Ask Bigger Questions

Cloudera Developer Training for Apache Spark: Hands-On Exercises

General Notes

Cloudera's training courses use a Virtual Machine running the CentOS Linux distribution. This VM has Spark and CDH 5 (Cloudera's Distribution, including Apache Hadoop) installed. Although Hadoop and Spark typically run on a cluster of multiple machines, in this course you will be using a cluster running locally on a single node, which is referred to as Pseudo-Distributed mode.

Getting Started

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`.

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.

Working with the Virtual Machine

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.

Some commands are to be executed in the Python or Scala Spark Shells; those are shown with `pyspark>` or `scala>` prompts respectively.

The backslash (`\`) at the end of the first line of a command 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.

Completing the Exercises

As the exercises progress, and you gain more familiarity with 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.

Solutions

Spark Shell solutions (Python or Scala)

Many of the exercises in this course are done in the interactive Spark Shell. If you need help completing an exercise, you can refer to the solutions files in `~/training_materials/sparkdev/solutions`. Files with a `.pyspark` extension contain commands that are meant to be pasted into the Python Spark Shell; files ending in `.scalaspark` are meant to be pasted into the Scala Spark Shell.

The Python shell is particular about code pastes because of the necessity for proper whitespace/tab alignment. To avoid this issue, we suggest you use iPython, and use the `%paste` "magic" function instead of the terminal window's paste function.

Spark Application solutions

Some of the exercises involve running complete programs rather than using the interactive Spark Shell. For these, Python solutions are in `~/training_materials/sparkdev/solutions` with the extension `.py`. Run these solutions using spark-submit as described in the "Writing a Spark Application" exercise.

Scala applications are provided in the context of a Maven project directory, located in `~/exercises/projects`. Within the project directory, the solution code is in `src/main/scala/solution`. Follow the instructions in the "Writing a Spark Application" exercise to compile, package, and run Scala solutions.

Hands-On Exercise: Setting Up

Files Used in This Exercise:

`~/scripts/sparkdev/training_setup_sparkdev.sh`

In this Exercise you will set up your course environment.

Complete this exercise before moving on to later exercises.

Set Up Your Environment

1. Before starting the exercises, run the course setup script in a terminal window:

```
$ ~/scripts/sparkdev/training_setup_sparkdev.sh
```

   This script does the following:

   - Sets up workspace projects for the course exercises in `~/exercises`
   - Makes sure the correct services are running for the cluster

This is the end of the Exercise

Hands-On Exercise: Viewing the Spark Documentation

In this Exercise you will familiarize yourself with the Spark documentation.

1. Start Firefox in your Virtual Machine and visit the Spark documentation on your local machine, using the provided bookmark or opening the URL
`file:/usr/lib/spark/docs/_site/index.html`

2. From the **Programming Guides** menu, select the **Spark Programming Guide**. Briefly review the guide. You may wish to bookmark the page for later review.

3. From the **API Docs** menu, select either **Scaladoc** or **Python API**, depending on your language preference. Bookmark the API page for use during class. Later exercises will refer you to this documentation.

This is the end of the Exercise

Hands-On Exercise: Using the Spark Shell

In this Exercise you will start the Spark Shell and read a 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.

Most of the later exercises assume you are using Python, but Scala solutions are provided on your virtual machine, so you should feel free to use Scala if you prefer.

Using 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.

   > Note: Your environment is set up to use IPython shell by default. If you would prefer to use the regular Python shell, set `IPYTHON=0` before starting pyspark.

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

```
pyspark> sc
```

   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`.

Using the Scala Spark Shell

1. 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.

2. 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
```

5. Using command completion, you can see all the available SparkContext methods: type `sc.` (`sc` followed by a dot) and then the [TAB] key.

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

This is the end of the Exercise

Hands-On Exercise: Getting Started with RDDs

Files Used in This Exercise:

Data files (local):

`~/training_materials/sparkdev/data/frostroad.txt`

`~/training_materials/sparkdev/data/weblogs/2013-09-15.log`

Solution:

`solutions/LogIPs.pyspark`

`solutions/LogIPs.scalaspark`

In this Exercise you will practice using RDDs in the Spark Shell.

You will start by reading a simple text file. Then you will use Spark to explore the Apache web server output logs of the customer service site of a fictional mobile phone service provider called Loudacre.

Load and view text file

1. Start the Spark Shell if you exited it from the previous exercise. You may use either Scala (`spark-shell`) or Python (`pyspark`). These instructions assume you are using Python.

2. Review the simple text file we will be using by viewing (without editing) the file in a text editor. The file is located at: `~/training_materials/sparkdev/data/frostroad.txt`.

3. Define an RDD to be created by reading in a simple test file:

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

4. Note that 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()
```

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

```
Out[4]: 23
```

5. 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.

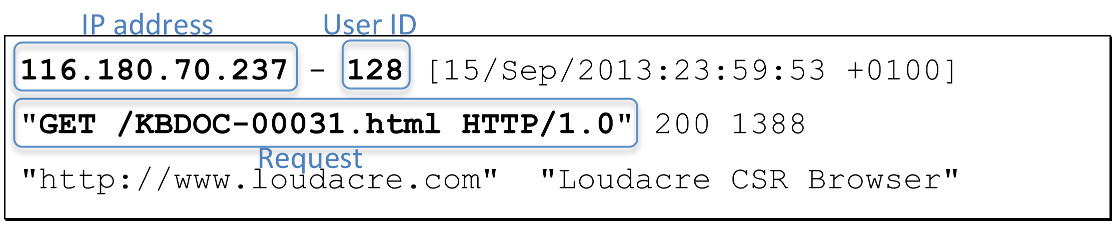
```
pyspark> mydata.collect()
```

6. 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.

