Ask Bigger Questions

Developer Training for Spark and Hadoop I: Hands-On Exercises

General Notes

Cloudera's training courses use a Virtual Machine running the CentOS Linux distribution. This VM has CDH (Cloudera's Distribution, including Apache Hadoop) installed in Pseudo-Distributed mode. Pseudo-Distributed mode is a method of running Hadoop whereby all Hadoop daemons run on the same machine. It is, essentially, a cluster consisting of a single machine. It works just like a larger Hadoop cluster, the only key difference (apart from speed, of course!) being that the block replication factor is set to 1, since there is only a single DataNode available.

Getting Started

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

```
$ $DEV1/scripts/training_setup_dev1.sh
```

This script will enable services and set up any data required for the course. You must run this script before starting the Hands-On Exercises.

Working with the Virtual Machine

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

2. Should you need it, the root password is `training`. You may be prompted for this if, for example, you want to change the keyboard layout. In general, you should not need this password since the `training` user has unlimited sudo privileges.

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

```
$ hdfs dfs -put shakespeare \
/user/training/shakespeare
```

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

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

Points to note during the exercises

1. The main directory for the exercises for this course is `$DEV1/exercises` (`~/training_materials/dev1/exercises`). Each directory under that one corresponds to an exercise or set of exercises – this is referred to in the instructions as "the exercise directory". Any scripts or files required for the exercise (other than data) are in the exercise directory.

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

   a. `solution` – Solution code for each exercise.

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

   c. Maven project directories – For exercises for which you must write Scala classes, you have been provided with preconfigured Maven project directories. Within these projects are two packages: `stubs`, where you will do your work using starter skeleton classes; and `solution`, containing the solution class.

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

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

5. There are additional bonus exercises for some of the exercises. If you finish the main exercise, please attempt the additional steps.

Catching Up

Most of the exercises in this course build on prior exercises. If you are unable to complete an exercise (even using the provided solution files), or if you need to catch up to the current exercise, run the provided catch up script:

```
$ $DEV1/scripts/catchup.sh
```

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

Warning: If you run the catch up script, you will lose all your work! (e.g. All data will be deleted from HDFS, tables deleted from Hive, etc.)

Hands-On Exercise: Write and Run a Spark Application

Files and Data Used in This Exercise:

Exercise Directory: `$DEV1/exercises/spark-application`

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

Scala Project: `countjpgs`

Scala Classes:

`stubs.CountJPGs`

`solution.CountJPGs`

Python Stub:

`CountJPGs.py`

Python Solution:

`$DEV1/exercises/spark-application/python-solution/CountJPGs.py`

In this Exercise you will write your own Spark application instead of using the interactive Spark Shell application.

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

This is the same task you did earlier in the "Use RDDs to Transform a Dataset" exercise. The logic is the same, but this time you will need to set up the SparkContext object yourself.

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

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

Write a Spark application in Python

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

1. A simple stub file to get started has been provided in the exercise project: `$DEV1/exercises/spark-application/CountJPGs.py`. This stub imports the required Spark class and sets up your main code block.

2. Set up a SparkContext using the following code:

```
sc = SparkContext()
```

3. In the body of the program, load the file passed in to the program, count the number of JPG requests, and display the count. You may wish to refer back to the "Getting Started with RDDs" exercise for the code to do this.

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

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

Write a Spark application in Scala

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

A Maven project to get started has been provided: `$DEV1/exercises/spark-application/countjpgs`.

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

```
$ sudo service archiva start
```

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

3. Set up a SparkContext using the following code:

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

4. In the body of the program, load the file passed in to the program, count the number of JPG requests, and display the count. You may wish to refer back to the "Use RDDs to Explore and Transform a Dataset" exercise for the code to do this.

