Developer Training for Spark and Hadoop: 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

2. 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`.

3. 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.

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

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

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

5. 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.

6. 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.

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

8. 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!

9. 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 previous exercises.

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

Hands-On Exercise: Access HDFS with Command Line and Hue

Files and Data Used in This Exercise:

Data files (local): `$DEV1DATA/kb/*`
`$DEV1DATA/base_stations.tsv`

In this exercise you will practice working with HDFS, the Hadoop Distributed File System.

You will use the HDFS command line tool and the Hue File Browser web-based interface to manipulate files in HDFS.

Setting Up Your Environment

1. Before starting the exercises, be sure you have run the course setup script in a terminal window. (You only need to run this script once; if you ran it earlier, you do not need to run it again.)

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

Exploring the HDFS Command Line Interface

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

The simplest way to interact with HDFS is by using the `hdfs` command. To execute file system commands within HDFS, use the `hdfs dfs` command.

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

3. 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`.

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

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

You will see your home directory in the directory listing.

5. 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 to HDFS

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

6. Start by creating a new top-level directory for exercises. You will use this directory throughout the rest of the course.

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

7. Change directories to the local filesystem directory containing the sample data we will be using in the course.

```
$ cd $DEV1DATA
```

If you perform a regular Linux `ls` command in this directory, you will see several files and directories used in this class. One of the data directories is `kb`. This directory holds Knowledge Base articles that are part of Loudacre's customer service website.

8. Insert this directory into HDFS:

```
$ hdfs dfs -put kb /loudacre/
```

This copies the local `kb` directory and its contents into a remote HDFS directory named `/loudacre/kb`.

9. List the contents of the new HDFS directory now:

```
$ hdfs dfs -ls /loudacre/kb
```

You should see the KB articles that were in the local directory.

Relative paths

In HDFS, relative (non-absolute) paths are considered relative to your home directory. There is no concept of a "current" or "working" directory as there is in Linux and similar file systems.

10. Practice uploading a directory, confirm the upload, and then remove it, as it is not actually needed for the exercises.

```
$ hdfs dfs -put calllogs /loudacre/
$ hdfs dfs -ls /loudacre/calllogs
$ hdfs dfs -rm -r /loudacre/calllogs
```

Viewing HDFS Files

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

11. Enter:

```
$ hdfs dfs -cat /loudacre/kb/KBDOC-00289.html | head \
-n 20
```

This prints the first 20 lines of the article to your terminal. This command is handy for viewing HDFS data. An individual file is often very large, making it inconvenient to view the entire file in the terminal. For this reason, it's often a good idea to pipe the output of the `dfs -cat` command into `head`, `more`, or `less`. You can also use the dfs `-tail` option to more efficiently view the end of the file, rather than piping the whole content.

12. To download a file to work with on the local filesystem use the `hdfs 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 \
/loudacre/kb/KBDOC-00289.html ~/article.html
$ less ~/article.html
```

Enter the letter `q` to quit the `less` command after reviewing the downloaded file.

13. There are several other operations available with the `hdfs dfs` command to perform most common filesystem manipulations such as `mv`, `cp`, and `mkdir`. In the terminal window, enter:

```
$ hdfs dfs
```

You see a help message describing all the file system commands provided by HDFS.

Try playing around with a few of these commands if you like.

Using the Hue File Browser to Browse, View and Manage Files

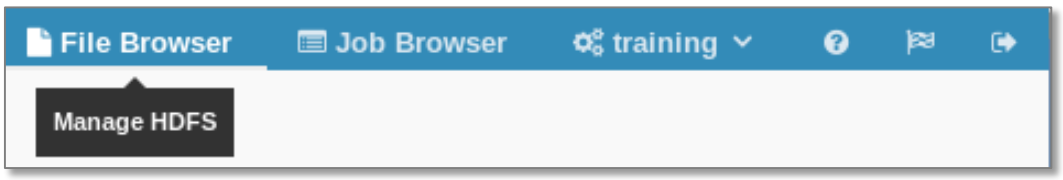
14. Start Firefox on the VM using the shortcut provided on main menu panel at the top of the screen.

15. Click the Hue bookmark, or visit `http://localhost:8888`.

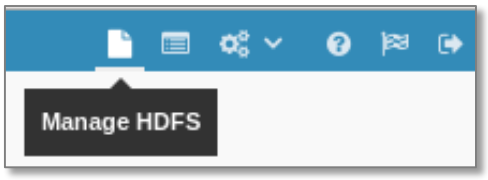
16. Because this is the first time anyone has logged into Hue on this server, you will be prompted to create a new user account. Enter username **`training`** and password **`training`**, and then click **Create Account**. (If prompted you may click "Remember Password").

    - **Note**: When you first log in to Hue you may see a misconfiguration warning. This is because not all the services Hue depends on are running on the course VM in order to save space. You can disregard the message.

17. Hue has many useful features, many of which will be covered later in the course. For now, to access HDFS, click **File Browser** in the Hue menu bar. (The mouse-over text is "Manage HDFS").



- **Note**: If your Firefox window is too small to display the full menu names, you will see just the icons instead.



18. By default, the contents of your HDFS home directory (`/user/training`) display. In the directory path name, click the leading slash (`/`) to view the HDFS root directory.

19. The contents of the root directory display, including the **loudacre** directory you created earlier. Click on that directory to see the contents.

20. Click on the name of the **kb** directory to see the knowledge base articles you uploaded.

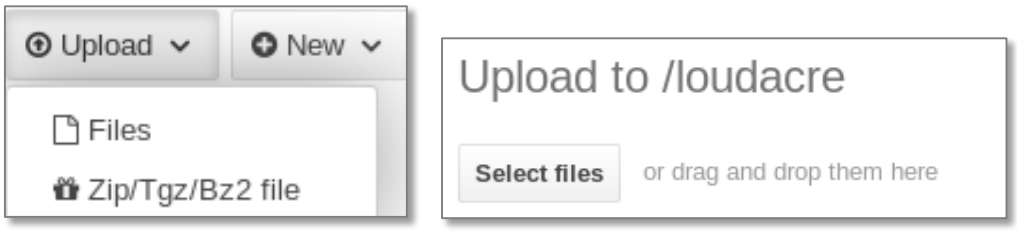
21. View one of the files by clicking on the name of any one of the articles.

    Note: In the file viewer, the contents of the file are displayed on the right. In this case, the file is fairly small, but typical files in HDFS are very large, so rather than displaying the entire contents on one screen, Hue provides buttons to move between pages.

22. Return to the directory review by clicking **View file location** in the Actions panel on the left.

23. Click the up arrow ( ) to return to the `/loudacre` base directory.

24. To upload a file, click the **Upload** button. You can choose to upload a plain file, or to upload a zipped file (which will be automatically unzipped after upload). In this case, select **Files**, then click **Select Files**.



25. A Linux file browser appears. Browse to `/home/training/training_materials/data`.

26. Choose `base_stations.tsv` and click the **Open** button.

27. When the file has uploaded, it will be displayed in the directory. Click the checkbox next to the file's icon, and then click the **Actions** button to see a list of actions that can be performed on the selected file(s).

28. *Optional*: explore the various file actions available. When you've finished, select any unneeded files you have uploaded and click the **Move to trash** button to delete.

This is the end of the Exercise

Hands-On Exercise: Run a YARN Job

Files and Data Used in this Exercise

Exercise directory: `$DEV1/exercises/yarn`

Dataset (HDFS): `/loudacre/kb`

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

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

Exploring the YARN Cluster

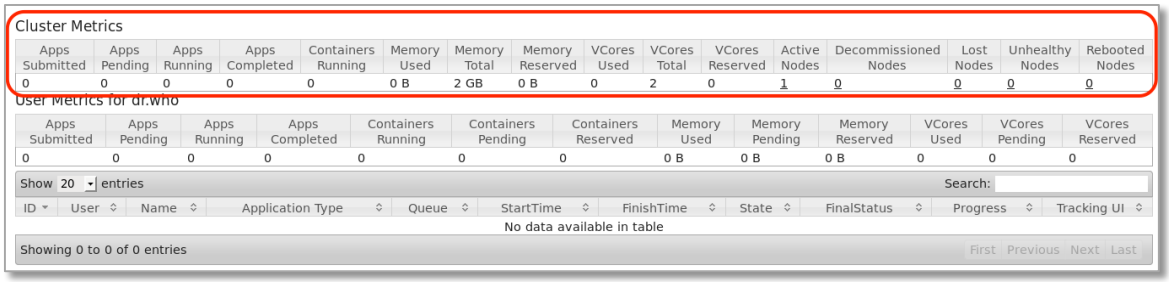
1. Visit the YARN Resource Manager (RM) UI in Firefox using the provided bookmark, or by going to URL `http://localhost:8088`.

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

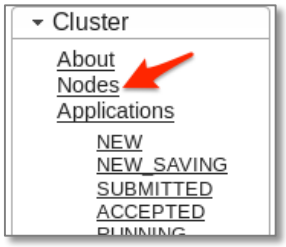
Who is Dr. Who?

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

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

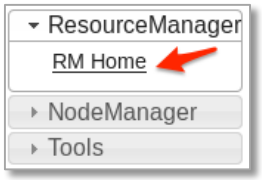


3. Click on the **Nodes** link in the Cluster menu on the left. The bottom section will display a list of worker nodes in the cluster. The pseudo-distributed cluster used for training has only a single node, which is running on the local machine. In the real world, this list would show multiple worker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



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

5. To return to the Resource Manager, expand **ResourceManager → RM Home** on the left.



Submitting an Application to the YARN Cluster

6. In a terminal window, change to the exercise directory:

```
$ cd $DEV1/exercises/yarn
```

7. Run the example `wordcount.py` program on the YARN cluster to count the frequency of words in the knowledge dataset:

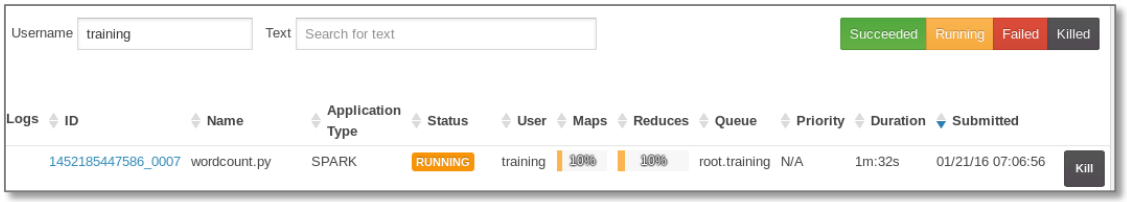
```
$ spark-submit --master yarn-client \
  wordcount.py /loudacre/kb/*
```

Later in the course you will learn how to write and submit your own Spark applications, as you are doing here, including what the various options mean. For now, focus on learning about the YARN UI.

Viewing the Application in the Hue Job Browser

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

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

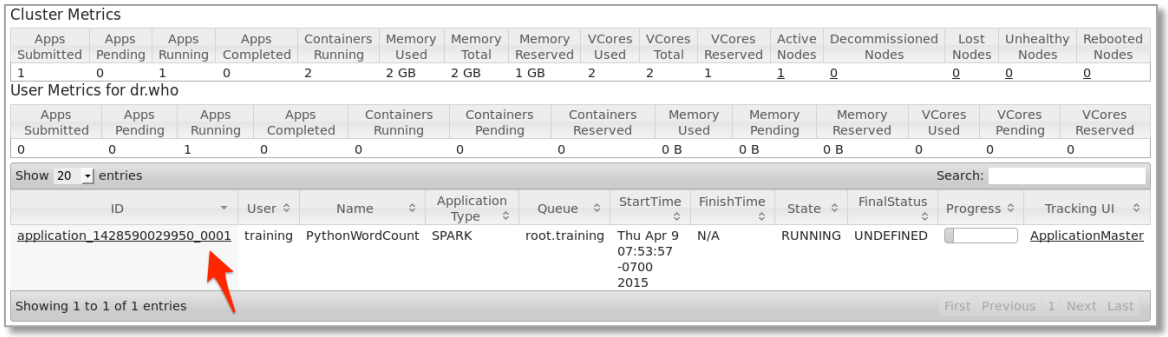


10. This page allows you to click the application ID to see details of the running application, or to kill a running job. (Do not do that now though!)

Viewing the Application in the YARN UI

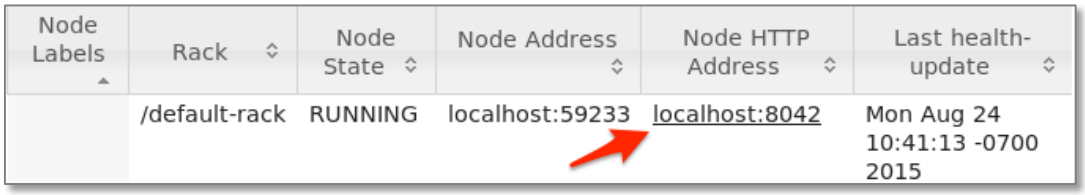
To get a more detailed view of the cluster, use the YARN UI.

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

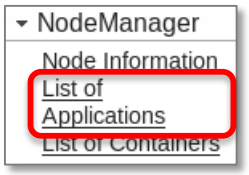


12. As you did in the first exercise section, select **Nodes**.

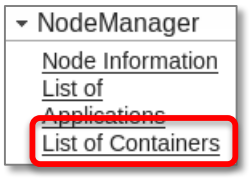
13. Select the **Node HTTP Address** to open the Node Manager UI.



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



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



This will display the containers the Resource Manager has allocated on the selected node for the current application. (No containers will show if no applications are running; if you missed it because the application completed, you can run the application again.)

This is the end of the Exercise

Hands-On Exercise: Import Data from MySQL Using Sqoop

Files and Data Used in this Exercise

Exercise directory:	`$DEV1/exercises/sqoop`
MySQL database:	`loudacre`
MySQL tables:	`accounts`
	`webpage`
HDFS paths:	`/loudacre/accounts`
	`/loudacre/webpage`

In this exercise, you will import tables from MySQL into HDFS using Sqoop.

Importing a Table From MySQL to HDFS

You can use Sqoop to look at the table layout in MySQL. With Sqoop, you can also import the table from MySQL to HDFS.

1. Open a new terminal window if necessary.

2. Run the `sqoop help` command to familiarize yourself with the options in Sqoop:

