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

Table of Contents

General Notes

Exercise Environment Overview

This course provides an exercise environment running the Cloudera and Hadoop services necessary to complete the exercises.

The exercise environment runs on a single host machine running CentOS Linux. Your user name is `training`. You will be logged in automatically, but if for any reason you need to log out and back in, the password is `training`.

Real-world production clusters almost always contain many machines, but for this course, the exercise environment uses a single host machine with a cluster running in "pseudo-distributed" mode to keep instructions simple.

Although all the cluster services are running on the same remote machine, different services are configured with different host names. In order for the exercises be as realistic as possible, instructions will refer to services using the host names corresponding to the types of nodes that are part of in a typical cluster. For example, services that would usually run on a master node are configured with the host name `master`.

The table below shows the various host names in the pseudo-distributed cluster and the corresponding role that type of host would play in a full cluster.

Host Names	Role
`gateway`	A gateway node (sometimes referred to as an edge node) is a node outside the cluster that provides you with access to the services running on the cluster. Users and developers typically do their work on gateway nodes rather than cluster nodes.
`cmhost`	This node runs Cloudera Manager, which installs, configures, and monitors the services on the Hadoop cluster.
`master-1` `master-2`	Master nodes run the services that manage the Hadoop cluster.

`worker-1`
`worker-2`
`worker-3`
	Worker nodes execute the distributed tasks for applications that run on the Hadoop cluster.

Course Exercise Directories

The main directory is `~/training_materials/devsh/exercises`. Within that directory you will find the following subdirectories:

- `exercises`—contains subdirectories corresponding to each exercise, which are referred to in the instructions as the "exercise directory." The exercises directories contain starter code (stubs), solutions, Maven project directories for Scala and Java applications, and other files needed to complete the exercise.
- `data`—contains the data files used in all the exercises. Usually you will upload the files to Hadoop's distributed file system (HDFS) before working with them.
- `examples`—contains example code and data presented in the chapter slides in the course.
- `scripts`—contains the course setup scripts and other scripts required to complete the exercises.

Working with the Linux Command Line

- In some steps in the exercises, you will see instructions to enter commands like this:

```
$ hdfs dfs -put mydata.csv \
  /user/training/example
```

  The dollar sign (`$`) at the beginning of each line indicates the Linux shell prompt. The actual prompt will include additional information (for example, `training@localhost:~/training_materials$`) but this is omitted from these instructions for brevity.

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

- The command-line environment defines a few environment variables that are often used in place of longer paths in the instructions. Since each variable is automatically replaced with its corresponding values when you run commands in the terminal, this makes it easier and faster for you to enter a command:
  - `$DEVSH` refers to the main course directory under `~/training_materials`.
  - `$DEVDATA` refers to the directory containing the data files used in the exercises.

  Use the `echo` to see the value of an environment variable:

```
$ echo $DEVSH
```

Viewing and Editing Exercise Files

- Command-line editors

  Some students are comfortable using UNIX text editors like vi or nano. These can be run on the Linux command line to view and edit files as instructed in the exercises.

- Graphical editors

  If you prefer a graphical text editor, you can use gedit. You can start gedit using an icon from the remote desktop tool bar. (You can also use emacs if you prefer.)

Points to Note during the Exercises

Step-by-Step Instructions

As the exercises progress and you gain more familiarity with the tools and environment, we provide fewer step-by-step instructions; as in the real world, we merely give you a requirement and it is up to you to solve the problem!

If you need help, refer to the solutions provided, ask your instructor for assistance, or consult with your fellow students.

Bonus Exercises

There are additional challenges for some of the hands-on exercises. If you finish the main exercise, please attempt the additional steps.

Catch-Up Script

If you are unable to complete an exercise, there is a script to catch you up automatically. Each exercise has instructions for running the catch-up script if the exercise depends on completion of prior exercises.

```
$ $DEVSH/scripts/catchup.sh
```

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

Note: If you run the catch-up script, you may lose your work. For example, all exercise data will be deleted from HDFS before uploading the required files.

Troubleshooting

If you have trouble or unexpected behavior in the exercise environment, refer to the tips in Appendix: Troubleshooting Tips.

