.html#pyspark.RDD

The Data

First, review the data in *$DEV1DATA/activations.* Each XML file contains data for all the devices activated by customers during a specific month.

- $ ls /home/training/training_materials/data/activations

Second, copy this data to /loudacre in HDFS

- $ cd /home/training/training_materials/data
- $ hdfs dfs -put activations /loudacre
- $ hdfs dfs -ls /loudacre/activations

Third, Show a file 2008-10.xml as a sample.

cloudera-training-capspark-student-rev_cdh5.4.3a - VMware Workstation
File Edit View VM Tabs Help
Library
Net-ID: jlc190004
Home
cloudera-training-capspark-...
Applications Places System
Mon Jul 6, 3:43 PM
training@localhost:~/training_materials/data
[training@localhost spark-etl]$ ls /home/training/training_materials/data/activations
2008-10.xml 2009-03.xml 2009-08.xml 2010-01.xml 2010-06.xml 2010-11.xml 2011-04.xml 2011-09.xml 2012-02.xml 2012-07.xml 2012-12.xml 2013-05.xml 2013-10.xml 2014-03.xml
2008-11.xml 2009-04.xml 2009-09.xml 2010-02.xml 2010-07.xml 2010-12.xml 2011-05.xml 2011-10.xml 2012-03.xml 2012-08.xml 2013-01.xml 2013-06.xml 2013-11.xml
2008-12.xml 2009-05.xml 2009-10.xml 2010-03.xml 2010-08.xml 2011-01.xml 2011-06.xml 2011-11.xml 2012-04.xml 2012-09.xml 2013-02.xml 2013-07.xml 2013-12.xml
2009-01.xml 2009-06.xml 2009-11.xml 2010-04.xml 2010-09.xml 2011-02.xml 2011-07.xml 2011-12.xml 2012-05.xml 2012-10.xml 2013-03.xml 2013-08.xml 2014-01.xml
2009-02.xml 2009-07.xml 2009-12.xml 2010-05.xml 2010-10.xml 2011-03.xml 2011-08.xml 2012-01.xml 2012-06.xml 2012-11.xml 2013-04.xml 2013-09.xml 2014-02.xml
[training@localhost spark-etl]$ cd /home/training/training_materials/data
[training@localhost data]$ hdfs dfs -put activations /loudacre
[training@localhost data]$ hdfs dfs -ls /loudacre/activations
Found 66 items
-rw-rw-rw- 1 training supergroup 19044 2020-07-06 15:43 /loudacre/activations/2008-10.xml
-rw-rw-rw- 1 training supergroup 73748 2020-07-06 15:43 /loudacre/activations/2008-11.xml
-rw-rw-rw- 1 training supergroup 68764 2020-07-06 15:43 /loudacre/activations/2008-12.xml
-rw-rw-rw- 1 training supergroup 90337 2020-07-06 15:43 /loudacre/activations/2009-01.xml
-rw-rw-rw- 1 training supergroup 67655 2020-07-06 15:43 /loudacre/activations/2009-02.xml
-rw-rw-rw- 1 training supergroup 86243 2020-07-06 15:43 /loudacre/activations/2009-03.xml
-rw-rw-rw- 1 training supergroup 82718 2020-07-06 15:43 /loudacre/activations/2009-04.xml
-rw-rw-rw- 1 training supergroup 90148 2020-07-06 15:43 /loudacre/activations/2009-05.xml
-rw-rw-rw- 1 training supergroup 88713 2020-07-06 15:43 /loudacre/activations/2009-06.xml
-rw-rw-rw- 1 training supergroup 86835 2020-07-06 15:43 /loudacre/activations/2009-07.xml
-rw-rw-rw- 1 training supergroup 79114 2020-07-06 15:43 /loudacre/activations/2009-08.xml
-rw-rw-rw- 1 training supergroup 83012 2020-07-06 15:43 /loudacre/activations/2009-09.xml
-rw-rw-rw- 1 training supergroup 91007 2020-07-06 15:43 /loudacre/activations/2009-10.xml
-rw-rw-rw- 1 training supergroup 93132 2020-07-06 15:43 /loudacre/activations/2009-11.xml
-rw-rw-rw- 1 training supergroup 96328 2020-07-06 15:43 /loudacre/activations/2009-12.xml
-rw-rw-rw- 1 training supergroup 192394 2020-07-06 15:43 /loudacre/activations/2010-01.xml
-rw-rw-rw- 1 training supergroup 191631 2020-07-06 15:43 /loudacre/activations/2010-02.xml
-rw-rw-rw- 1 training supergroup 200690 2020-07-06 15:43 /loudacre/activations/2010-03.xml
-rw-rw-rw- 1 training supergroup 200510 2020-07-06 15:43 /loudacre/activations/2010-04.xml
-rw-rw-rw- 1 training supergroup 227033 2020-07-06 15:43 /loudacre/activations/2010-05.xml
-rw-rw-rw- 1 training supergroup 235350 2020-07-06 15:43 /loudacre/activations/2010-06.xml
-rw-rw-rw- 1 training supergroup 238267 2020-07-06 15:43 /loudacre/activations/2010-07.xml
-rw-rw-rw- 1 training supergroup 241761 2020-07-06 15:43 /loudacre/activations/2010-08.xml
-rw-rw-rw- 1 training supergroup 234168 2020-07-06 15:43 /loudacre/activations/2010-09.xml
-rw-rw-rw- 1 training supergroup 237083 2020-07-06 15:43 /loudacre/activations/2010-10.xml
-rw-rw-rw- 1 training supergroup 209493 2020-07-06 15:43 /loudacre/activations/2010-11.xml
-rw-rw-rw- 1 training supergroup 235910 2020-07-06 15:43 /loudacre/activations/2010-12.xml
-rw-rw-rw- 1 training supergroup 441580 2020-07-06 15:43 /loudacre/activations/2011-01.xml
-rw-rw-rw- 1 training supergroup 421089 2020-07-06 15:43 /loudacre/activations/2011-02.xml
-rw-rw-rw- 1 training supergroup 472902 2020-07-06 15:43 /loudacre/activations/2011-03.xml
-rw-rw-rw- 1 training supergroup 456871 2020-07-06 15:43 /loudacre/activations/2011-04.xml
-rw-rw-rw- 1 training supergroup 466384 2020-07-06 15:43 /loudacre/activations/2011-05.xml
-rw-rw-rw- 1 training supergroup 454844 2020-07-06 15:43 /loudacre/activations/2011-06.xml
-rw-rw-rw- 1 training supergroup 466854 2020-07-06 15:43 /loudacre/activations/2011-07.xml
-rw-rw-rw- 1 training supergroup 483014 2020-07-06 15:43 /loudacre/activations/2011-08.xml
-rw-rw-rw- 1 training supergroup 464367 2020-07-06 15:43 /loudacre/activations/2011-09.xml
-rw-rw-rw- 1 training supergroup 500909 2020-07-06 15:43 /loudacre/activations/2011-10.xml
Hue - File Browser - M...
exercises - File Browser
training@localhost:~/t...
To return to your computer, move the mouse pointer outside or press Ctrl+Alt.