Explore the Loudacre web log files

In this exercise you will be using data in `~/training_materials/sparkdev/data/weblogs`. Initially you will work with the log file from a single day. Later you will work with the full data set consisting of many days worth of logs.

7. Review one of the `.log` files in the directory. Note the format of the lines, e.g.



```
116.180.70.237 - 128 [15/Sep/2013:23:59:53 +0100]
"GET /KBDOC-00031.html HTTP/1.0" 200 1388
"http://www.loudacre.com"  "Loudacre CSR Browser"
```

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

```
pyspark> logfile="file:/home/training/\
training_materials/sparkdev/data/weblogs/\
2013-09-15.log"
```

9. Create an RDD from the data file.

```
pyspark> logs = sc.textFile(logfile)
```

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

```
pyspark> jpglogs=\
logs.filter(lambda x: ".jpg" in x)
```

11. View the first 10 lines of the data using `take`:

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

12. Sometimes you do not need to store intermediate data 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:

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

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

```
pyspark> logs.map(lambda s: len(s)).take(5)
```

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

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

```
pyspark> logs.map(lambda s: s.split()).take(5)
```

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

15. 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 s: s.split()[0])
pyspark> ips.take(5)
```

16. 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 x in ips.take(10): print x
```

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

```
pyspark> ips.saveAsTextFile(\
"file:/home/training/iplist")
```

18. In a terminal window, list the contents of the `/home/training/iplist` folder. You should see multiple files. The one you care about is `part-00000`, which should contain the list of IP addresses. "Part" (partition) files are numbered because there may be results from multiple tasks running on the cluster; you will learn more about this later.

If You Have More Time

If you have more time, attempt the following challenges:

1. Challenge 1: As you did in the previous step, save a list of IP addresses, but this time, use the whole web log data set (`weblogs/*`) instead of a single day's log.

   - Tip: You can use the up-arrow to edit and execute previous commands. You should only need to modify the lines that read and save the files.

2. Challenge 2: Use RDD transformations to create a dataset consisting of the IP address and corresponding user ID for each request for an HTML file. (Disregard requests for other file types). The user ID is the third field in each log file line.

   Display the data in the form *`ipaddress/userid`*, e.g.:

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

Review the API Documentation for RDD Operations

1. 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.

This is the end of the Exercise

Hands-On Exercise: Working with Pair RDDs

Files Used in This Exercise:

Data files (local)

```
~/training_materials/sparkdev/data/weblogs/*
~/training_materials/sparkdev/data/accounts.csv
```

Solution:

```
solutions/UserRequests.pyspark
solutions/UserRequests.scalaspark
```

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.

This time, work with the entire set of data files in the weblog folder rather than just a single day's logs.

1. Using MapReduce, 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. Display the user IDs and hit count for the users with the 10 highest hit counts.

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

   ```
   (5,userid)
   (7,userid)
   (2,userid)
   …
   ```

   b. Use `sortByKey(False)` to sort the swapped data by count.

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])
…
```

4. The data set in the `~/training_materials/sparkdev/data/accounts.csv` consists of information about Loudacre's user 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.

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. Map the accounts data to key/value-list pairs: (userid, [values…])

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

b. Join the Pair RDD with the set of userid/hit counts calculated in the first step.

```
(userid1,([userid1,2008-11-24
10:04:08,\N,Cheryl,West,4905 Olive Street,San
Francisco,CA,…],4))
(userid2,([ userid2,2008-11-23
14:05:07,\N,Elizabeth,Kerns,4703 Eva Pearl
Street,Richmond,CA,…],8))
(userid3,([ userid3,2008-11-02 17:12:12,2013-07-18
16:42:36,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
```

If You Have More Time

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.

    - Hint: refer to the Spark API for more information on the `keyBy` operation

    - 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
…
```

This is the end of the Exercise

Hands-On Exercise: Using HDFS

Files Used in This Exercise:

Data files (local)

`~/training_materials/sparkdev/data/weblogs/*`

Solution:

`solutions/SparkHDFS.pyspark`

`solutions/SparkHDFS.scalaspark`

In this Exercise you will practice working with files in HDFS, the Hadoop Distributed File System.

Exploring HDFS

HDFS is already installed, configured, and running on your virtual machine.

1. Open a terminal window (if one is not already open) by double-clicking the Terminal icon on the desktop.

2. Most of your interaction with the system will be through a command-line wrapper called `hadoop`. If you run this program with no arguments, it prints a help message. To try this, run the following command in a terminal window:

```
$ hdfs
```

3. The `hdfs` command is subdivided into several subsystems. The subsystem for working with the files on the file is called `FsShell`. This subsystem can be invoked with the command `hdfs dfs`. In the terminal window, enter:

```
$ hdfs dfs
```

   You see a help message describing all the commands associated with the `FsShell` subsystem.

4. Enter:

```
$ hdfs dfs -ls /
```

This shows you the contents of the root directory in HDFS. There will be multiple entries, one of which is `/user`. Individual users have a "home" directory under this directory, named after their username; your username in this course is `training`, therefore your home directory is `/user/training`.

5. Try viewing the contents of the `/user` directory by running:

```
$ hdfs dfs -ls /user
```

You will see your home directory in the directory listing.

6. List the contents of your home directory by running:

```
$ hdfs dfs -ls /user/training
```

There are no files yet, so the command silently exits. This is different than if you ran `hdfs dfs -ls /foo`, which refers to a directory that doesn't exist and which would display an error message.