Hands-On Exercise: Starting the Exercise Environment

In this exercise, you will start your exercise environment and the cluster on which you will do the course exercises.

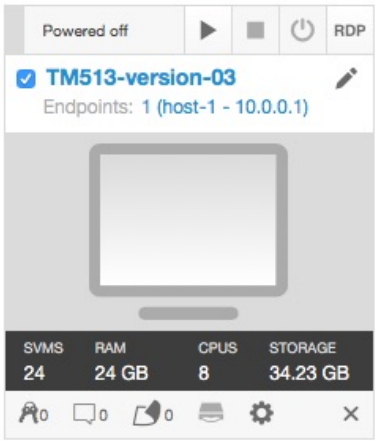
This course provides a single-host exercise environment running a pseudo-distributed Cloudera cluster (that is, a cluster running on a single host that emulates a multi-host environment) to complete the exercises. This section will walk you through starting the host in your environment, then starting the cluster within the environment.

Exercise Instructions

Start Your Exercise Environment

1. In your local browser, open the URL provided by your instructor to view the exercise environment portal.

2. The environment portal page displays a thumbnail image for your exercise environment remote host. The host will initially be powered off, indicated by a gray background. The icon background will change to green when the machine is running. (The machine name version may be different in your environment than the one below.)



   Click the host icon to open a new window showing the remote host machine. Start the machine by clicking the play button (triangle icon). It will take a few minutes to start.

3. When it is fully started, the remote host's desktop will display. These exercises refer to this as the "remote desktop" to distinguish it from your own local machine's desktop.

   The remainder of the exercises in the course will be performed on the remote desktop.

Start Your Cloudera Cluster

4. From the remote desktop's **Applications** menu, choose **Training > Start Cloudera Cluster**. A terminal window will open and the script to start the cluster services will run automatically.

5. Wait for about five minutes after the script finishes before continuing.

Verify Your Cluster Services

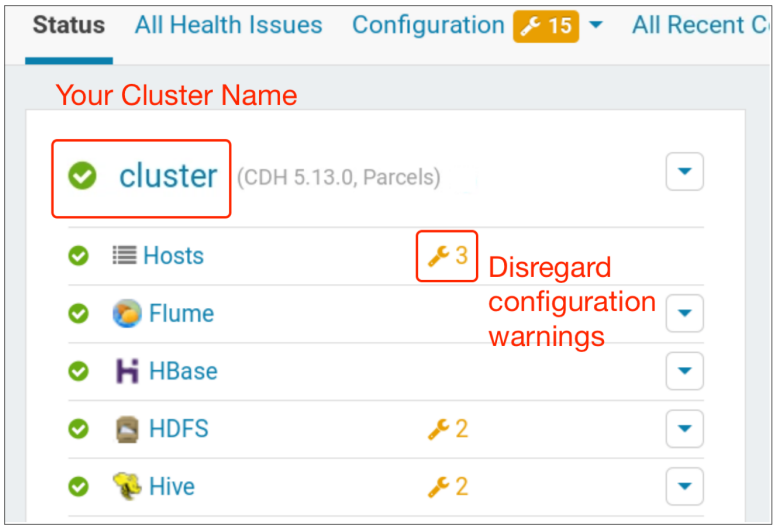
Confirm that your exercise environment is set up and the cluster services are running correctly by reviewing the cluster status using Cloudera Manager.

6. Start the Firefox browser using the icon on the remote desktop, then click the **Cloudera Manager** bookmark.

7. Log in as user **admin** with password **admin**.

8. Verify that all the services in your cluster are healthy (indicated by a green dot), as shown below. You may disregard yellow configuration warning icons.

   Note: *Some health warnings may appear as the cluster is starting. They will typically resolve themselves within five minutes after the Start Cluster script finishes. Please be patient.* If health issues remain, refer to the tip entitled "Cloudera Manager displays unhealthy status of services" in Appendix: Troubleshooting Tips.



   Your cluster may not be running exactly the services shown. This is okay and will not interfere with completing the exercises.

Your exercise environment is now ready.

See Managing Your Exercise Environment for instructions on stopping and restarting your environment.

Optional: Download and View the Exercise Manual on Your Remote Desktop

In order to be able to copy and paste from the Exercise Manual (the document you are currently viewing) to your remote desktop, you need to view the document on your remote machine rather than on your local machine.