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

```
$ mvn package
```

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

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

Submit a Spark application to the cluster

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

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

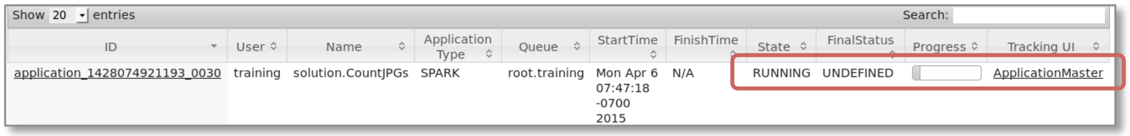
   To run Python:

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

   To run Scala:

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

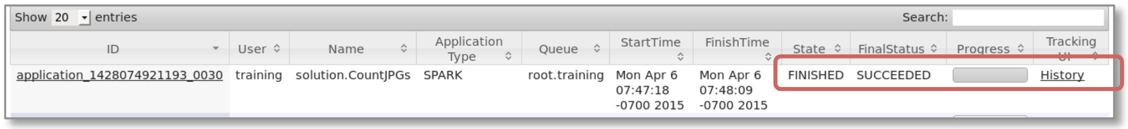
2. After starting the application, open Firefox and visit the YARN Resource Manager UI using the provided bookmark (or going to URL `http://localhost:8088`). While the application is running, it appears in the list of applications something like this:



Show 20 entries Search:

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

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



Show 20 entries Search:

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

3. Take note of the your application's ID (e.g. `application_12345…`). (You may wish to copy it.) In a terminal window, enter

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

This is the end of the Exercise

Hands-On Exercise: Configure a Spark Application

Files Used in This Exercise:

Exercise Directory: `$DEV1/exercises/spark-application`

Data files (HDFS)

`/loudacre/weblogs/*`

Properties files (local)

`spark.conf`

`log4j.properties`

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

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

Set configuration options at the command line

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

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

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

2. Visit the Resource Manager UI again and note the application name listed is the one specified in the command line.

3. Optional: View the Spark Application UI. From the RM application list, follow the ApplicationMaster link (if the application is still running) or the History link to visit the Spark Application UI. View the Environment tab. Take note of the `spark.*` properties such as `master`, `appName`, and `driver` properties.

Set configuration options in a configuration file

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

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

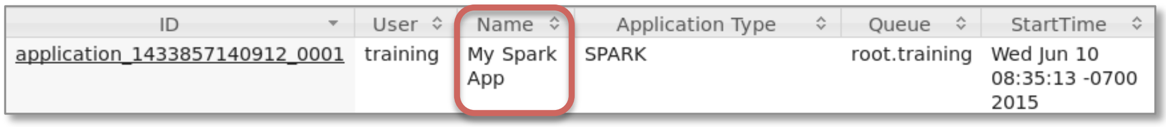
```
spark.app.name   My Spark App
spark.master     yarn-client
spark.executor.memory     400M
```

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

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

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

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



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

Set logging levels

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

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

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

Replace `INFO` with `DEBUG`:

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

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

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

```
15/03/19 11:40:45 INFO MemoryStore: ensureFreeSpace(154293) called
with curMem=0, maxMem=311387750
15/03/19 11:40:45 INFO MemoryStore: Block broadcast_0 stored as
values to memory (estimated size 150.7 KB, free 296.8 MB)
15/03/19 11:40:45 DEBUG BlockManager: Put block broadcast_0 locally
took  79 ms
15/03/19 11:40:45 DEBUG BlockManager: Put for block broadcast_0
without replication took  79 ms
```

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

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

13. You can also set the log level for the Spark Shell by placing the `log4j.properties` file in your working directory before starting the shell.

Try starting the shell from the directory in which you placed the file and note that only WARN messages now appear.

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

This is the end of the Exercise

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

Files and Data Used in This Exercise:

Exercise Directory: `$DEV1/exercises/spark-stages`

Data files (HDFS):

`/loudacre/weblogs/*`

`/loudacre/accounts/*`

Test Scripts:

`SparkStages.pyspark`

`SparkStages.scalaspark`

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

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

Explore Partitioning of file-based RDDs

1. Start (or restart, if necessary) the Spark Shell. Although you would typically run a Spark application on a cluster, your course VM cluster has only a single worker node that can support only a single executor. To simulate a more realistic multi-node cluster, run in local mode with 2 threads:

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

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

3. Create an RDD based on a *single file* in the dataset, e.g. `/loudacre/accounts/part-m-00000` and then call `toDebugString` on the RDD, which displays the number of partitions in parentheses `()` before the RDD id. How many partitions are in the resulting RDD?

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

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

4. Repeat this process, but specify a minimum of three of partitions: `sc.textFile(filename,3)`. Does the RDD correctly have three partitions?

5. Finally, create an RDD based on *all the files* in the accounts dataset. How does the number of files in the dataset compare to the number of partitions in the RDD?