cloudera-training-capspark-student-rev_cdh5.4.3a - VMware Workstation
File Edit View VM Tabs Help
Library
Net-ID: jlc190004
Home
cloudera-training-capspark-...
Applications Places System
Mon Jul 6, 3:44 PM
Hue - File Browser - 2008-10.xml - File Viewer - Mozilla Firefox
Hue - File Browser - ...
localhost:8888/filebrowser/view/loudacre/activations/2008-10.xml
Most Visited Cloudera Hue YARN RM Spark UI (local) Spark Doc Solr Admin UI Kite SDK Doc
Query Editors Data Browsers Workflows Search
File Browser Job Browser training
File Browser
ACTIONS
View as binary
Edit file
Download
View file location
Refresh
INFO
Last modified
July 6, 2020 3:43 p.m.
User
training
Group
supergroup
Size
18.6 KB
Mode
100666
Home / loudacre / activations / 2008-10.xml
Page 1 of 5
<activations>
<activation timestamp="1225462088" type="phone">
<account-number>9763</account-number>
<device-id>debea35e-3ecd-4ee7-b0dd-ad428d953f32</device-id>
<phone-number>7600763387</phone-number>
<model>MeeToo 1.0</model>
</activation>
<activation timestamp="1225461447" type="phone">
<account-number>426</account-number>
<device-id>38a1566d-524e-4137-bad8-b597d09b54b0</device-id>
<phone-number>5102521038</phone-number>
<model>Titanic 1000</model>
</activation>
<activation timestamp="1225446947" type="phone">
<account-number>383</account-number>
<device-id>513024a3-a828-4674-9ff6-041e9a851f18</device-id>
<phone-number>9162206560</phone-number>
<model>Sorrento F00L</model>
</activation>
<activation timestamp="1225445908" type="phone">
<account-number>404</account-number>
Hue - File Browser - 20...
exercises - File Browser
training@localhost:~/t...
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

The Task

I follow the instruction and practice to process a set of activation XML files and extract the account number and device model for each activation, and save the list to a file formatted as *account_number:model*.