Note that the directory structure in HDFS has nothing to do with the directory structure of the local filesystem; they are completely separate namespaces.

Uploading Files

Besides browsing the existing filesystem, another important thing you can do with `FsShell` is to upload new data into HDFS.

7. Change directories to the local filesystem directory containing the sample data for the course.

```
$ cd ~/training_materials/sparkdev/data
```

If you perform a regular Linux `ls` command in this directory, you will see a few files, including the weblogs directory you used in previous exercises.

8. Insert this directory into HDFS:

```
$ hdfs dfs -put weblogs /user/training/weblogs
```

This copies the local `weblogs` directory and its contents into a remote HDFS directory named `/user/training/weblogs`.

9. List the contents of your HDFS home directory now:

```
$ hdfs dfs -ls /user/training
```

You should see an entry for the `weblogs` directory.

10. Now try the same `dfs -ls` command but without a path argument:

```
$ hdfs dfs -ls
```

You should see the same results. If you do not pass a directory name to the `-ls` command, it assumes you mean your home directory, i.e. `/user/training`.

> **Relative paths**
>
> If you pass any relative (non-absolute) paths to `FsShell` commands, they are considered relative to your home directory.

Viewing and Manipulating Files

Now view some of the data you just copied into HDFS.

11. Enter:

```
$ hdfs dfs -cat weblogs/2014-03-08.log | tail -n 50
```

This prints the last 50 lines of the file to your terminal. This command is useful for viewing the output of Spark programs. Often, an individual output file is very large, making it inconvenient to view the entire file in the terminal. For this reason, it is often a good idea to pipe the output of the `fs -cat` command into `head`, `tail`, `more`, or `less`.

12. To download a file to work with on the local filesystem use the `dfs -get` command. This command takes two arguments: an HDFS path and a local path. It copies the HDFS contents into the local filesystem:

```
$ hdfs dfs -get weblogs/2013-09-22.log ~/logfile.txt
$ less ~/logfile.txt
```

13. There are several other operations available with the `hdfs dfs` command to perform most common filesystem manipulations: `mv`, `rm`, `cp`, `mkdir`, and so on. Enter:

```
$ hdfs dfs
```

This displays a brief usage report of the commands available within `FsShell`. Try playing around with a few of these commands.

Accessing HDFS files in Spark

14. In the Spark Shell, create an RDD based on one of the files you uploaded to HDFS. For example:

```
pyspark> logs=sc.textFile("hdfs://localhost/\
user/training/weblogs/2014-03-08.log")
```

15. Save the JPG requests in the dataset to HDFS:

```
pyspark> logs.filter(lambda s: ".jpg" in s).\
 saveAsTextFile("hdfs://localhost/user/training/jpgs")
```

16. View the created directory and files it contains.

```
$ hdfs dfs -ls jpgs
$ hdfs dfs -cat jpgs/* | more
```

17. *Optional*: Explore the NameNode UI: `http://localhost:50070`. In particular, try menu selection **Utilities → Browse the Filesystem**.

This is the end of the Exercise

Hands-On Exercise: Running Spark Shell on a Cluster

In this Exercise you will start the Spark Standalone master and worker daemons, explore the Spark Master and Spark Worker User Interfaces (UIs), and start the Spark Shell on the cluster.

Note that in this course you are running a "cluster" on a single host. This would never happen in a production environment, but is useful for exploration, testing, and practicing.

Start the Spark Standalone Cluster

1. In a terminal window, start the Spark Master and Spark Worker daemons:

```
$ sudo service spark-master start
$ sudo service spark-worker start
```

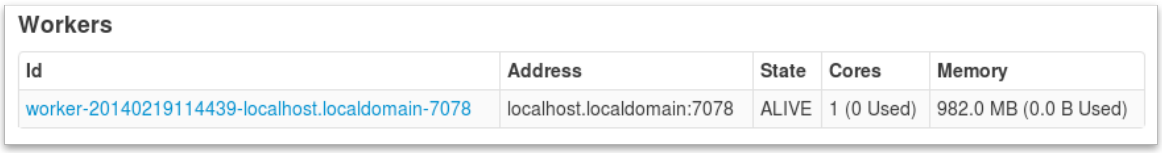
   Note: You can stop the services by replacing `start` with `stop`, or force the service to restart by using `restart`. You may need to do this if you suspend and restart the VM.

View the Spark Standalone Cluster UI

2. Start Firefox on your VM and visit the Spark Master UI by using the provided bookmark or visiting `http://localhost:18080/`.

3. You should not see any applications in the Running Applications or Completed Applications areas because you have not run any applications on the cluster yet.

4. A real-world Spark cluster would have several workers configured. In this class we have just one, running locally, which is named by the date it started, the host it is running on, and the port it is listening on. For example:


Workers

Id	Address	State	Cores	Memory
worker-20140219114439-localhost.localdomain-7078	localhost.localdomain:7078	ALIVE	1 (0 Used)	982.0 MB (0.0 B Used)

5. Click on the worker ID link to view the Spark Worker UI and note that there are no executors currently running on the node.

6. Return to the Spark Master UI and take note of the URL shown at the top. You may wish to select and copy it into your clipboard.