6. Bonus: use `foreachPartition` to print out the first record of each partition.

Set up the job

7. Create an RDD of accounts, keyed by ID and with `first name, last name` for the value:

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

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

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

Tip: In this exercise you will be reducing and joining large datasets, which can take a lot of time running on a single machine, as you are using in the course. Therefore, rather than use all the web log files in the dataset, specify a subset of web log files using a wildcard, e.g. `textFile("/loudacre/weblogs/*6")`.

```
pyspark> userreqs = sc \
  .textFile("/loudacre/weblogs/*6") \
  .map(lambda line: line.split()) \
  .map(lambda words: (words[2],1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
```

```
scala> var userreqs = sc.
   textFile("/loudacre/weblogs/*6").
   map(line => line.split(' ')).
   map(words => (words(2),1)).
   reduceByKey((v1,v2) => v1 + v2)
```

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

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

```
scala> var accounthits =
  accountsByID.join(userreqs).map(pair => pair._2)
```

10. Print the results of `accounthits.toDebugString` and review the output. Based on this, see if you can determine

a. How many stages are in this job?

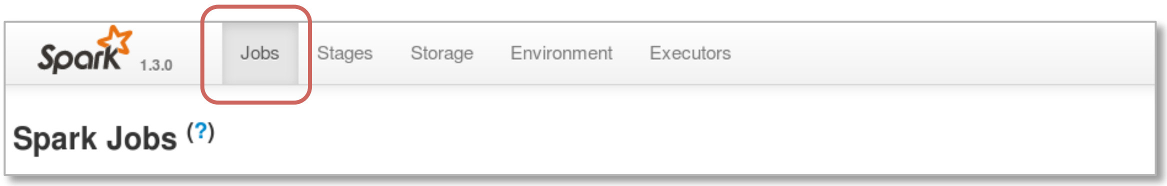
b. Which stages are dependent on which?

c. How many tasks will each stage consist of?

Run the job and review it in the Spark Application UI

11. In your browser, visiting the Spark Application UI by using the provided toolbar bookmark, or visiting URL `http://localhost:4040`.

12. In the Spark UI, make sure the **Jobs** tab is selected. No jobs are yet running so the list will be empty.

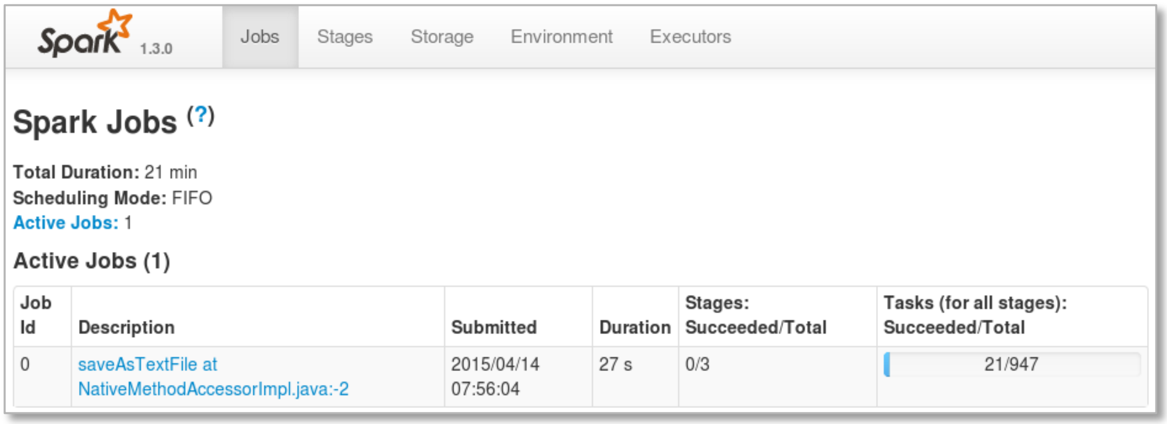


13. Return to the shell and start the job by executing an action (`saveAsTextFile`):

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

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

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



Spark Jobs (?)

Total Duration: 21 min
Scheduling Mode: FIFO
Active Jobs: 1

Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	saveAsTextFile at NativeMethodAccessorImpl.java:-2	2015/04/14 07:56:04	27 s	0/3	21/947

15. Click on the job description (which is the last action in the job) to see the stages. As the job progresses you may want to refresh the page a few times.