1. Start with the *ActivationModels* by using Python.

```
# Stub code to copy into Spark Shell

        import xml.etree.ElementTree as ElementTree

# Given a string containing XML, parse the string, and

# return an iterator of activation XML records (Elements) contained in the string

        def getactivations(s):

          filetree = ElementTree.fromstring(s)

          return filetree.getiterator('activation')

# Given an activation record (XML Element), return the model name

        def getmodel(activation):

          return activation.find('model').text

# Given an activation record (XML Element), return the account number

        def getaccount(activation):

          return activation.find('account-number').text
```

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.

```
# Read XML files into an RDD

        files="/loudacre/activations/*.xml"

        activationFiles = sc.wholeTextFiles(files)
```

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.

```
# Parse each file (as a string) into a collection of activation XML records

activationRecords = activationFiles.flatMap(lambda (filename,xmlstring): getactivations(xmlstring))
```

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.

Map each activation record to "account-number:model-name"

```
models = activationRecords.map(lambda record: getaccount(record) + ":" + getmodel(record))
```

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

Save the data to a file

```
models.saveAsTextFile("/loudacre/account-models")
```

Check the result

Optional Homework

Another standard part of the ETL process is data scrubbing. In this preferred homework, I will process the data in a standardized format for later processing.
After reviewing the contents of the file $ DEV1DATA / devicestatus.txt. This file contains data collected from mobile devices on Loud Docker's network, including device ID, current status, location and more. Since Loud Dacre previously purchased other mobile provider networks, data from different subnetworks have a different format. Note that records in this file have different field limits: some utility commas, some utility pipes (|), and so on.

```
# Load the data file
devstatus = sc.textFile("/loudacre/devicestatus.txt")

# Filter out lines with < 10 characters, use the 20th character as the delimiter, parse the line, and filter out bad lines
cleanstatus = devstatus. \
   filter(lambda line: len(line) > 20). \
   map(lambda line: line.split(line[19:20])). \
   filter(lambda values: len(values) == 14)

# Create a new RDD containing date, manufacturer, device ID, latitude and longitude
devicedata = cleanstatus. \
```

```
    map(lambda values: (values[0], values[1].split(' ')[0], values[2], values[12], values[13]))

# Save to a CSV file as a comma-delimited string (trim parenthesis from tuple toString)
devicedata. \
    map(lambda values: ','.join(values)). \
    saveAsTextFile("/loudacre/devicestatus_etl")
```

The result,

Lab-Chapter-12-Use-Pair-RDDs-to-Join-Two-Datasets

In this part, I was exploring the Loudacre web server log files, as well as the Loudacre user account data, using key-value Pair RDDs.

Explore Web Log Files

It is recommended to use only a subset of files before starting since the operations can take a while. With a wild car, I created a specify wildcard, *textFile("/loudacre/weblogs/*6")*, would include only filenames ending with the digit 6. Using map-reduce, count the number of requests from each user.

a. Use the map to create a Pair RDD with the user ID as the key, and the integer one as the value. (The user ID is the third field in each line.)
b. Use reduce to sum the values for each user ID

```
# Step 1 - Create an RDD based on a subset of weblogs (those ending in digit 6)
        logs=sc.textFile("/loudacre/weblogs/*6")
# map each request (line) to a pair (userid, 1), then sum the values
        userreqs = logs \
          .map(lambda line: line.split()) \
          .map(lambda words: (words[2],1))  \
          .reduceByKey(lambda count1,count2: count1 + count2)
# Step 2 - Show the records for the 10 users with the highest counts

        freqcount = userreqs.map(lambda (userid,freq): (freq,userid)).countByKey()

        print freqcount
```

Step 3 - Group IPs by user ID

```
userips = logs \
    .map(lambda line: line.split()) \
    .map(lambda words: (words[2],words[0])) \
    .groupByKey()
```

```
# print out the first 10 user ids, and their IP list
        for (userid,ips) in userips.take(10):
          print userid, ":"
          for ip in ips: print "\t",ip
```