```
$ sqoop help
```

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

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

4. Run the `sqoop import` command to see its options:

```
$ sqoop import --help
```

5. Use Sqoop to import the `accounts` table in the `loudacre` database and save it in HDFS under `/loudacre`:

```
$ sqoop import \
  --connect jdbc:mysql://localhost/loudacre \
  --username training --password training \
  --table accounts \
  --target-dir /loudacre/accounts \
  --null-non-string '\\N'
```

   The `--null-non-string` option tells Sqoop to represent `null` values as `\N`, which makes the imported data compatible with Hive and Impala.

6. *Optional*: While the Sqoop job is running, try viewing it in the Hue Job Browser or YARN Web UI, as you did in the previous exercise.

Viewing the Imported Data

Sqoop imports the contents of the specified tables to HDFS. You can use the command line or the Hue File Browser to view the files and their contents.

7. List the contents of the `accounts` directory:

```
$ hdfs dfs -ls /loudacre/accounts
```

   - **Note**: Output of Hadoop processing jobs is saved as one or more numbered "partition" files. Partitions are covered later in the course.

8. Use either the Hue File Browser or the `-tail` option to the `hdfs` command to view the last part of the file for each of the MapReduce partition files, for example:

```
$ hdfs dfs -tail /loudacre/accounts/part-m-00000
$ hdfs dfs -tail /loudacre/accounts/part-m-00001
$ hdfs dfs -tail /loudacre/accounts/part-m-00002
$ hdfs dfs -tail /loudacre/accounts/part-m-00003
```

9. The first six digits in the output are the account ID. From the last line of the last file, take note of highest account ID because you will use it in the next step.

Importing Incremental Updates

As Loudacre adds new accounts, the account data in HDFS must be updated as accounts are created. You can use Sqoop to append these new records.

10. Run the `add_new_accounts.py` script to add the latest accounts to MySQL.

```
$ $DEV1/exercises/sqoop/add_new_accounts.py
```

11. Incrementally import and append the newly added accounts to the `accounts` directory. Use Sqoop to import on the last value on the `acct_num` column largest account ID:

```
$ sqoop import \
 --connect jdbc:mysql://localhost/loudacre \
 --username training --password training \
 --incremental append \
 --null-non-string '\\N' \
 --table accounts \
 --target-dir /loudacre/accounts \
 --check-column acct_num \
 --last-value <largest_acct_num>
```

Note: replace *<largest_acct_num>* with the largest account number.

12. List the contents of the `accounts` directory to verify the Sqoop import:

```
$ hdfs dfs -ls /loudacre/accounts
```

13. You should see three new files. Use Hadoop's `cat` command to view the entire contents of these files.

```
$ hdfs dfs -cat /loudacre/accounts/part-m-0000[456]
```

Importing Data Using an Alternate Field Delimiter

14. We also want to import the `webpage` table to HDFS, but first, use the `sqoop eval` command to look at a few records in that table:

```
$ sqoop eval \
  --query "SELECT * FROM webpage LIMIT 10" \
  --connect jdbc:mysql://localhost/loudacre \
  --username training --password training
```

Notice that the values in the last column contain commas. By default, `sqoop` uses commas as field separators, but because the data itself uses commas, we cannot do that this time.

15. Use Sqoop to import the webpage table, but use the tab character (`\t`) instead of the default (comma) as the field terminator.

```
$ sqoop import \
  --connect jdbc:mysql://localhost/loudacre \
  --username training --password training \
  --table webpage \
  --target-dir /loudacre/webpage \
  --fields-terminated-by "\t"
```

16. Using Hue or the `hdfs` command line utility, review the data files imported to the `/loudacre/webpage` directory. Take note of the structure of the data; you will use this data in the next exercises.

Bonus Exercise

Use Sqoop to import only accounts where the person lives in California (`state = "CA"`) and has an active account (`acct_close_dt IS NULL`).

Note: This bonus exercise must output to a different HDFS directory. Otherwise, the original table will be overwritten and you will need to delete the directory and import the table again.

If you need a hint or would like to check your work, you can find a solution in the `$DEV1/exercises/sqoop/solution/bonus-sqoop-query.sh` file.

This is the end of the Exercise

Hands-On Exercise: Create and Populate Tables in Impala or Hive

Files and Directories Used in this Exercise

Exercise directory: `$DEV1/exercises/impala`

MySQL database: `loudacre`
MySQL table: `device`
HDFS path: `/loudacre/webpage`

In these exercises you will define Impala/Hive tables to model and view data in HDFS.

Note: The accounts data will not be used in this exercise but will in a subsequent exercise.

You may perform this and subsequent exercises in either Impala or Hive. Most of the instructions are the same whichever tool you choose; where the instructions differ, the difference is noted. Following whichever set of instructions you prefer; if you have no preference, we suggest Impala because it is faster.

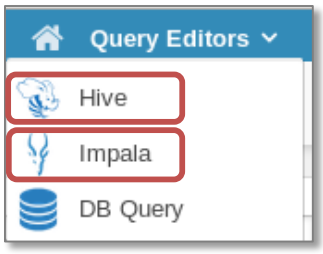
Creating and Querying a Table in Impala or Hive

In this section you will use Impala or Hive to query the webpage data you imported in the previous exercise.

There are a number of ways to interact with Impala and Hive. In this exercise you will use the Impala or Hive Query Editor in Hue.

1. Visit the Hue page in Firefox.

2. Open either the Impala query editor or Hive query editor, by selecting the editor of your choice from the **Query Editors** menu.



3. In the query editor pane on the right, enter an SQL command to create a table for the web page data imported in previous exercises:

```
CREATE EXTERNAL TABLE webpage
   (page_id SMALLINT,
    name STRING,
    assoc_files STRING)
   ROW FORMAT DELIMITED
      FIELDS TERMINATED BY '\t'
      LOCATION '/loudacre/webpage'
```

4. Click the **Execute** button to execute the command.

5. To see the table you just created, refresh the table list on the left.

   - Note: In the Impala Query Editor you will be prompted with the type of refresh to perform; choose **Clear Cache** and click **Refresh.**

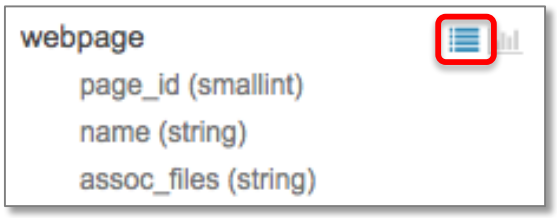
6. Click on the **webpage** table to see the column definitions. (You may need to scroll down through the list of tables to see the columns.)

7. Click the **New Query** button, then enter and execute a test query such as:

```
SELECT * FROM webpage WHERE name LIKE "ifruit%"
```

The resulting data is shown in the **Results** tab of the pane below the query.

8. Click on the **Preview Sample Data** icon. (Note: Hover your mouse pointer over the table to view the controls.)



Using Sqoop to Import Directly into Hive and Impala

In this section you will use Sqoop to import data from MySQL into HDFS and also automatically create the corresponding table in the Hive Metastore.

9. In a terminal window, import the `device` table directly into the Hive Metastore.

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

Use `--hive-import` for either Impala or Hive; this adds metadata to the Metastore, which both tools use.

Note: You may get a warning message that `chgrp` is unable to change the ownership of the generated files; you can disregard the warning, it does not affect the import.

10. Using Hue or the command line, review the imported data files. The Sqoop Hive import copies the data to the default Hive warehouse location: `/user/hive/warehouse/device`.

11. If you are using Impala, refresh the Impala metadata cache by entering the command in the Hue Impala Query Editor:

```
INVALIDATE METADATA
```

12. As in the previous exercise, view the columns and execute a test query:

```
SELECT * FROM device LIMIT 10
```

This is the end of the Exercise

Hands-On Exercise: Store and Access Data in a Non-Text Format Data File

Files and Data Used in this Exercise

Exercise directory: `$DEV1/exercises/data-format/`

MySQL database: `loudacre`

MySQL table: `accounts`

In this exercise, you will use import data in Avro format and create an Impala/Hive table to access it.

1. Change directories to the exercise directory.

```
$ cd $DEV1/exercises/data-format/
```

2. Import the `accounts` table to an Avro data format.

```
$ sqoop import \
  --connect jdbc:mysql://localhost/loudacre \
  --username training --password training \
  --table accounts \
  --target-dir /loudacre/accounts_avro \
  --null-non-string '\\N' \
  --as-avrodatafile
```

3. View the files imported by Sqoop into HDFS using the `hdfs` command. What do you see when you try to view the content of the data files?

4. Optional: Download one of the generated Avro files to a local (non-HDFS) directory, and use the `avro-tools tojson` command to view the file contents.

```
$ hdfs dfs -get \
 /loudacre/accounts_avro/part-m-00000.avro
$ avro-tools tojson part-m-00000.avro | more
```

5. Sqoop generates a schema (in the file `accounts.avsc`) in the current directory. Review this file and copy it to HDFS below `/loudacre/`.

```
$ hdfs dfs -put accounts.avsc /loudacre/
```

6. In Impala or Hive, create a table using this schema:

```
CREATE EXTERNAL TABLE accounts_avro
  STORED AS AVRO
  LOCATION '/loudacre/accounts_avro'
  TBLPROPERTIES ('avro.schema.url'=
    'hdfs:/loudacre/accounts.avsc')
```

7. Confirm correct creation of the table by issuing a query, such as:

```
SELECT * FROM accounts_avro LIMIT 10
```

8. Optional: Use the `DESCRIBE` or `DESCRIBE FORMATTED` command to list the columns and data types of the `accounts_avro` table created from the Avro schema.

Bonus Exercise

Create a new table based on the existing `accounts_avro` table data, using Parquet for the storage format.

This is the end of the Exercise

Hands-On Exercise: Partition Data in Impala or Hive

Files and Data Used in this Exercise

Exercise directory: `$DEV1/exercises/data-partition`

HDFS path: `/loudacre/accounts_avro`

In this exercise you will create and load an Impala/Hive table with account data, partitioned by area code.

Previously you imported data from the accounts table using Sqoop, into a table called `accounts_avro`. In this exercise, you will create a new table with some of the account data, partitioned by area code (the first three digits of the phone number).

1. Create a new, empty table with Impala or Hive:

```
CREATE EXTERNAL TABLE accounts_by_areacode (
    acct_num INT,
    first_name STRING,
    last_name STRING,
    phone_number STRING)
  PARTITIONED BY (areacode STRING)
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  LOCATION '/loudacre/accounts_by_areacode'
```

2. In order to populate the new table, you will need to extract the area code from the phone number. Try executing the following query to demonstrate:

```
SELECT acct_num, first_name, last_name,
   phone_number, SUBSTR(phone_number,1,3) AS areacode
  FROM accounts_avro
```

3. Use the `SELECT` statement above in an `INSERT INTO TABLE` command to copy the specified columns to the new table, dynamically partitioning by area code.

4. Execute a simple query to confirm that the table was populated correctly, such as:

```
SELECT * FROM accounts_by_areacode LIMIT 10
```

5. Using Hue or the `hdfs` command line interface, confirm that the directory structure of the `accounts_by_areacode` table includes partition directories. Review the data in the directories to verify that the partitioning is correct.

This is the end of the Exercise

Hands-On Exercise: Collect Web Server Logs with Flume

Files and Data Used in this Exercise

Exercise directory: `$DEV1/exercises/flume`

Data files (local): `$DEV1DATA/weblogs/*`

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

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

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

Creating an HDFS Directory for Flume Ingested Data

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

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

Creating a Local Directory for Web Server Log Output

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

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

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

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

Configuring Flume

In `$DEV1/exercises/flume` under `stubs` edit the provided Flume starter configuration file with the characteristics listed below. If you need help getting started, you may refer to configuration file in `hints`, or view the solution in the `solutions` directory.

- The source is a spooling directory source that pulls from `/flume/weblogs_spooldir`
- The sink is an HDFS sink that:
  - Writes files to the `/loudacre/weblogs` directory
  - Disables time-based file rolling by setting the `hdfs.rollInterval` property to `0`
  - Disables event-based file rolling by setting the `hdfs.rollCount` property to `0`
  - Sets the `hdfs.rollSize` property as `524288` to enable size-based file rolling at 512KB
  - Writes raw text files (instead of SequenceFile format) by setting `hdfs.fileType` to `DataStream`
- The channel is a Memory Channel that:
  - Can store 10,000 events using the `capacity` property
  - Has a transaction capacity of 1,000 events using the `transactionCapacity` property

Flume Performance Note

Flume's performance on a VM will vary. Sometimes, Flume will exhaust the memory capacity of its channel and give the following error:

```
Space for commit to queue couldn't be acquired. Sinks are
likely not keeping up with sources, or the buffer size is
too tight
```

If you get this error, you will need to increase the `capacity` and `transactionCapacity` properties to a higher value. Then, you will need to restart the agent, clean up any files in HDFS, and repeat the Flume operation.

Running the Agent

Once you have created the configuration file, you need to start the agent and copy the files to the spooling directory.

4. Change directories to the `$DEV1/exercises/flume` directory.

5. Start the Flume agent using the configuration you just made. For example, if you wish to use the solution configuration, enter this command (replace `solution` with `stubs` to run your own configuration):

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

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

Simulating Apache Web Server Output

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

```
$ cd $DEV1/exercises/flume
$ ./copy-move-weblogs.sh /flume/weblogs_spooldir
```

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

8. Return to the terminal that is running the Flume agent and watch the logging output. The output will give information about the files Flume is putting into HDFS.

9. Once the Flume agent has finished, enter `CTRL+C` to terminate the process.

10. Using the command line or Hue File Browser, list the files in HDFS that were added by the Flume agent.

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

Bonus Exercise

Loudacre has a data source that comes in via the network. You want to see some examples since the format is undocumented.

Flume has a `netcat` source, which allows Flume to listen on a port and process the contents received. The address to bind to is specified by the `bind` property, and the port to bind to is specified by the `port` property.

Flume also has a `logger` sink, which logs any incoming data at the `INFO` level in Log4J. This is helpful for debugging and testing Flume configuration. This sink does not require any configuration properties.

Create a Flume configuration file with the following characteristics:

- Uses the `netcat` source to capture data from the current host on port `12345`.
    - **Hint:** use the `hostname` command to get the correct name of your host
- Uses the `logger` sink
- Uses the `memory` channel

Start the Flume agent using the new configuration file.

Unresolved Address Error

Depending on your VM environment, you may get an error that Flume is unable to resolve the address for the hostname you specified in the configuration file. If this happens, you can solve the problem by adding the hostname to the end of the last line of the `/etc/hosts` file. Note that editing that file requires superuser privileges, so start the editor using the `sudo` command, such as:

```
$ sudo gedit /etc/hosts
```

In another terminal window, start the test data feed script using the command:

```
$ $DEV1/exercises/flume/script/rng_feed.py
```

Observe the log messages in the Flume agent's terminal and stop the agent after you have seen a few example records.

This is the end of the Exercise

Hands-On Exercise: Explore RDDs Using the Spark Shell

Files and Data Used in this Exercise

Exercise directory:	`$DEV1/exercises/spark-shell`
Data files (local):	`$DEV1DATA/frostroad.txt`
Data files (HDFS):	`/loudacre/weblogs/*`

In this Exercise you will use the Spark Shell to work with RDDs.

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

Viewing 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 **Scala** or **Python**, depending on your language preference. Bookmark the API page for use during class. Later exercises will refer you to this documentation.

Starting the Spark Shell

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

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

Starting the Python Spark Shell

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

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

```
$ pyspark
```

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

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

```
pyspark> sc
```

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

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

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

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

7. You can exit the shell by hitting control-d or by typing `exit`. But stay in the shell now to complete the remainder of this exercise.

Starting the Scala Spark Shell

Follow these instructions if you are using Scala to complete this exercise. Otherwise, skip this section and continue with Reading and Displaying a Text File.

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

```
$ spark-shell
```

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

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

```
scala> sc
```

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

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

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

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

12. You can exit the shell at any time by typing `sys.exit`. But stay in the shell now to complete the remainder of this exercise.

Reading and Displaying a Text File (Python or Spark)

13. Review the simple text file you will be using by viewing (without editing) the file in a text editor in a separate window (not the Spark shell). The file is located at: `$DEV1DATA/frostroad.txt`.

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

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

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

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

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

```
pyspark> mydata.count()
```

```
scala> mydata.count()
```

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

`Out[4]: 23` (Python) *or*
`res0: 23` (Scala)

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

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

```
scala> mydata.collect()
```

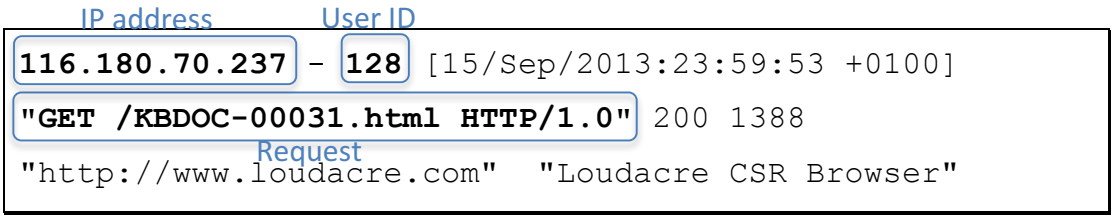
17. Using command completion, you can see all the available transformations and operations you can perform on an RDD. Type `mydata.` and then the [TAB] key.

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

Exploring the Loudacre Web Log Files

In this section you will be using web server log files previously imported into HDFS using Flume.

18. Using the HDFS command line or Hue File Browser, review one of the files in the HDFS `/loudacre/weblogs` directory, such as `FlumeData.1423586038966`. Note the format of the lines, for example:



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

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

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

20. Create an RDD from the data file.

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