Start Spark Shell on the cluster

7. Return to your terminal window and exit Spark Shell if it is still running.

8. Start Spark Shell again, this time setting the MASTER environment variable to the Master URL you noted in the Spark Standalone Web UI. For example, to start pyspark:

```
$ MASTER=spark://localhost:7077 pyspark
```

Or the Scala shell:

```
$ spark-shell --master spark://localhost:7077
```

You will see additional info messages confirming registration with the Spark Master. (You may need to hit Enter a few times to clear the screen log and see the shell prompt.) For example:

```
…INFO cluster.SparkDeploySchedulerBackend: Connected to
Spark cluster with app ID app-20140604052124-0017
…INFO client.AppClient$ClientActor: Executor added:
app-20140604052124-0017/0 on worker-20140603111601-
localhost-7078 (localhost:7078) with 1 cores
```

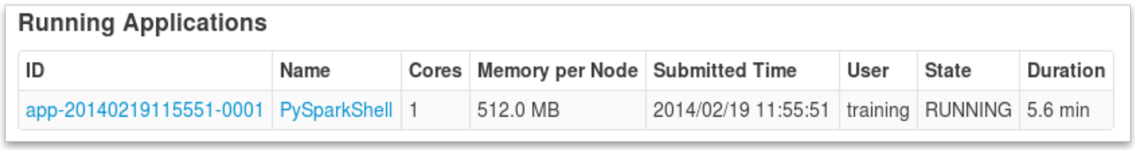
9. You can confirm that you are connected to the correct master by viewing the `sc.master` property:

```
pyspark> sc.master
```

10. Execute a simple operation to test execution on the cluster. For example,

```
pyspark> sc.textFile("weblogs/*").count()
```

11. Reload the Spark Standalone Master UI in Firefox and note that now the Spark Shell appears in the list of running applications.



Running Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20140219115551-0001	PySparkShell	1	512.0 MB	2014/02/19 11:55:51	training	RUNNING	5.6 min

12. Click on the application ID (`app-`*`xxxxxxx`*) to see an overview of the application, including the list of executors running (or waiting to run) tasks from this application. In our small classroom cluster, there is just one, running on the single node in the cluster, but in a real cluster there could be multiple executors running on each cluster.

This is the end of the Exercise

Hands-On Exercise: Working With Partitions

Files Used in This Exercise:

Data files (local):

`~/training_materials/sparkdev/data/activations`

Stubs:

`stubs/TopModels.pyspark`

`stubs/TopModels.scalaspark`

Solutions:

`solutions/TopModels.pyspark`

`solutions/TopModels.scalaspark`

In this Exercise you will find the most common type of cellular device activated in a given data set.

As you work through the exercise, you will explore RDD partitioning.

The Data

Review the data in `~/training_materials/sparkdev/data/activations`. Each XML file contains data for all the devices activated by customers during a specific month.

1. Copy this data to 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 output the top *n* device models activated.

The output will look something like:

```
iFruit 1 (392)
Sorrento F00L (224)
MeeToo 1.0 (12)
```

1. Start with the `TopModels` stub script. 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. Read the XML files into an RDD, then call `toDebugString` on that RDD. This will display the number of partitions, which will be the same as the number of files that were read:

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

This will display the lineage (the list of dependent RDDs; this will be discussed more in the next chapter). The one you care about here is the current RDD, which is at the top of the list.

3. Use `mapPartitions` to map each partition to an XML Tree structure based on parsing the contents of that partition as a string. You can call the provided function `getactivations` by passing it the partition iterator from `mapPartitions`; it will return an array of XML elements for each activation tag in the partition. For example:

```
pyspark> activations.mapPartitions(lambda xml: \
    getactivations(xml))
```

4. Map each activation tag to the model name of the device activated using the provided `getmodel` function.

5. Call `toDebugString` on the new RDD and note that the partitioning has been maintained: one partition for each file.

6. Count the number of occurrences of each model and display the top few. (Refer to earlier chapters for a reminder on how to use MapReduce to count occurrences if you need to.)

7. Use the `top(`*n*`)` method to display the 10 most popular models. Note that you will need to key the RDD by count.

Note: Leave the Spark Shell running for the next exercise.

This is the end of the Exercise

Hands-On Exercise: Viewing Stages in the Spark Application UI

Files Used in This Exercise:

Data files (HDFS):

`activations/*`

Solutions:

`solutions/TopModels.pyspark`

`solutions/TopModels.scalaspark`

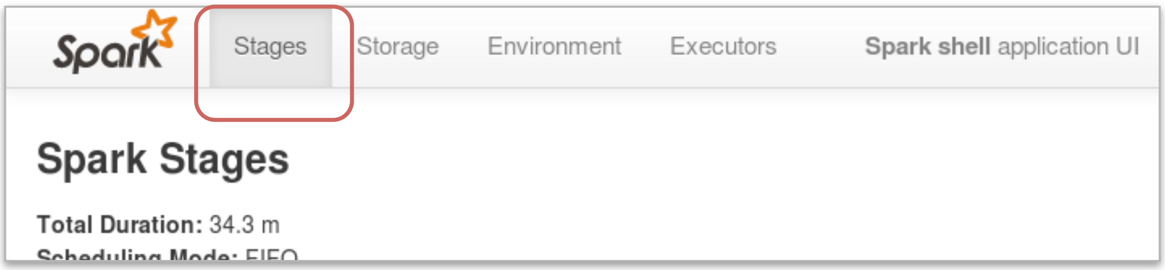
In this Exercise you will use the Spark Application UI to view the execution stages for a job.

In the last Exercise, you wrote a script in the Spark Shell to parse XML files containing device activation data, and count the number of each type of device model activated. Now you will review the stages and tasks that were executed.

1. Make sure the Spark Shell is still running from the last Exercise. If it is not, or if you did not complete the last Exercise, restart the shell and paste in the code from the solution file for the previous Exercise.