Things to note:

a. How many stages are in the job? Does it match the number you expected from the RDD's `toDebugString` output?

b. The stages are numbered, but numbers do not relate to the order of execution. Note the times the stages were submitted to determine the order. Does the order match what you expected based on RDD dependency?

c. How many tasks are in each stage? The number of tasks in the first stages correspond to the number of partitions, which for this example corresponds to the number of files processed.

d. The Shuffle Read and Shuffle Write columns indicate how much data was copied between tasks. This is useful to know because copying too much data across the network can cause performance issues.

2. Click on the stages to view details about that stage. Things to note:

    a. The Summary Metrics area shows you how much time was spend on various steps. This can help you narrow down performance problems.

    b. The Tasks area lists each task. The Locality Level column indicates whether the process ran on the same node where the partition was physically stored or not. Remember that Spark will attempt to always run tasks where the data is, but may not always be able to, if the node is busy.

    c. In a real-world cluster, the executor column in the Task area would display the different worker nodes that ran the tasks. (In this single-node cluster, all tasks run on the same host: localhost.)

3. When the job is complete, return to the **Jobs** tab to see the final statistics for the number of tasks executed and the time the job took.

4. *Optional*: Try re-running the last action. (You will need to either delete the `saveAsTextFile` output directory in HDFS, or specify a different directory name.) You will probably find that the job completes much faster, and that several stages (and the tasks in them) show as "skipped".

    Bonus question: Which tasks were skipped and why?

Leave the Spark Shell running for the next exercise.

This is the end of the Exercise

Hands-On Exercise: Persist an RDD

Files and Data Used in This Exercise:

Exercise Directory: `$DEV1/exercises/spark-persist`

Data files (HDFS):

`/loudacre/weblogs/*`

`/loudacre/accounts/*`

Job Setup Script:

`$DEV1/spark-stages/SparkStages.pyspark`

`$DEV1/spark-stages/SparkStages.scalaspark`

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

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

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

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

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

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

4. In your browser, view the Spark Application UI and select the **Storage** tab. At this point, you have marked your RDD to be persisted, but have not yet

performed an action that would cause it to be materialized and persisted, so you will not yet see any persisted RDDs.

5. In the Spark Shell, execute the count again.

6. View the RDD's `toDebugString`. Notice that the output indicates the persistence level selected.

7. Reload the Storage tab in your browser, and this time note that the RDD you persisted is shown. Click on the RDD ID to see details about partitions and persistence.

8. Click on the **Executors** tab and take note of the amount of memory used and available for your one worker node.

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

9. Optional: Set the RDD's persistence level to `StorageLevel.DISK_ONLY` and compare the storage report in the Spark Application Web UI. (Hint: Because you have already persisted the RDD at a different level, you will need to `unpersist()` first before you can set a new level.)

This is the end of the Exercise

Hands-On Exercise: Implement an Iterative Algorithm with Spark

Files and Data Used in This Exercise:

Exercise Directory: `$DEV1/exercises/spark-iterative`

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

Stubs:

`KMeansCoords.pyspark`

`KMeansCoords.scalaspark`

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

Review the Data

In the bonus section of the "Use RDDs to Explore and Transform a Dataset" exercise, you used Spark to extract the date, maker, device ID, latitude and longitude from the `devicestatus.txt` data file, and store the results in the HDFS directory `/loudacre/devicestatus_etl`.

If you did not have time to complete that bonus exercise, run the solution script now following the steps below. (If you have run the course catch-up script, this is already done for you.)

- Copy `$DEV1DATA/devicestatus.txt` to HDFS directory `/loudacre/`
- Run the Spark script `$DEV1/exercises/spark-etl/bonus/DeviceStatusETL` (either `.pyspark` or `.scalapark` depending on which language you are using)

Examine the data in the dataset. Note that the latitude and longitude are the 4th and 5th fields, respectively, e.g.

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

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 use K=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. For this exercise, use a value of `0.1`.

4. Parse the input file, which is delimited by the character '`,`', into `(latitude,longitude)` pairs (the 4th and 5th fields in each line). Only include known locations (that is, filter out (0,0) locations). Be sure to persist (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)` 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.