```
scala> var logs = sc.textFile(logfile)
```

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

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

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

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

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

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

23. Sometimes you do not need to store intermediate objects in a variable, in which case you can combine the steps into a single line of code. For instance, 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 line: \
  ".jpg" in line).count()
```

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

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

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

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

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

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

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

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

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

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

```
pyspark> ips = logs.map(lambda line: line.split()[0])
pyspark> ips.take(5)
```

```
scala> var ips = logs.map(line =>line.split(' ')(0))
scala> ips.take(5)
```

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

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

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

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

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

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

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

Bonus Exercise

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

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

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

This is the end of the Exercise

Hands-On Exercise: Process Data Files with Spark

Files and Data Used in This Exercise:

Exercise directory:	`$DEV1/exercises/spark-etl`
Data files (local):	`$DEV1DATA/activations/*` `$DEV1DATA/devicestatus.txt` (Bonus)
Stubs:	`ActivationModels.pyspark` `ActivationModels.scalaspark`

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

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

Reviewing the API Documentation for RDD Operations

1. Visit the Spark API page you bookmarked previously. Follow the link for the `RDD` class and review the list of available methods. (In the Scala API, the link will be near the top of the main window; in Python scroll down to the Core Classes area.)

Reviewing the Data

2. Review the data in `$DEV1DATA/activations`. Each XML file contains data for all the devices activated by customers during a specific month.

   Sample input data:

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

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

Processing the Files

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

The output will look something like:

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

4. Start with the `ActivationModels` stub script in the exercise directory. (A stub is provided for Scala and Python; use whichever language you prefer.) Note that for convenience you have been provided with functions to parse the XML, as that is not the focus of this Exercise. Copy the stub code into the Spark Shell.

5. Use `wholeTextFiles` to create an RDD from the activations dataset. The resulting RDD will consist of tuples, in which the first value is the name of the file, and the second value is the contents of the file (XML) as a string.

6. Each XML file can contain many activation records; use `flatMap` to map the contents of each file to a collection of XML records by calling the provided `getactivations` function. `getactivations` takes an XML string, parses it, and returns a collection of XML records; `flatMap` maps each record to a separate RDD element.

7. Map each activation record to a string in the format `account-number:model`. Use the provided `getaccount` and `getmodel` functions to find the values from the activation record.

8. Save the formatted strings to a text file in the directory `/loudacre/account-models`.

Bonus Exercise

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

Review the contents of the file `$DEV1DATA/devicestatus.txt`. This file contains data collected from mobile devices on Loudacre's network, including device ID, current status, location and so on. Because Loudacre previously acquired other mobile provider's networks, the data from different subnetworks has a different format. Note that the records in this file have different field delimiters: some use commas, some use pipes (`|`) and so on. Your task is to

- Load the dataset
- Determine which delimiter to use (hint: the character at position 19 is the first use of the delimiter)
- Filter out any records which do not parse correctly (hint: each record should have exactly 14 values)

- Extract the date (first field), model (second field), device ID (third field), and latitude and longitude (13th and 14th fields respectively)
- The second field contains the device manufacturer and model name (such as `Ronin S2`.) Split this field by spaces to separate the manufacturer from the model (for example, manufacturer `Ronin`, model `S2`.)
- Save the extracted data to comma delimited text files in the `/loudacre/devicestatus_etl` directory on HDFS.
- Confirm that the data in the file(s) was saved correctly.

The solutions to the bonus exercise are in `$DEV1/exercises/spark-etl/bonus`.

This is the end of the Exercise

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

Files and Data Used in This Exercise:

Exercise directory: `$DEV1/exercises/spark-pairs`

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

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

Exploring Web Log Files

Continue working with the web log files, as in the previous exercise.

Tip: In this exercise you will be reducing and joining large datasets, which can take a lot of time. You may wish to perform the exercises below using a smaller dataset, consisting of only a few of the web log files, rather than all of them. Remember that you can specify a wildcard; `textFile("/loudacre/weblogs/*56")` would include only filenames ending with the digits 56.

1. Using map-reduce, count the number of requests from each user.

a. Use `map` to create a Pair RDD with the user ID as the key, and the integer 1 as the value. (The user ID is the third field in each line.) Your data will look something like this:

(*userid*,1)
(*userid*,1)
(*userid*,1)
…

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

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

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

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

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

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

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

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

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

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

Joining Web Log Data with Account Data

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

4. Join the accounts data with the weblog data to produce a dataset keyed by user ID which contains the user account information and the number of website hits for that user.

a. Create an RDD based on the accounts data consisting of key/value-array pairs: `(userid,[values…])`

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

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

```
(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 (4th value) and last name (5th value) for the first 5 elements, for example:

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

Managed Memory Leak Error Message

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

```
ERROR Executor: Managed memory leak detected
```

This message is a Spark bug and can be disregarded.

Bonus Exercises

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

1. Use `keyBy` to create an RDD of account data with the postal code (9th field in the CSV file) as the key.
   - Tip: Assign this new RDD to a variable for use in the next bonus exercise
2. Create a pair RDD with postal code as the key and a list of names (Last Name,First Name) in that postal code as the value.
   - Hint: First name and last name are the 4th and 5th fields respectively
   - Optional: Try using the `mapValues` operation

3. Sort the data by postal code, then for the first five postal codes, display the code and list the names in that postal zone, for example:

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

This is the end of the Exercise

Hands-On Exercise: Write and Run 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.

Writing 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. If you are using Python, follow these instructions; otherwise, skip this section and continue to Writing a Spark Application in Scala below.

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

3. Set up a SparkContext using the following code:

```
sc = SparkContext()
```

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

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

```
$ cd $DEV1/exercises/spark-application/
$ spark-submit CountJPGs.py /loudacre/weblogs/*
```

6. Skip the section below on writing a Spark application in Scala and continue with Submitting a Spark Application to the Cluster.

Writing a Spark Application in Scala

> 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`.

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

8. Set up a SparkContext using the following code:

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

9. 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.

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

```
$ cd $DEV1/exercises/spark-application/countjpgs
$ mvn package
```

11. 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/*
```

Submitting a Spark Application to the Cluster

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

12. 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.