2. In a browser, view the Spark Application UI: `http://localhost:4040/`

3. Make sure the Stages tab is selected.



Spark Stages

Total Duration: 34.3 m

Scheduling Mode: FIFO

4. Look in the Completed Stages section and you should see the stages of the exercise you completed.

   Things to note:

   a. The stages are numbered, but numbers do not relate to the order of execution. Note the times the stages were submitted.

   b. The number of tasks in the first stage corresponds to the number of partitions, which for this example corresponds to the number of files processed.

   c. The Shuffle Write column indicates how much data that stage copied between tasks. This is useful to know because copying too much data across the network can cause performance issues.

5. 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 which ran the tasks. (In this single-node cluster, all tasks run on the same host.)

Note: Leave the Spark Shell running for the next exercise.

This is the end of the Exercise

Hands-On Exercise: Caching RDDs

Files Used in This Exercise:

Data files (HDFS):

`activations/*`

Solutions:

`solutions/TopModels.pyspark`

`solutions/TopModels.scalaspark`

In this Exercise you will explore the performance effect of caching an RDD.

The easiest way to see caching in action is to compare the time it takes to complete an operation on a cached and uncached RDD.

1. Make sure the Spark Shell is still running from the last exercise. If it isn't, restart it and paste in the code from the solution file.

2. Execute a count action on the RDD containing the list of activated models:

```
pyspark> models.count()
```

3. Take note of the time it took to complete the `count` operation. The output will look something like this:

```
14/04/07 05:47:17 INFO SparkContext: Job finished:
count at <ipython-input-3-986fd9b5da19>:1, took
17.718823392 s
```

4. Now cache the RDD:

```
pyspark> models.cache()
```

5. Execute the count again. This time should take about the same amount of time the last one did. It may even take a little longer, because in addition to running the operation, it is also caching the results.

6. Re-execute the count. Because the data is now cached, you should see a substantial reduction in the amount of time the operation takes.

7. In your browser, view the Spark Application UI and select the **Storage** tab. You will see a list of cached RDDs (in this case, just the models RDD you cached above). Click the RDD description to see details about partitions and caching.

8. Click on the **Executors** tab and take note of the amount of memory used and available for our 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 both the compute times and 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.)

This is the end of the Exercise

Hands-On Exercise: Checkpointing RDDs

Files Used in This Exercise:

Stubs:

`stubs/IterationTest.pyspark`

`stubs/IterationTest.scalaspark`

Solutions:

`solutions/IterationTest.pyspark`

`solutions/IterationTest.scalaspark`

In this Exercise you will see how checkpointing affects an RDD's lineage

Create an iterative RDD that results in a stack overflow

1. Create an RDD by parallelizing an array of numbers.

```
pyspark> mydata = sc.parallelize([1,2,3,4,5])
```

2. Loop 200 times. Each time through the loop, create a new RDD based on the previous iteration's result by adding 1 to each element.

```
pyspark> for i in range(200):
    mydata = mydata.map(lambda myInt: myInt + 1)
```

3. Collect and display the data in the RDD.

```
pyspark> for x in mydata.collect(): print x
```

4. Show the final RDD using `toDebugString()`. Note that the base RDD of the lineage is the parallelized array, e.g. `ParallelCollectionRDD[1]`

5. Repeat the loop, which adds another 200 iterations to the lineage. Then collect and display the RDD elements again. Did it work?

> Tip: When an exception occurs, the application output may exceed your terminal window's scroll buffer. You can adjust the size of the scroll buffer by selecting **Edit** > **Profile Preferences** > **Scrolling** and changing the Scrollback lines setting.

6. Keep adding to the lineage by repeating the loop, and testing by collecting the elements. Eventually, the `collect()` operation should give you an error indicating a stack overflow.

   - In Python, the error message will likely report "excessively deep recursion required".

   - In Scala the base exception will be StackOverflowError, which you will have to scroll up in your shell window to see; the immediate exception will probably be related to the Block Manager thread not responding, such as "Error sending message to BlockManagerMaster".

7. Take note of the total number of iterations that resulted in the stack overflow.

Fix the stack overflow problem by checkpointing the RDD

8. Exit and restart the Spark Shell following the error in the previous section.

9. Enable checkpointing by calling `sc.setCheckpointDir("checkpoints")`

10. Paste in the previous code to create the RDD.

11. As before, create an iterative RDD dependency, using at least the number of iterations that previously resulted in stack overflow.

12. Inside the loop, add two steps that are executed every 10 iterations:

    a. Checkpoint the RDD

   b. Materialize the RDD by performing an action such as `count`.

13. After looping, collect and view the elements of the RDD to confirm the job executes without a stack overflow.

14. Examine the lineage of the RDD; note that rather than going back to the base, it goes back to the most recent checkpoint.

This is the end of the Exercise

Hands-On Exercise: Writing and Running a Spark Application

Files and Directories Used in This Exercise:

Data files (HDFS)

`/user/training/weblogs`

Scala Project Directory:

`~/exercises/projects/countjpgs`

Scala Classes:

`stubs.CountJPGs`

`solution.CountJPGs`

Python Stub:

`stubs/CountJPGs.py`

Python Solution:

`solutions/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 "Getting Started With RDDs" 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: `~/training_materials/sparkdev/stubs/CountJPGs.py`. This stub imports the required Spark class and sets up your main code block. Copy this stub to your work area and edit it to complete this exercise.