9. Download the Exercise Manual

a. On your remote desktop, start the Firefox browser using the shortcut icon in the menu bar. The default page will display the Cloudera University home page. (You can return to this page at any time by clicking the home icon in Firefox or by visiting `https://university.cloudera.com/user/learning/enrollments`.)

b. Log in to your Cloudera University account by clicking **Returning Student Login**. Then from the Dashboard, find this course under **Current**.

c. Select the course title, then click to download the Exercise Manual under **Materials**. This will save the Exercise Manual PDF file in the `Downloads` folder in the training user's home directory.

10. View the Exercise Manual on Your Remote Host

a. Open a terminal window using the shortcut on the remote desktop, and then start the Evince PDF viewer:

```
$ evince &
```

b. In Evince, select menu item **File** > **Open** and open the Exercise Manual PDF file in the `Downloads` directory.

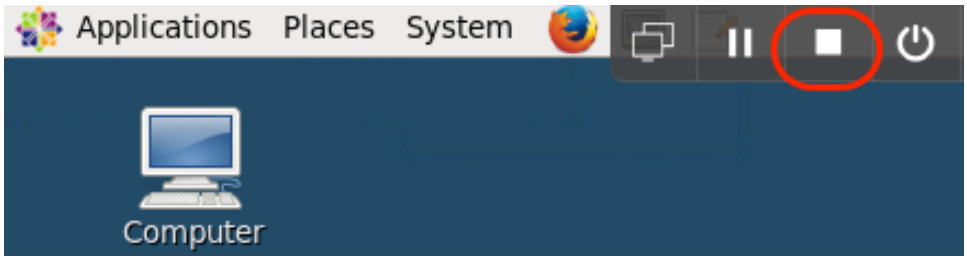
Managing Your Exercise Environment

Stopping Your Exercise Environment

During class, your remote host will stop automatically 90 minutes after the end of each day of class, and will automatically restart at the beginning of the next day.

After the class is over, you may continue using your environment for 10 hours of use or 30 days, whichever comes first. In order to minimize your usage of the limited time available, you should stop your environment whenever you are not using it, as described below.

In the toolbar at the top of the remote desktop display, click the stop button (a square).



Restarting Your Exercise Environment and Cluster

1. Restart your remote host.

   a. In your local browser, return to the exercise environment portal using the URL provided by your instructor.
   Click the remote host thumbnail to open the remote desktop, then click the play button to start the host.

2. Restart the cluster services.

   Whenever your exercise environment is stopped or suspended, you need to restart your cluster services after restarting the host.

   a. Close all terminal or browser windows on the remote desktop.

   b. From the desktop, select **Applications** > **Training** > **Start Cloudera Cluster** to restart your cluster services.

   c. Wait about five minutes after the script has completed, then verify that the services in your cluster services are running correctly by following the steps in Verify Your Cluster Services.

This is the end of the exercise.

Hands-On Exercise: Querying Hadoop Data with Apache Impala

Files and Data Used in This Exercise:	
Impala/Hive table	`accounts`

In this exercise, you will use the Hue Impala Query Editor to explore data in a Hadoop cluster.

This exercise is intended to let you begin to familiarize yourself with the course exercise environment as well as Hue. You will also briefly explore the Impala Query Editor.

Before starting this exercise, confirm that your cluster is running correctly by following the steps in Verify Your Cluster Services.

Log in to Hue

1. Start Firefox on the remote desktop using the shortcut provided on the main menu bar at the top of the screen.

2. View the Hue UI using the browser bookmark or visit `http://master-1:8889/`.

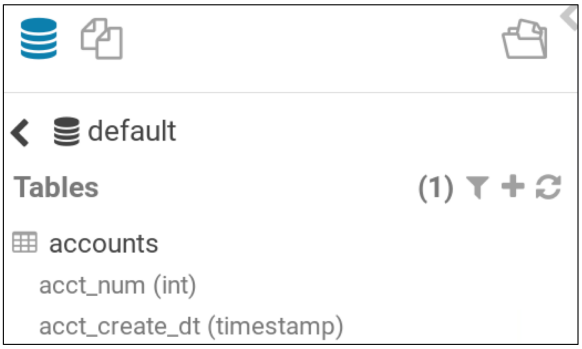
3. Because this is the first time anyone has logged in to Hue on this server, you will be prompted to create a new user account. Enter username **training** and password **training**, and then click **Create account**. When prompted, click **Remember**.