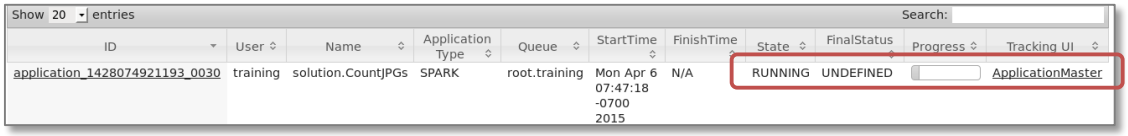
To run Python:

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

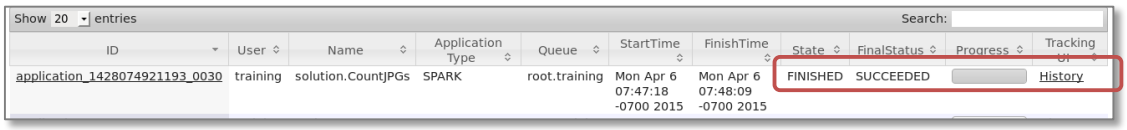
To run Scala:

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

13. 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:



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



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:	`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.

Setting Configuration Options at the Command Line

1. Re-run 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.

Setting Configuration Options in a Properties 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:

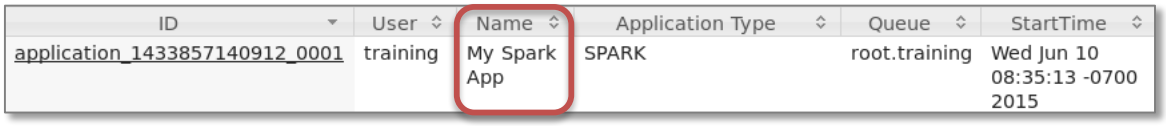
```
spark.app.name  My Spark App
spark.master    yarn-client
spark.executor.memory     400M
```

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

Setting Logging Levels

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

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

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

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

10. The first line currently reads:

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

Replace `INFO` with `DEBUG`:

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

11. Save the file and exit the editor.

12. Rerun your Spark application. Notice that the output now contains both `INFO` and `DEBUG` messages, like this:

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

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

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

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

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

This is the end of the Exercise

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

Files and Data Used in This Exercise:

Exercise directory:	`$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.

Exploring Partitioning of File-Based RDDs

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

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

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

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

3. Create an RDD based on a *single file* in the dataset, such as `/loudacre/accounts/part-m-00000`, and then call `toDebugString` on the RDD, which displays the number of partitions in parentheses `()` before the RDD 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.

Setting up the Job

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

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

```
scala> 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; for example, select only filenames ending in 6 by specifying `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?

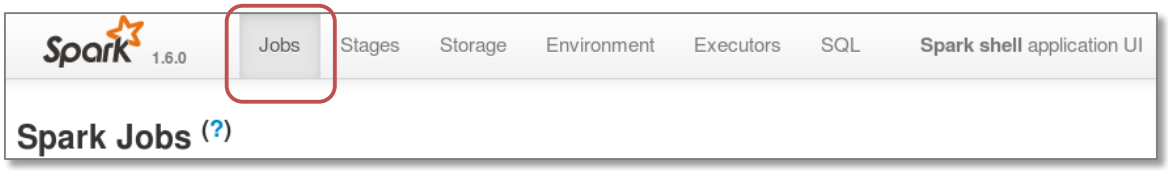
    b. Which stages are dependent on which?

    c. How many tasks will each stage consist of?

Running the Job and Reviewing the Job 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.

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 corresponds 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.

16. 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`.)

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

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

Bonus question: Which tasks were skipped and why?

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:	`$DEV1/exercises/spark-persist`
Data files (HDFS):	`/loudacre/weblogs/*` `/loudacre/accounts/*`
Job setup scripts:	`$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 two 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.

Reviewing 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 `$DEV1DATA/devicestatus.txt` to HDFS directory `/loudacre/`
- In the Spark shell, copy in and execute the Spark script in `$DEV1/exercises/spark-etl/bonus/DeviceStatusETL` (either `.pyspark` or `.scalaspark` 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, as shown in the sample data below:

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

Calculating k-means for Device Location

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.

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

2. The stub code also sets the variable `convergeDist`. This will be used to decide when the k-means calculation is done – when the amount the locations of the means changes between iterations is less than `convergeDist`. A "perfect" solution would be 0; this number represents a "good enough" solution. For this exercise, use a value of `0.1`.

3. 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.

4. Create a K-length array called `kPoints` by taking a random sample of K location points from the RDD as starting means (center points). For example, in Python:

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

Or in Scala:

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

5. 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.) For example:

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

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

center, and the number of closest points. For example:

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

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

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

This is the end of the Exercise

Optional Hands-On Exercise: Partition Data Files Using Spark

Files and Data Used in this Exercise

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

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

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

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 working on your solution using the stub files in the exercise directory, which provide code to read the region location datafile into a Dictionary,

using the region as the key, and (latitude,longitude) pairs as the 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 an Impala/Hive table, process it, and store it to a new table.

Note that this exercise depends on completion of a prior exercise in which you imported the `webpage` table from MySQL to Impala/Hive using Sqoop. If you did not complete that exercise, be sure to run the `$DEV1/scripts/catchup.sh` script.

Creating a DataFrame from a Table

1. If necessary, start the Spark Shell.

2. The Spark Shell predefines a SQL Context object as `sqlContext`. What type is the SQL Context? In either Python or Scala, view the `sqlContext` object:

```
> sqlContext
```

3. View the tables currently defined in the Metastore:

```
scala> sqlContext.tableNames().foreach(println)
```

```
pyspark> for table in sqlContext.tableNames(): \
  print table
```

The list should include the `webpage` table previously imported.

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

```
scala> val webpagedf = sqlContext.read.table("webpage")
```

```
pyspark> webpagedf = sqlContext.read.table("webpage")
```

5. Examine the schema of the new DataFrame by calling `webpagedf.printSchema()`.

6. View the first few records in the table by calling `webpagedf.show(5)`.

   Note that the data in the `assoc_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 `assoc_files` data must be extracted and normalized. Your goal in the next section is to use the DataFrames API to extract the data in the column, split the string, and create a new dataset in HDFS containing each page ID, and its associated files in separate rows.

Querying a DataFrame

7. Create a new DataFrame by selecting the `page_id` and `assoc_files` columns from the existing DataFrame:

```
scala> val assocfilesdf =
  webpagedf.select($"page_id",$"assoc_files")
```

```
python> assocfilesdf = \
  webpagedf.select(webpagedf.page_id, \
  webpagedf.assoc_files)
```

8. View the schema and the first few rows of the returned DataFrame to confirm that it was created correctly.

9. In order to manipulate the data using core Spark, convert the DataFrame into a Pair RDD using the `map` method. The input into the map method is a `Row` object. The key is the `page_id` value, and the value is the `assoc_files` string.

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

```
scala> val afilesrdd = assocfilesdf.
  map(row => (row.getAs[Short]("page_id"),
    row.getAs[String]("assoc_files")))
```

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

```
pyspark> afilesrdd = assocfilesdf.map(lambda row: \
  (row.page_id,row.assoc_files))
```

10. 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(','))
```

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

```
scala> import org.apache.spark.sql.Row
scala> val afilesrowrdd = afilesrdd2.map(pair =>
Row(pair._1,pair._2))
```

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

```
scala> val afiledf = sqlContext.
  createDataFrame(afilesrowrdd,assocfilesdf.schema)
```

```
pyspark> afiledf = sqlContext. \
  createDataFrame(afilesrdd2,assocfilesdf.schema)
```

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

14. Create a new DataFrame with the `assoc_files` column renamed to `associated_file`:

```
scala> val finaldf = afiledf.
  withColumnRenamed("assoc_files","associated_file")
```

```
pyspark> finaldf = afiledf. \
  withColumnRenamed('assoc_files','associated_file')
```

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

16. Your final DataFrame contains the processed data, so save it in Parquet format (the default) in table `webpage_files`.

```
scala> finaldf.write.
  mode("overwrite").
  saveAsTable("webpage_files")
```

```
pyspark> finaldf.write. \
  mode("overwrite"). \
  saveAsTable("webpage_files")
```

Viewing the Output

17. Using Hue or the HDFS command line, list the Parquet files that were saved by Spark SQL. (The directory will be created in the default location for Hive/Impala tables, `/user/hive/warehouse`.)

18. Using the Hue Impala or Hive Query Editor, view the data in the new `webpage_files` table. (Remember that if you are using Impala you will need to invalidate the cached metadata to see the new table.)

19. Optional: In the Spark Web UI (`http://localhost:4040`), try viewing the **SQL** tab. How many queries were completed as part of this exercise? How many jobs?

Bonus: Writing Avro Data

In the previous section, you saved the `webpage_files` as a Parquet format table in Hive/Impala. Another format option is Avro, but Spark SQL does not support directly defining an Avro table in the Metastore. Instead you will need to complete the steps separately:

- From Spark, save the file using the format `com.databricks.spark.avro`
- In Hive or Impala, define an external table to access the data.
    - Hint: the table will have two columns: a `BIGINT` for the page ID and a `STRING` for the associated file.

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`

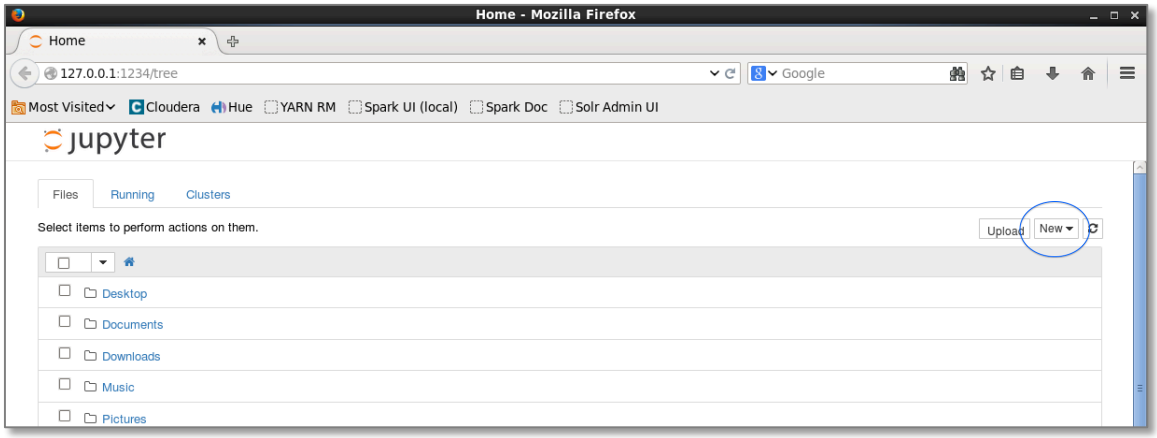
2. Uncomment out the following line (remove the leading #).

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

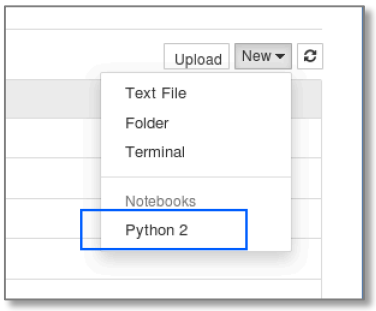
3. Save the file.

4. Open a new terminal window. (It must be a new terminal so it reloads your edited `.bashrc` file).

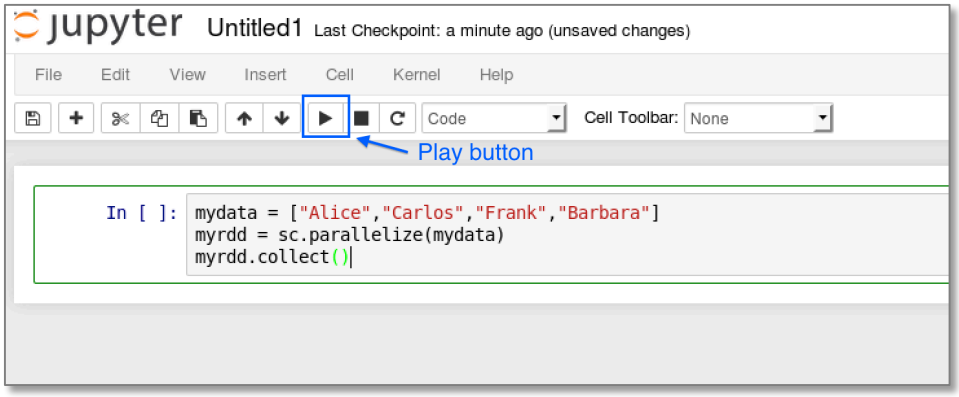
5. Enter `pyspark` in the terminal. This will cause a browser window to open, and you should see the following web page:



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



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



8. Notice the output displayed.