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 weblogs/*
```

5. By default, the program will run locally. Re-run the program, specifying the cluster master in order to run it on the cluster:

```
$ spark-submit --master spark://localhost:7077 \
CountJPGs.py weblogs/*
```

6. Visit the Standalone Spark Master UI and confirm that the program is running on the cluster.

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 are familiar with the Idea IntelliJ IDE, you may choose to use that; the provided project directories include IntelliJ configuration.

1. A Maven project to get started has been provided: `~/exercises/projects/countjpgs.`

2. Edit the Scala code in `src/main/scala/stubs/CountJPGs.scala`.

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 "Getting Started with RDDs" exercise for the code to do this.

5. From the `countjpgs` working 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 weblogs/*
```

7. By default, the program will run locally. Re-run the program, specifying the cluster master in order to run it on the cluster:

```
$ spark-submit \
--class stubs.CountJPGs \
--master spark://localhost:7077 \
target/countjpgs-1.0.jar weblogs/*
```

8. Visit the Standalone Spark Master UI and confirm that the program is running on the cluster.

This is the end of the Exercise

Hands-On Exercise: Configuring Spark Applications

Files Used in This Exercise:

Data files (HDFS)

`/user/training/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 spark://localhost:7077 \
--name 'Count JPGs' \
CountJPGs.py weblogs/*
```

2. Visit the Standalone Spark Master UI (`http://localhost:18080/`) and note the application name listed is the one specified in the command line.

3. *Optional*: While the application is running, visit the Spark Application UI and 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:

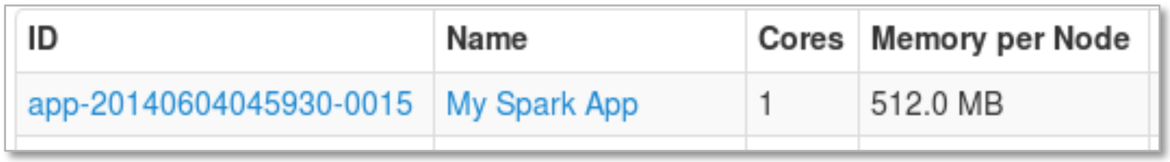
```
spark.app.name  My Spark App
spark.ui.port    4141
spark.master     spark://localhost:7077
```

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 \
weblogs/*
```

7. While the application is running, view the Spark Application UI at the alternate port you specified to confirm that it is using the correct port:
`http://localhost:`**`4141`**

8. Also visit the Standalone Spark Master UI to confirm that the application correctly ran on the cluster with the correct app name, e.g.:



ID	Name	Cores	Memory per Node
app-20140604045930-0015	My Spark App	1	512.0 MB

Optional: Set configuration properties programmatically

9. Following the example from the slides, modify the CountJPGs program to set the application name and UI port programmatically.

a. First create a SparkConf object and set its `spark.app.name` and `spark.ui.port` properties

b. Then use the SparkConf object when creating the SparkContext.

Set logging levels

10. Copy the template file `$SPARK_HOME/conf/log4j.properties.template` to `log4j.properties` in your exercise working directory.

11. Edit `log4j.properties`. The first line currently reads:

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

Replace `INFO` with `DEBUG`:

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

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

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

```
14/03/19 11:40:45 INFO MemoryStore: ensureFreeSpace(154293) called
with curMem=0, maxMem=311387750
14/03/19 11:40:45 INFO MemoryStore: Block broadcast_0 stored as
values to memory (estimated size 150.7 KB, free 296.8 MB)
14/03/19 11:40:45 DEBUG BlockManager: Put block broadcast_0 locally
took  79 ms
14/03/19 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.

14. 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.

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

This is the end of the Exercise

Hands-On Exercise: Exploring Spark Streaming

Files Used in This Exercise:

Solution Scala script:

`solutions/SparkStreaming.scalaspark`

In this Exercise, you will explore Spark Streaming using the Scala Spark Shell.

This exercises has two parts:

- Review the Spark Streaming documentation
- A series of step-by-step instructions in which you will use Spark Streaming to count words in a stream

Review the Spark Streaming documentation

1. View the Spark Streaming API by visiting the Spark Scaladoc API (which you bookmarked previously in the class) and selecting the `org.apache.spark.streaming` package in the package pane on the left.

2. Follow the links at the top of the package page to view the `DStream` and `PairDStreamFunctions` classes – these will show you the functions available on a DStream of regular RDDs and Pair RDDs respectively.

3. You may also wish to view the Spark Streaming Programming Guide (select **Programming Guides** > **Spark Streaming** on the Spark documentation main page).

Count words in a stream

For this section, you will simulate streaming text data through a network socket using the `nc` command. This command takes input from the console (stdin) and sends it to the port you specify, so that the text you type is sent to the client program (which will be your Spark Streaming application.)

4. In a terminal window, enter the command

```
 $ nc -lkv 1234
```

   Anything you type will be sent to port 1234. You will return to this window after you have started your Spark Streaming Context.

5. Start a separate terminal for running the Spark Shell. Copy `/usr/lib/spark/conf/log4j.properties` to the local directory and edit it to set the logging level to `ERROR`. (This is to reduce the level of logging output, which otherwise would make it difficult to see the interactive output from the streaming job.)

6. Start the Spark Scala Shell. In order to use Spark Streaming interactively, you need to either run the shell on a Spark cluster, or locally with at least two threads. For this exercise, run locally with two threads, by typing:

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

7. In the Spark Shell, import the classes you need for this example. You may copy the commands from these instructions, or if you prefer, copy from the solution script file provided (`SparkStreaming.scalaspark`).

```
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.Seconds
```

8. Create a Spark Streaming Context, starting with the Spark Context provided by the shell, with a batch duration of 5 seconds:

```
val ssc = new StreamingContext(sc,Seconds(5))
```

9. Create a DStream to read text data from port 1234 (the same port you are sending text to using the nc command, started in the first step.)