   Note: *Make sure to use this exact username and password. Your exercise environment is configured with a system user called* `training` *and your Hue username must match.* If you accidentally use the wrong username, refer to the instructions in the Appendix: Troubleshooting Tips section at the end of the exercises.

4. The first time you log in to Hue, you may see a welcome message with that offers a tour of new features in the latest version of Hue. The tour is optional. However, it is very short and you might find it valuable. To start the tour, click **Next**. To skip it, close the welcome popup using the **X** in the upper right corner.

View Table Columns and Details

5. The Hue default page should display the Impala query editor, with the `default` database and `accounts` displayed in the data source navigation panel on the left, as shown below.

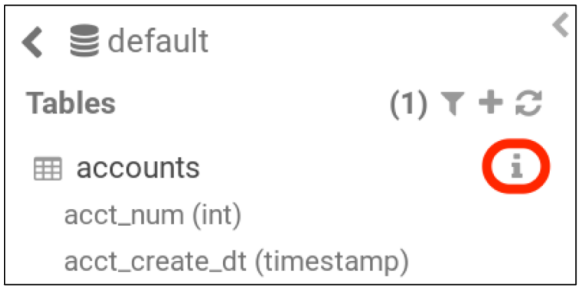


If a different data source is shown, click the SQL data source symbol.

Then use the back arrow symbol in the navigation panel (❮) and navigate to the **Impala** > **default** data source.

6. In the left panel under the `default` database, select the `accounts` table. This will display the table's column definitions.

7. Hover your pointer over the `accounts` table to reveal the associated **Show details** icon (labeled **i**), as shown below, then click the icon to bring up the details popup.



8. Select the **Sample** tab. The tab will display the first several rows of data in the table. When you are done viewing the data, click the **X** in the upper right corner of the popup to close it.

Query a Table Using Hue

9. Make sure the Impala query editor is displayed in the main panel by clicking the **Query** button.

10. In the query editor text box, enter a SQL query like the one below:

```
SELECT * FROM accounts WHERE first_name LIKE 'An%';
```

11. Click the **Execute** button (labeled as a blue "play" symbol: ▶) to execute the command.

12. View the returned data in the **Results** tab below the query area.

13. *Optional:* If you have extra time, continue exploring the Impala Query Editor on your own. For instance, try selecting other tabs after viewing the results.

This is the end of the exercise.

Hands-On Exercise: Accessing HDFS with the Command Line and Hue

Files and Data Used in This Exercise:

Data files (local)	`$DEVDATA/kb/` `$DEVDATA/base_stations.parquet`
Data files (HDFS)	`/user/hive/warehouse/accounts`

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.

Explore HDFS Using the Command Line Interface

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

1. Open a terminal window using the shortcut on the remote desktop menu bar.

2. In the new terminal session, use the HDFS command line to list the content of the HDFS root directory using the following command:

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

   There will be multiple entries, one of which is `/user`. Each user has a "home" directory under this directory, named after their username; your username in this course is **training**, therefore your home directory is `/user/training`.

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

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

   You will see your home directory in the directory listing.

> **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 filesystems.

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

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

Note that the directory structure in HDFS is unrelated to the directory structure of the local filesystem on the remote host; they are completely separate namespaces.

Upload 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.

5. 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
```

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

```
$ cd $DEVDATA
```

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

7. 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`.

8. 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.

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

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

View an HDFS File

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

```
$ 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 text data in HDFS. An individual file is often very large, making it inconvenient to view the entire file in the terminal. For this reason, it is often a good idea to pipe the output of the `dfs -cat` command into `head`, `more`, or `less`. You can also use `hdfs dfs -tail` to more efficiently view the end of the file, rather than piping the whole content.

Download an HDFS File

In an earlier exercise, you used Impala to explore data in HDFS in the `accounts` table. You can view and work with that data directly by downloading it from HDFS to the Linux local filesystem.