Join Web Log Data with Account Data

Continue on step 4

```
# Step 4a - Map account data to (userid,[values....])
        accounts = sc.textFile("/loudacre/accounts") \
          .map(lambda s: s.split(',')) \
          .map(lambda account: (account[0],account))
```

```
# Step 4b - Join account data with userreqs then merge hit count into valuelist
        accounthits = accounts.join(userreqs)
# Step 4c - Display userid, hit count, first name, last name for the first 5 elements
        for (userid,(values,count)) in accounthits.take(5) :
            print  userid, count, values[3],values[4]
```

Optional Homework

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.

```
accountsByPCode = sc.textFile("/loudacre/accounts") \
  .map(lambda s: s.split(','))\
  .keyBy(lambda account: account[8])
```

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.

```
namesByPCode = accountsByPCode\
  .mapValues(lambda account: account[4] + ',' + account[3]) \
  .groupByKey()
```

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.

```
for (pcode,names) in namesByPCode.sortByKey().take(5):
  print "---" ,pcode
  for name in names: print name
```

Lab-Chapter-13- Write-and-Run-a-Spark-Application

In this section, I will practice to write my own Spark application instead of using the interactive Spark Shell application.

Write a simple program that counts the number of JPG requests in a weblog file. The name of the file should be passed into 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, and I have to set up the SparkContext object. I will use Python and follow the appropriate set of instructions below to write a Spark program. Before running the program, I need to be sure to exit from the Spark Shell.

Write a Spark application in Python

I use Gedit to write Python. And save as name CountJPGs.py. Set up a SparkContext using the following code to sc = SparkContext(), and in the body of the program, load the file passed into the program, count the number of JPG requests, and display the count.

Run the program code at Exercise Directory, and the number of JPG is 17251.

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

Submit a Spark application to the cluster

In the previous section, I ran a Python using spark-submit. By default, spark-submit runs the application locally. In this section, run the application on the YARN cluster instead. Re-run the program by Python, specifying the cluster master in order to run it on the cluster.

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

The process application browses by YARN Resource Manager UI.
And the processing is from unsuccess to success.

The number of JPG is 17251.

In a terminal window, enter: $ yarn logs -applicationId application_1594058023303_0007

Lab-Chapter-13-Configure-a-Spark-Application

In this part, I will practice setting various Spark configuration options. I will work with the CountJPGs program in the prior section.

Set configuration options at the command line

Rerun the CountJPGs Python, this time specifying an application name. For example:

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

I visit the Resource Manager UI again and note the application name listed is in the command line.

Set configuration options in a configuration file

Change directories to your exercise working directory, and using a text editor, create a file in the working directory called myspark.conf.

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/*
```

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

Copy the template file */etc/spark/conf/log4j.properties.template* to *log4j.properties* in your exercise working directory. Edit log4j.properties and replace the first line currently reads INFO into DEBUG.

Notice that the output now contains both the INFO messages it did before and DEBUG messages, Debug logging can be useful when debugging, testing, or optimizing your code, but in most cases generates unnecessarily distracting output.

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.

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-Chapter-14-View-Jobs & Stages-in-the-Spark-Application-UI

In this section, I will use the Spark Application UI to view the execution stages for a job. In a previous article, I run 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 I will explore the stages and tasks involved in that job.

Explore Partitioning of file-based RDDs

Start the Spark Shell. Although in this Cloudera practice typically run a Spark application on a cluster, and the 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]

Review the accounts dataset (/loudacre/accounts/) using Hue. The number of files is 4

Create an RDD based on a single file in the dataset and call toDebugString on the RDD
First run,

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

The number of partitions in parentheses () before the RDD id is 2 and the number of partitions in the resulting RDD is 1.
Second run,

- pyspark> accounts=sc.textFile("/loudacre/accounts/part-m-00000",3)
- pyspark> print accounts.toDebugString()

The number of partitions in parentheses () before the RDD id is 3 and the number of partitions in the resulting RDD is 3. The RDD correctly have three partitions.

Finally, create an RDD based on all the files in the accounts dataset.

- pyspark> accounts=sc.textFile("/loudacre/accounts/*")
- pyspark> print accounts.toDebugString()

The number of partitions in parentheses () before the RDD id is 4, and the number of partitions in the resulting RDD is 5. The RDD correctly has three partitions.

Bonus: Use *foreachPartition* to print out the first record of each partition. I create the iterator inside this function and use lambda apply the function to each partition.