```
val mystream = ssc.socketTextStream("localhost",1234)
```

10. Use MapReduce to count the occurrence of words on the stream.

```
val words = mystream.flatMap(line => line.split("\\W"))
val wordCounts = words.map(x =>
   (x, 1)).reduceByKey((x,y) => x+y)
```

11. Print out the first 10 word count pairs in each batch:

```
wordCounts.print()
```

12. Start the Streaming Context. This will trigger the DStream to connect to the socket, and start batching and processing the input every 5 seconds. Call `awaitTermination` to wait for the task to complete.

```
ssc.start()
ssc.awaitTermination()
```

13. Go back to the terminal window in which you started the `nc` command. You should see a message indicating that `nc` accepted a connection from your DStream.

14. Enter some text. Every five seconds you should see output in the Spark Shell window such as:

```
-------------------------------------------
Time: 1396631265000 ms
-------------------------------------------
(never,1)
(purple,1)
(I,1)
(a,1)
(ve,1)
(seen,1)
(cow,1)
```

15. To restart the application, type Ctrl-C to exit Spark Shell, then restart the shell and use command history or paste in the application commands again.

16. When you are done, close the `nc` process in the other terminal window.

Hands-On Exercise: Writing a Spark Streaming Application

Files and Directories Used in This Exercise:

Project:

`~/exercises/projects/streaminglogs`

Stub class:

`stubs.StreamingLogs`

Solution class:

`solution.StreamingLogs`

Test script:

`~/training_materials/sparkdev/examples/streamtest.py`

In this Exercise, you will write a Spark Streaming application to count Knowledge Base article requests by User ID.

Count Knowledge Base article requests

Now that you are familiar with using Spark Streaming, try a more realistic task: read in streaming web server log data, and count the number of requests for Knowledge Base articles.

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

1. Stream the Loudacre weblog files at a rate of 20 per second. In a separate terminal window, run:

   ```
   $ python \
   ~/training_materials/sparkdev/examples/streamtest.py \
   localhost 1234 20 \
   /home/training/training_materials/sparkdev/data/weblogs/*
   ```

   Note that this script exits after the client disconnects; you will need to restart the script when you restart your Spark Application.

2. A Maven project folder has been provided for your Spark Streaming application: `exercises/projects/streaminglogs`. To complete the exercise, start with the stub code in `src/main/scala/stubs/StreamingLogs.scala`, which imports the necessary classes and sets up the Streaming Context.

3. Create a DStream by reading the data from the host and port provided as input parameters.

4. Filter the DStream to only include lines containing the string "`KBDOC`".

5. For each RDD in the filtered DStream, display the number of items – that is, the number of requests for KB articles.

6. Save the filtered logs to text files.

7. To test your application, build your application JAR file using the `mvn package` command. Run your application locally and be sure to specify two threads; at least two threads or nodes are required to running a streaming application, while our VM cluster has only one. The `StreamingLogs` application takes two parameters: the host name and port number to connect the DStream to; specify the same host and port that the test script is listening on.

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

(Use `--class` **`solution.`**`StreamingLogs` to run the solution class instead.)

8. Verify the count output, and review the contents of the files.

9. Challenge: In addition to displaying the count every second (the batch duration), count the number of KB requests over a window of 10 seconds. Print out the updated 10 second total every 2 seconds.

    a. Hint 1: Use the `countByWindow` function.

    b. Hint 2: Use of window operations requires checkpointing. Use the `ssc.checkpoint(`*`directory`*`)` function before starting the SSC to enable checkpointing.

This is the end of the Exercise

Hands-On Exercise: Iterative Processing with Spark

Files Used in This Exercise:

Data files (local):

`~/training_materials/sparkdev/data/devicestatus.txt`

Stubs:

`stubs/KMeansCoords.pyspark`

`stubs/KMeansCoords.scalaspark`

Solutions:

`solutions/KMeansCoords.pyspark`

`solutions/KMeansCoords.scalaspark`

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

Review the Data

Review the data file, then copy it to HDFS:
`~/training_materials/sparkdev/data/devicestatus.txt`.

This file contains a sample of device status data. For this exercise, the fields you care about are the last two, which represent the location (latitude and longitude) of the device as the last two fields (fields 13 and 14):