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

This is the end of the Exercise

Bonus Hands-On Exercise: Partition Data Files Using Spark

Files and Data Used in this Exercise

Exercise Directory: `$DEV1/exercises/spark-partfile`

Data files:

`/loudacre/devicestatus_etl/*` (HDFS)

`$DEV1DATA/status-regions.txt` (local)

Stubs: `partition-status.pyspark` / `.scalaspark`

Output directory (HDFS): `/loudacre/devstatusByRegion`

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

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

The Task

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

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

- Save the five data points you calculated above, or use the provided data file: `$DEV1DATA/status-regions.txt`, which is in the format:

```
region,latitude,longitude
region,latitude,longitude
...
```

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

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

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

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

This is the end of the Exercise

Hands-On Exercise: Use Spark SQL for ETL

Files and Data Used in this Exercise

Exercise Directory: `$DEV1/exercises/spark-sql`

MySQL table: `loudacre.webpage`

Output directory (HDFS): `/loudacre/webpage_files`

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

Review the Data in MySQL

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

1. List the columns and types in the table:

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

2. View the first few rows from the table:

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

Note that the data in the `associated_files` column is a comma-delimited string. Loudacre would like to make this data available in an Impala table, but in order to perform required analysis, the `associated_files` data must be extracted and normalized. Your goal in the next section is to use Spark SQL to extract the data in the column, split the string, and create a new dataset in HDFS containing each web page number, and its associated files in separate rows.

Load the Data from MySQL

3. If necessary, start the Spark Shell.

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

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

```
pyspark> from pyspark.sql import SQLContext
pyspark> sqlCtx = SQLContext(sc)
```

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

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

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

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

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

```
scala> val assocfiles =
webpages.select(webpages("web_page_num"),webpages("asso
ciated_files"))
```

```
python> assocfiles = \
  webpages.select(webpages.web_page_num,\
  webpages.associated_files)
```

8. In order to manipulate the data using Spark, convert the DataFrame into a to a Pair RDD using the map method. The input into the map method is a Row object. They key is the `web_page_num` value (the first value in the Row), and the value is the `associated_files` string (the second value in the Row).

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

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

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

```
pyspark> afilesrdd = assocfiles.map(lambda row: \
  (row.web_page_num,row.associated_files))
```

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

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

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

10. Create a new DataFrame from the RDD:

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

```
pyspark> afiledf = sqlCtx.createDataFrame(afilesrdd2)
```

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

12. Create a new DataFrame by renaming the columns to reflect the data they hold.

    In Scala, you can use the `toDF` shortcut method to create a new DataFrame based on an existing one with the columns renamed:

```
scala> val finaldf = afiledf.
  toDF("web_page_num","associated_file")
```

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

```
pyspark> finaldf = afiledf. \
  withColumnRenamed('_1','web_page_num'). \
  withColumnRenamed('_2','associated_file')
```

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

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

```
> finaldf.save("/loudacre/webpage_files")
```

View the Output

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

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

```
CREATE EXTERNAL TABLE webpage_files LIKE PARQUET
 '/loudacre/webpage_files/part-r-00001.parquet'
  STORED AS PARQUET
  LOCATION '/loudacre/webpage_files'
```