```
def printFirstLine(iter):
    print iter.next()
accounts.foreachPartition(lambda i: printFirstLine(i))
```

cloudera-training-capspark-student-rev_cdh5.4.3a - VMware Workstation
File Edit View VM Tabs Help
Library
Net-ID: jlc190004
Home
cloudera-training-capspark-...
Applications Places System
Mon Jul 6, 4:48 PM
training@localhost:~/training_materials/dev1/exercises/spark-application
In [4]: accounts=sc.textFile("/loudacre/accounts/part-m-00000")
In [5]: print accounts.toDebugString()
(2) /loudacre/accounts/part-m-00000 MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:-2 []
| /loudacre/accounts/part-m-00000 HadoopRDD[0] at textFile at NativeMethodAccessorImpl.java:-2 []
In [6]: accounts=sc.textFile("/loudacre/accounts/part-m-00000",3)
In [7]: print accounts.toDebugString()
(3) /loudacre/accounts/part-m-00000 MapPartitionsRDD[3] at textFile at NativeMethodAccessorImpl.java:-2 []
| /loudacre/accounts/part-m-00000 HadoopRDD[2] at textFile at NativeMethodAccessorImpl.java:-2 []
In [8]: accounts=sc.textFile("/loudacre/accounts/*")
In [9]: print accounts.toDebugString()
(4) /loudacre/accounts/* MapPartitionsRDD[5] at textFile at NativeMethodAccessorImpl.java:-2 []
| /loudacre/accounts/* HadoopRDD[4] at textFile at NativeMethodAccessorImpl.java:-2 []
In [10]:
In [10]: def printFirstLine(iter):
....: print iter.next()
....:
In [11]: accounts.foreachPartition(lambda i: printFirstLine(i))
32441,2012-07-06 12:35:29.0,\N,Peter,Zachary,1891 Jessie Street,Long Beach,CA,90743,5628719002,2014-03-18 13:30:46.0,2014-03-18 13:30:46.0
1,2008-10-23 16:05:05.0,\N,Donald,Becton,2275 Washburn Street,Oakland,CA,94660,5100032418,2014-03-18 13:29:47.0,2014-03-18 13:29:47.0
97321,2013-09-17 19:51:32.0,2014-01-17 09:33:00.0,Alan,Good,1628 Red Dog Road,Reno,NV,89408,7757963641,2014-03-18 13:32:51.0,2014-03-18 13:32:51.0
64881,2012-04-22 23:00:08.0,2014-02-24 21:23:35.0,Virginia,Etter,1297 Geneva Street,San Francisco,CA,94032,4150467661,2014-03-18 13:31:52.0,2014-03-18 13:31:52.0
In [12]:
Hue - File Browser - M...
spark-application - Fil...
training@localhost:~/t...
log4j.properties (~/trai...
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

Set up the job

First, 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]))
```

The first step is to 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. 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)
```

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()
```

Print the results of accounthits.toDebugString and review the output.

```
pyspark> print accounthits.toDebugString()
```

a. There are 3 stages in this job.

b. 3rd-stage is dependent on 2nd-stage, and 2nd-stage is dependent on 1st-stage.

c. There are 9 tasks in 1st-stage, 13 tasks in 2nd-stage, and 13 in 3rd-stage. A total of 35 tasks.

Run the job and review it in the Spark Application UI

Visiting the Spark Application UI and make sure the Jobs tab is selected. No jobs are yet running, so the list will be empty.

Return to the shell and start the job by executing an action

```
pyspark> accounthits.saveAsTextFile("/loudacre/userreqs")
```

Reload the Spark UI Jobs page in, and the job will appear in the Active Jobs list until it completes, and then it will display in the Completed Jobs List.

Click on the job description (which is the last action in the job) to see the stages.

- There are 3 stages of the job and it matches the number on the previous, which is 35.
- The numbers and the order is related to previous. the times of the stages were :

  3rd-stage:2020/07/06 17:01:26

  2nd-stage:2020/07/06 17:01:24

  1st-stage:2020/07/06 17:01:22
- There are 9 tasks in 1st-stage, 13 tasks in 2nd-stage and 13 in 3rd-stage. A total of 35 tasks. The number of tasks in the 1st-stage corresponds to the number of partitions, which is the number of files I processed.
- 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.

Notice:

- The Summary Metrics area shows how much time has been spent on various steps. This can help reduce performance problems.
- The Tasks section lists each task. The Locality Level column indicates whether or not this process works on the same node where the partition is physically stored. Keep in mind that Spark always tries to run tasks where the data is present, but not always if the node is busy.
- In a real-world cluster, the executor in the Task area will display different worker nodes that run the column tasks. (In this single node cluster, all tasks run on the same host: localhost.)
- When the job is finished, return to the Jobs tab to see the final statistics for the number of tasks executed and the time taken for the job.
- If re-running the last action, it will probably find that the job completes much faster and that several stages (and the tasks in them) show as “skipped.”