```
2014-03-15:13:10:20|Titanic 2500|15e758be-8624-46aa-80a3-
     b6e08e979600|77|70|40|22|13|0|enabled|connected|enabled|38.92539179
     59|-122.78959506
2014-03-15:13:10:20|Sorrento F41L|2d6862a6-2659-4e07-9c68-
     6ea31e94cda0|4|16|23|enabled|enabled|connected|33|79|44|35.48129955
     43|-120.306768128
```

Calculate k-means for device location

If you are already familiar with calculating k-means, try doing the exercise on your own. Otherwise, follow the step-by-step process below.

1. Start by copying the provided `KMeansCoords` stub file, which contains the following convenience functions used in calculating k-means:

    - `closestPoint`: given a (latitude/longitude) point and an array of current center points, returns the index in the array of the center closest to the given point

    - `addPoints`: given two points, return a point which is the sum of the two points – that is, (x1+x2, y1+y2)

    - `distanceSquared`: given two points, returns the squared distance of the two. This is a common calculation required in graph analysis.

2. Set the variable `K` (the number of means to calculate). For this exercise we recommend you start with `5`.

3. 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. We recommend starting with a value of `0.1`.

4. Parse the input file, which is delimited by the character '`|`', into (latitude,longitude) pairs (the 13th and 14th fields in each line). Only include known locations (that is, filter out `(0,0)` locations). Be sure to cache the resulting RDD because you will access it each time through the iteration.

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

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

6. 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.) E.g.

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

   b. Reduce the result: for each center in the kPoints array, sum the latitudes and longitudes, respectively, of all the points closest to that center, and the number of closest points. E.g.

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

   c. The reduced RDD should have (at most) K members. Map each to a new center point by calculating the average latitude and longitude for each set of closest points: that is, map (*index*,(*totalX*,*totalY*),*n*) → (*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.

7. When the iteration is complete, display the final K center points.

This is the end of the Exercise

Hands-On Exercise: Using Broadcast Variables

Files Used in This Exercise:

Data files (HDFS):

`weblogs/*`

Data files (local):

`~training_materials/sparkdev/data/targetmodels.txt`

Stubs:

`stubs/TargetModels.pyspark`

`stubs/TargetModels.scalaspark`

Solutions:

`solutions/TargetModels.pyspark`

`solutions/TargetModels.scalaspark`

In this Exercise you will filter web requests to include only those from devices included in a list of target models.

Loudacre wants to do some analysis on web traffic produced from specific devices. The list of target models is in `~training_materials/sparkdev/data/targetmodels.txt`

Filter the web server logs to include only those requests from devices in the list. (The model name of the device will be in the line in the log file.) Use a broadcast variable to pass the list of target devices to the workers that will run the filter tasks.

Hint: Use the stub file for this exercise in `~/training_materials/sparkdev/stubs` for the code to load in the list of target models.

This is the end of the Exercise

Hands-On Exercise: Using Accumulators

Files Used in This Exercise:

Data files (HDFS):

`weblogs/*`

Solutions:

`RequestAccumulator.pyspark`

`RequestAccumulator.scalaspark`

In this Exercise you will count the number of different types of files requested in a set of web server logs.

Using accumulators, count the number of each type of file (HTML, CSS and JPG) requested in the web server log files.

Hint: use the file extension string to determine the type of request,
i.e. `.html`,`.css`, `.jpg`.

This is the end of the Exercise

Hands-On Exercise: Importing Data With Sqoop

Files Used in This Exercise:

Solutions:

`solutions/sqoop-movie-import.sh`

`solutions/AverageMovieRatings.pyspark`

`solutions/AverageMovieRatings.scalaspark`

In this Exercise you will import data from a relational database using Sqoop. The data you load here will be used in subsequent exercises.

Consider the MySQL database `movielens`, derived from the MovieLens project from University of Minnesota. (See note at the end of this Exercise.) The database consists of several related tables, but we will import only two of these: `movie`, which contains information on about 3,900 movies; and `movierating`, which has about 1,000,000 ratings of those movies.

Review the Database Tables

First, review the database tables to be loaded into Hadoop.

1. Log in to MySQL:

```
$ mysql --user=training --password=training movielens
```

2. Review the structure and contents of the `movie` table:

```
mysql> DESCRIBE movie;
. . .
mysql> SELECT * FROM movie LIMIT 5;
```

3. Note the column names for the table:

____

4. Review the structure and contents of the `movierating` table:

```
mysql> DESCRIBE movierating;
…
mysql> SELECT * FROM movierating LIMIT 5;
```

5. Note these column names:

____

6. Exit mysql:

```
mysql> quit
```

Import with Sqoop

You invoke Sqoop on the command line to perform several commands. With it you can connect to your database server to list the databases (schemas) to which you have access, and list the tables available for loading. For database access, you provide a connection string to identify the server, and – if required – your username and password.

1. Show the commands available in Sqoop:

```
$ sqoop help
```

2. List the databases (schemas) in your database server:

```
$ sqoop list-databases \
--connect jdbc:mysql://localhost \
--username training --password training
```

(Note: Instead of entering `--password training` on the command line, you may prefer to enter `-P`, and let Sqoop prompt you for the password, which is then not displayed when you type it.)

3. List the tables in the `movielens` database:

```
$ sqoop list-tables \
  --connect jdbc:mysql://localhost/movielens \
  --username training --password training
```

4. Import the `movie` table into HDFS:

```
$ sqoop import \
  --connect jdbc:mysql://localhost/movielens \
  --username training --password training \
  --fields-terminated-by '\t' --table movie
```

5. Verify that the command has worked. Note that like Spark output, Sqoop output is stored in multiple partition files rather than a single file. Take note of the format of the file: `movieID[tab]name[tab]year`

```
$ hdfs dfs -ls movie
$ hdfs dfs -tail movie/part-m-00000
```

6. Import the `movierating` table into HDFS by repeating the last two steps, but for the `movierating` table.

Read and process the data in Spark

7. Start the Spark Shell.

8. Read in all the movie ratings, keyed by movie ID. (Split the input line on the tab character: `\t`)

9. Challenge: calculate the average rating for each movie

10. Challenge: Save the average ratings to files in the format:
`movieID[tab]name[tab]rating`
(Hint: join with the data from the movie table)

This is the end of the Exercise

Note:

This exercise uses the MovieLens data set, or subsets thereof. This data is freely available for academic purposes, and is used and distributed by Cloudera with the express permission of the UMN GroupLens Research Group. If you would like to use this data for your own research purposes, you are free to do so, as long as you cite the GroupLens Research Group in any resulting publications. If you would like to use this data for commercial purposes, you must obtain explicit permission. You may find the full dataset, as well as detailed license terms, at http://www.grouplens.org/node/73