11. 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 Linux path. It copies the HDFS contents into the local filesystem:

```
$ hdfs dfs -get \
  /user/hive/warehouse/accounts /tmp/accounts
$ less /tmp/accounts/part-m-00000
```

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

View HDFS Command Line Help

12. 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 filesystem commands provided by HDFS.

Try experimenting with a few of these commands if you like.

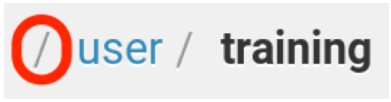
Use the Hue File Browser to Browse, View, and Manage Files

13. In the browser running on your remote desktop, visit Hue by clicking the **Hue** bookmark, or going to URL `http://master-1:8889/`.

If your prior session has expired, log in again using the login credentials you created earlier: username **training** and password **training**.

14. To access HDFS, open the triple bar menu in the upper left, then select **Browsers > Files**.

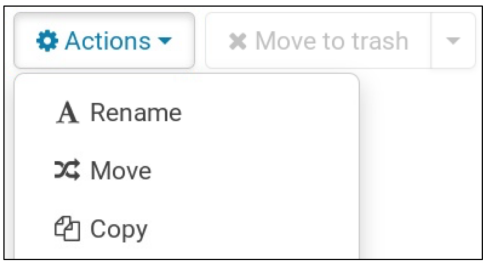
15. By default, the contents of your HDFS home directory (`/user/training`) are displayed. In the directory path name, click the leading slash (`/`) to view the HDFS root directory.



16. The contents of the root directory are displayed, including the `loudacre` directory you created earlier. Click that directory to see the contents.

17. Click the name of the `kb` directory to see the Knowledge Base articles you uploaded.

18. Click the checkbox next to any of the files, and then click the **Actions** button to see a list of actions that can be performed on the selected file(s).



19. View the contents of one of the files by clicking on the name of the file.

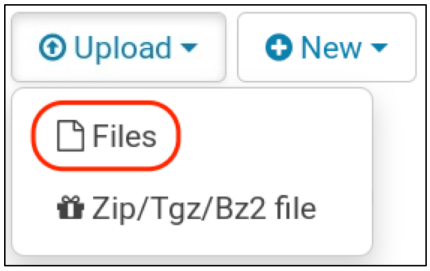
- **Note:** In the file viewer, the contents of the file are displayed on the right. In this case, the file is fairly small, but typical files in HDFS are very large, so rather than displaying the entire contents on one screen, Hue provides buttons to move between pages.

20. Return to the directory view by clicking **View file location** in the **Actions** panel on the left of the file contents.

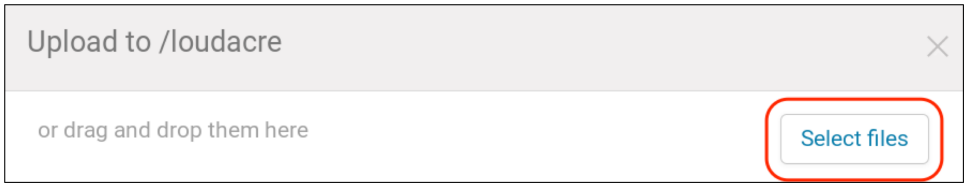
21. Click the up arrow to return to the `/loudacre` base directory.

22. Upload the `/home/training/training_materials/devsh/data/base_stations.parquet` file to the `/loudacre` HDFS directory.

a. Click the **Upload** button on the right. You can choose to upload a plain file, or to upload a zipped file (which will automatically be unzipped after upload). In this case, select **Files**.



b. Click **Select Files** to open a Linux file browser.



c. Browse to `/home/training/training_materials/devsh/data`, choose `base_stations.parquet`, and click the **Open** button.

d. Confirm that the file was correctly uploaded into the current directory.

23. *Optional:* Explore the various file actions available. When you have finished, select any additional files you have uploaded and click the **Move to trash** button to delete. (Do not delete `base_stations.parquet`; that file will be used in later exercises.)

This is the end of the exercise.

Hands-On Exercise: Running and Monitoring a YARN Job

Files and Data Used in This Exercise	
Exercise directory	`$DEVSH/exercises/yarn`
Data files (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 a previous 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.