The details screenshots of about 1st-stage ~ 3rd-stage are on the next page

PySparkShell - Details for Stage 1 - Mozilla Firefox

Spark 1.3.0 — Jobs | Stages | Storage | Environment | Executors — PySparkShell application UI

Details for Stage 1

Total task time across all tasks: 2 s
Input Size / Records: 4.5 MB / 31397
Shuffle write: 118.2 KB / 405

▸ Show additional metrics

Summary Metrics for 9 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	85 ms	0.2 s	0.2 s	0.3 s	0.4 s
GC Time	0 ms	0 ms	0 ms	0 ms	8 ms
Input Size / Records	515.4 KB / 3482	515.4 KB / 3486	515.4 KB / 3487	515.5 KB / 3492	515.5 KB / 3493
Shuffle Write Size / Records	13.0 KB / 45	13.1 KB / 45	13.1 KB / 45	13.2 KB / 45	13.4 KB / 45

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input Size / Records	Shuffle Write Size / Records
<driver>	localhost:45589	2 s	9	0	9	4.5 MB / 31397	118.2 KB / 405

Tasks

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records	Write Time	Shuffle Write Size / Records	Errors

PySparkShell - Details for Stage 2 - Mozilla Firefox

Spark 1.3.0 — Jobs | Stages | Storage | Environment | Executors — PySparkShell application UI

Details for Stage 2

Total task time across all tasks: 5 s
Input Size / Records: 17.9 MB / 129761
Shuffle read: 118.2 KB / 405
Shuffle write: 3.9 MB / 975

▸ Show additional metrics

Summary Metrics for 13 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	91 ms	0.1 s	0.2 s	0.8 s	0.9 s
GC Time	0 ms	0 ms	0 ms	7 ms	10 ms
Input Size / Records	0.0 B / 0	0.0 B / 0	0.0 B / 0	4.4 MB / 32440	4.5 MB / 32441
Shuffle Read Size / Records	0.0 B / 0	0.0 B / 0	12.7 KB / 45	13.5 KB / 45	13.8 KB / 45
Shuffle Write Size / Records	9.7 KB / 39	10.1 KB / 39	10.2 KB / 39	981.5 KB / 156	997.1 KB / 156

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input Size / Records	Shuffle Read Size / Records	Shuffle Write Size / Records
<driver>	localhost:45589	5 s	13	0	13	17.9 MB / 129761	118.2 KB / 405	3.9 MB / 975

PySparkShell - Details for Stage 3 - Mozilla Firefox

Spark 1.3.0 — Jobs | Stages | Storage | Environment | Executors — PySparkShell application UI

Details for Stage 3

Total task time across all tasks: 4 s
Output: 102.5 KB / 4750
Shuffle read: 3.9 MB / 975

▸ Show additional metrics

Summary Metrics for 13 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.2 s	0.2 s	0.3 s	0.3 s	0.4 s
GC Time	0 ms	0 ms	0 ms	8 ms	14 ms
Output Size / Records	7.3 KB / 338	7.6 KB / 355	7.9 KB / 365	8.1 KB / 376	8.6 KB / 396
Shuffle Read Size / Records	307.9 KB / 75	309.3 KB / 75	310.5 KB / 75	311.6 KB / 75	313.6 KB / 75

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Output Size / Records	Shuffle Read Size / Records
<driver>	localhost:45589	4 s	13	0	13	102.5 KB / 4750	3.9 MB / 975

Tasks

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Output Size / Records	Shuffle Read Size / Records	Errors

Lab-Chapter-15-Presist an RDD

In this section, I will explore the performance effect of caching (that is, persisting to memory) an RDD. I restart the Spark in local mode with 2 threads and paste in the job setup code.

```
$ pyspark --master local[2]
```

This time to start the job to perform a different filter code than last time, which will count the number of users' accounts with a total hit count greater than five. The numbers are 2090.

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

Cache (persist to memory) the RDD by calling accounthits.persist().

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

I executed the count again and viewed the RDD's toDebugString. (the output indicates the persistence level selected.)

Reload the Storage tab in your browser, and this time note that the RDD you persisted is shown.

Click on the Executors tab and the amount of memory used is 85.4 KB and the remaining memory is 267.2 MB for one worker node.

the Executors tab that in our class vm has only one node showing one executor. Also, not all of the dataset is cached in our example. If have more nodes can have more memory, more data can be cached.