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

This is the end of the Exercise

Appendix A: Enabling iPython Notebook

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

1. Open the following file for editing: `/home/training/.bashrc`

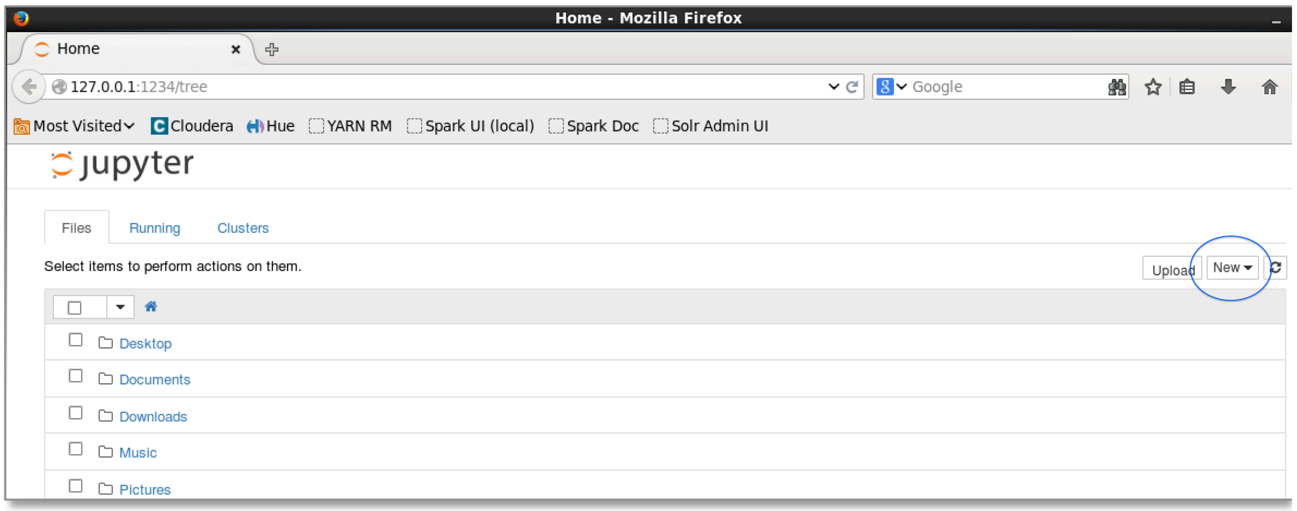
2. Uncomment out the following line (remove the leading #).

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

3. Save the file.

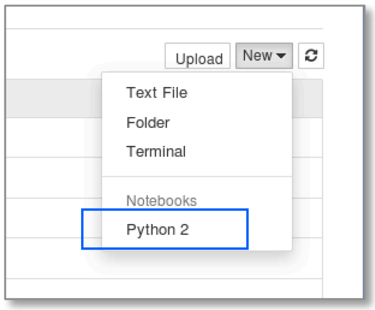
4. Open a new terminal window. (Must be a new terminal so it loads your edited `.bashrc` file).

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

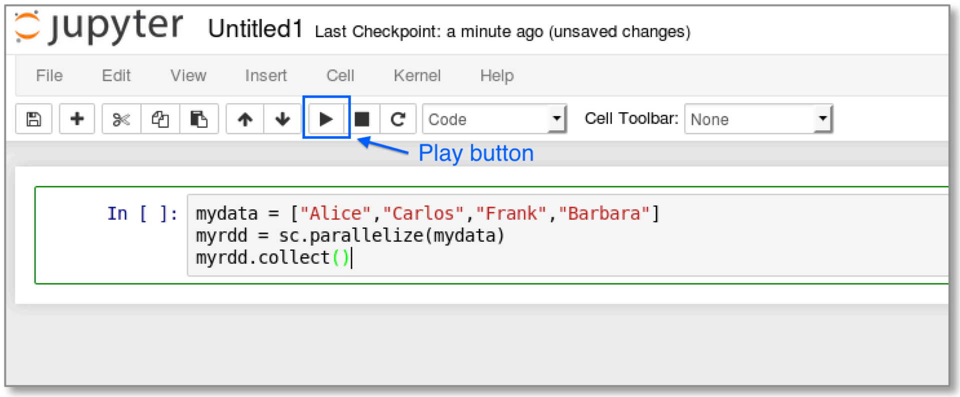


Select items to perform actions on them.

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



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

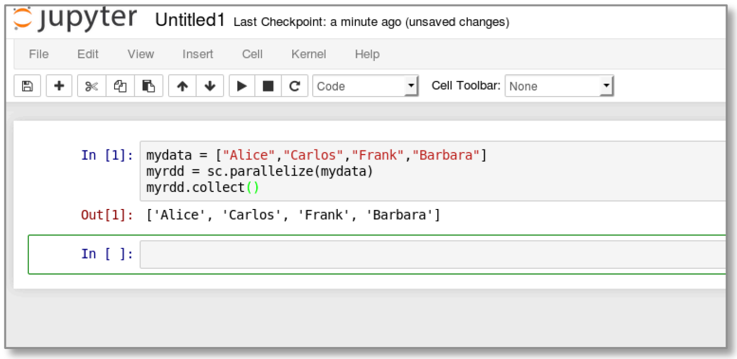


Last Checkpoint: a minute ago (unsaved changes)

Code   Cell Toolbar: None

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

8. Notice the output displayed.



Last Checkpoint: a minute ago (unsaved changes)

Code   Cell Toolbar: None

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

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

 In [ ]:
```