Important: This exercise depends on a previous exercise: "Access HDFS with the Command Line and Hue." If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Explore the YARN Cluster

1. In Firefox , visit the YARN Resource Manager (RM) UI using the provided bookmark (labeled **RM**), or by going to URL `http://master-1:8088/`.

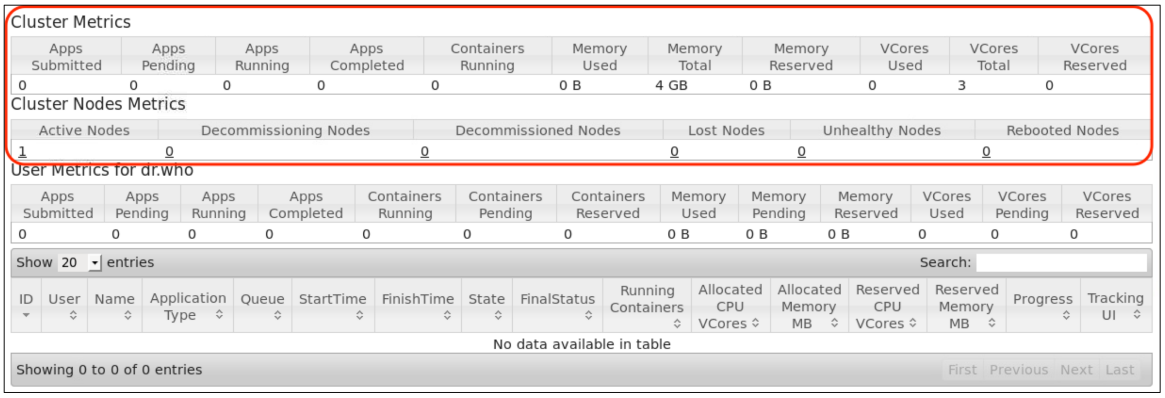
   No jobs are currently running so the current view shows the cluster "at rest."

   Who Is Dr. Who?

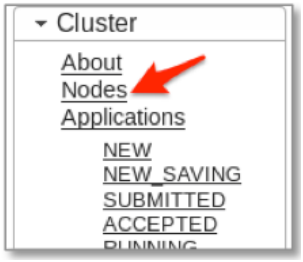
   You may notice that YARN says you are logged in as `dr.who`. This is what is displayed when user authentication is disabled for the cluster, as it is in the exercise environment. If user authentication was enabled, you would have to log in as a valid user to view the YARN UI, and your actual username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

2. Take note of the values in the **Cluster Metrics** section, which displays information 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.

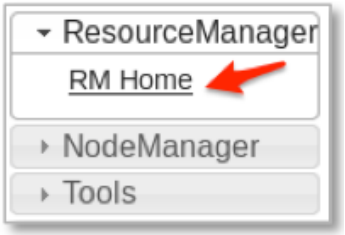


3. Click the **Nodes** link in the Cluster menu on the left. The bottom section will display a list of worker nodes in the cluster.



4. Click the `localhost.localdomain:8042` link under **Node HTTP Address** to open the Node Manager UI on that node. This displays statistics about the selected node, including the amount of available memory, currently running applications (there are none), and so on.

5. To return to the Resource Manager, click **ResourceManager** > **RM Home** on the left.



Submit an Application to the YARN Cluster

6. If you do not have an open terminal window, start one now.

7. In your terminal, run the example `wordcount.py` program on the YARN cluster to count the frequency of words in the Knowledge Base file set:

```
$ spark2-submit \
  $DEVSH/exercises/yarn/wordcount.py /loudacre/kb/*
```

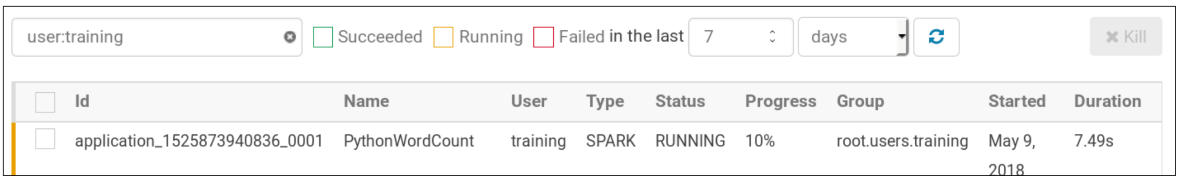
The `spark2-submit` command is used to submit a Spark program for execution on the cluster. Since Spark is managed by YARN on the course cluster, this gives us the opportunity to see how the YARN UI displays information about a running job. For now, focus on learning about the YARN UI.