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.)

```
pyspark> accounthits.unpersist()
pyspark> from pyspark import StorageLevel
pyspark> accounthits.persist(StorageLevel.DISK_ONLY)
```

And run count again,

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

After running the code, you can observe that size in Memory and Size in Disk and the Memory Usage, …etc., are not the same.

Lab-Chapter-16-Implement an Iterative Algorithm with Spark

In this section, I will calculate k-means for a set of points by practice implementing iterative algorithms in Spark.

Review the Data

Examine the data I create in Lab-Chapter-11-Process-Data-Files-with-Spark option homework part

Calculate k-means for device location

Addition: k-means clustering is a vector calibration method, originally from signal processing, which aims to divide n observations into k clusters, in which each observation is owned by the cluster near the cluster (cluster centers or cluster centroid), acting as a model. Masonry. It divides the data space into Voronoi cells. It is famous for cluster analysis in data mining. k-means clustering reduces within-cluster variables (squared Euclidean distances) but not conventional Euclidean distances, which is a more difficult Weber problem: improving the mean squared error, while the geometric mean only reduces the Euclidean distance. For example, you can find better Euclidean solutions using K-medians and K-medoids (https://en.wikipedia.org/wiki/K-means_clustering)

Start by copying the provided KMeansCoords stub file

• 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.

```
# for a point p and an array of points, return the index in the array of the point closest to p
        def closestPoint(p, points):
            bestIndex = 0
            closest = float("+inf")
            # for each point in the array, calculate the distance to the test point, then return
            # the index of the array point with the smallest distance
            for i in range(len(points)):
                dist = distanceSquared(p,points[i])
                if dist < closest:
                    closest = dist
                    bestIndex = i
            return bestIndex
 # The squared distances between two points
        def distanceSquared(p1,p2):
            return (p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2
# The sum of two points
        def addPoints(p1,p2):
            return [p1[0] + p2[0], p1[1] + p2[1]]
# The files with device status data
        filename = "/loudacre/devicestatus_etl/*"
```

Set the variable K (the number of means to calculate). For this use **K=5**.

Set 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.**

```
# Split by delimiter,
# Parse latitude and longitude (4th and 5th fields) into two-element arrays
# Filter out records where lat/long is unavailable -- ie: 0/0 points
        points = sc.textFile(filename)\
            .map(lambda line: line.split(","))\
            .map(lambda fields: [float(fields[3]),float(fields[4])])\
            .filter(lambda point: sum(point) != 0)\
            .persist()

# start with K randomly selected points from the dataset
        kPoints = points.takeSample(False, K, 34)
        print "Starting points:", kPoints
```

The starting K center points:

[[32.3421945104, -111.683188189], [34.124514613, -118.137235223], [42.7513633178, -122.337468023], [33.7450189116, -117.718768896], [38.3144423144, -122.341396989]]

Create a K-length array called kPoints by taking a random sample of K location points from the RDD as starting means (center points).

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

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 each 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.)
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 the number of closest points.
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 *(index,(totalX,totalY),n)* to *(index,(totalX/n, 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 each center "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.

```
# loop until the total distance between one iteration's points and the next is less than the convergence distance specified

tempDist = float("+inf")
while tempDist > convergeDist:
  # for each point, find the index of the closest kpoint.  map to (index, (point,1))
  closest = points.map(lambda p : (closestPoint(p, kPoints), (p, 1)))
  # For each key (k-point index), reduce by adding the coordinates and number of points
  pointStats = closest.reduceByKey(lambda (point1,n1),(point2,n2):  (addPoints(point1,point2),n1+n2) )
  # For each key (k-point index), find a new point by calculating the average of each closest point
  newPoints = pointStats.map(lambda (i,(point,n)): (i,[point[0]/n,point[1]/n])).collect()
  # calculate the total of the distance between the current points and new points
  tempDist=0
  for  (i,point) in newPoints: tempDist += distanceSquared(kPoints[i],point)
  print "Distance between iterations:",tempDist
  # Copy the new points to the kPoints array for the next iteration
  for (i, point) in newPoints: kPoints[i] = point

print "Final center points: " + str(kPoints)
```

The Final K center points:

[[36.84127339936045, -118.59276074471572], [34.124514613, -118.137235223], [42.7513633178, -122.337468023], [33.7450189116, -117.718768896], [38.3144423144, -122.341396989]]

Lab-Chapter-16-Optional Homework: Partition-Data-Files-Using-Spark

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

The Task

In the previous exercise, I calculated the five k-means data points for the locations in a data set of device status records (above). Now I will use those five locations to define five *regions*: A, B, C, D and E. For each region, I use the location of the Final K center points: [[36.84127339936045, -118.59276074471572], [34.124514613, -118.137235223], [42.7513633178, -122.337468023], [33.7450189116, -117.718768896], [38.3144423144, -122.341396989]] to be the center points.

Save the five data points you calculated above to *$DEV1DATA/status-regions.txt*

Start with the stub files in the exercise directory "partition-status" which including the 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.

```
regionfile="/home/training/training_materials/data/status-regions.txt"
r1 = map(lambda x: x.strip().split(','),open(regionfile,'r'))
regions = dict(map(lambda (region,lat,lon): [region,(float(lat),float(lon))],r1))

def distanceSquared(p1,p2):
   return (p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2
```

```
# given a dictionary of region IDs and their locations, and a line the devstatus file, return the region ID
# of the region closest to the given point
def findRegion(s, regions):
    # parse the input string to find the point
    fields=s.split(',')
    p=(float(fields[3]),float(fields[4]))
    # starting distance is infinity
    closest = float("+inf")
    # for each region in the dict, calculate the distance to the test point, then return
    # the key of the point with the smallest distance
    for (region,location) in regions.items():
        dist = distanceSquared(p,location)
        if dist < closest:
            closest = dist
            closestRegion = region
    return closestRegion
```

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.

```
# The device status data file(s)
devstatus = sc.textFile("/loudacre/devicestatus_etl/*")
devstatusByRegion = devstatus.keyBy(lambda line: findRegion(line,regions))
for region in regions.iterkeys():
    print "Saving data for region: ",region
    devstatusByRegion.filter(lambda (k,v): region == k).\
      values().\
      saveAsTextFile("/loudacre/devstatusByRegion/region="+region)
```

cloudera-training-capspark-student-rev_cdh5.4.3a - VMware Workstation
training@localhost:~
In [29]: for region in regions.iterkeys():
....:   print "Saving data for region: ",region
....:   devstatusByRegion.filter(lambda (k,v): region == k).\
....:   values().\
....:   saveAsTextFile("/loudacre/devstatusByRegion/region="+region)
Saving data for region: A

cloudera-training-capspark-student-rev_cdh5.4.3a - VMware Workstation
training@localhost:~
Saving data for region: B
Saving data for region: C
Saving data for region: E

cloudera-training-capspark-student-rev_cdh5.4.3a - VMware Workstation
training@localhost:~
Saving data for region: D
In [30]:

In Impala, define a table for the device status data that is partitioned by region.

```
DROP TABLE IF EXISTS devstatus_by_region;
CREATE EXTERNAL TABLE devstatus_by_region
    (dt STRING,
     manufacturer STRING,
     dev_id STRING,
     latitude FLOAT,
     longitude FLOAT)
  PARTITIONED BY (region STRING)
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  LOCATION '/loudacre/devstatusByRegion';
```

I am using Impala, manually add the five partitions to the table using ADD PARTITION.

Check the result by Hue.

Lab-Chapter-17-Use-Spark-SQL-for-ETL

In this exercise, I 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

List the columns and types in the table:

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

View the first few rows from the table:

```
$ mysql -utraining -ptraining loudacre -e"select * from webpage limit 5"
```

The data in the *associated_files* column is a comma-separated string. Loudacre wants to make this data available in the Impala table, but to perform the necessary analysis, the data of the *associated_files* must be extracted and normalized. My goal is to use Spark SQL to extract the data in the column, separate the string, and create a new dataset in HDFS with each webpage number and associated files in separate rows.

Load the Data from MySQL

I use Scala to Import the SQLContext class definition, and define a SQL context:

```
scala> import org.apache.spark.sql.SQLContext
scala> val sqlCtx = new SQLContext(sc)
```

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"))
```

Examine the schema of the new DataFrame by calling

```
scala> webpages.printSchema()
```

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("associated_files"))
```

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. The 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)))
```

Now that I have an RDD, so I 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(','))
```

Create a new DataFrame from the RDD.

```
scala> val afiledf = sqlCtx.createDataFrame(afilesrdd2)
```

Call printSchema on the new DataFrame.

Finally, 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

Using Hue to list the files that were saved by Spark SQL.

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'

I used a simple query to confirm the table, whether set up correctly.

~~ This is the End of the Homework ~~