While the application is running, continue with the next steps. If it completes before you finish the exercise, go to the terminal, press the up arrow until you get to the `spark2-submit` command again, and rerun the application.

View the Application in the Hue Job Browser

8. Go to Hue in the browser on your remote desktop and click the **Jobs** icon the Hue menu bar.

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



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 kill the job now though!)

View the Application in the YARN UI

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

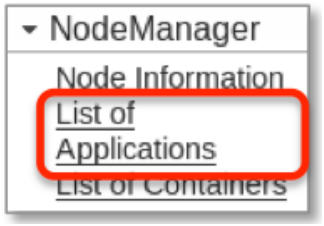
10. Reload the YARN RM page in Firefox. Notice that the application you just started is displayed in the list of applications in the bottom section of the RM home page.

ID	User	Name	Application Type	Queue	StartTime
application_1498225406752_0001	training	PythonWordCount	SPARK	root.users.training	Fri Jun 23 07:45:50 -0700

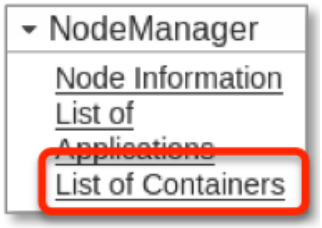
11. As you did in the first exercise section, select **Nodes**.

12. Select the node HTTP address link for localhost.localdomain to open the Node Manager UI .

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



14. 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. In the terminal window, use the up arrow key to recall previous commands.)

Show 20 entries Search:

ContainerId	ContainerState	logs
container_1498225406752_0006_01_000001	RUNNING	logs

Showing 1 to 1 of 1 entries

View the Application Using the `yarn` Command

15. Open a second terminal window on your remote desktop.

Tip: Resize the terminal window to be as wide as possible to make it easier to read the command output.

16. View the list of currently running applications.

```
$ yarn application -list
```

If your application is still running, you should see it listed, including the application ID (such as `application_1469799128160_0001`), the application name (`PythonWordCount`), the type (`SPARK`), and so on.

If there are no applications on the list, your application has probably finished running. By default, only current applications are included. Use the `-appStates ALL` option to include all applications in the list:

```
$ yarn application -list -appStates ALL
```

17. Take note of your application's ID (such as `application_1469799128160_0001`), and use it in place of *`app-id`* in the command below to get a detailed status report on the application.

```
$ yarn application -status app-id
```

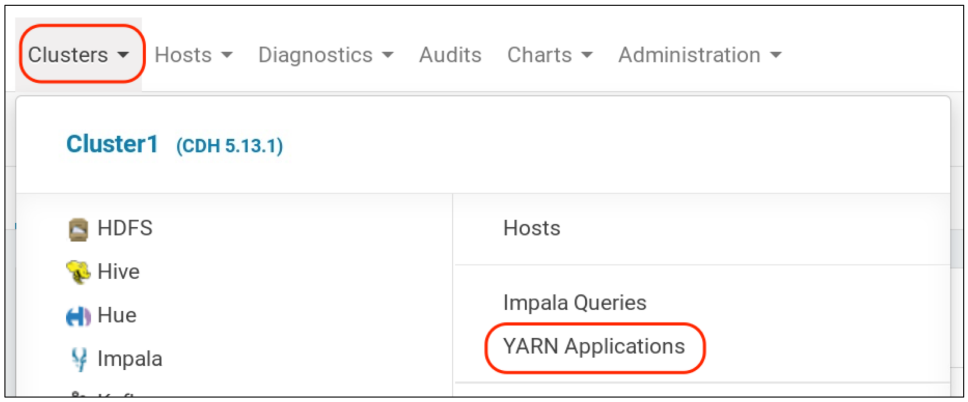
Bonus Exercise: View the Application in Cloudera Manager

If you have more time, attempt this extra bonus exercise.

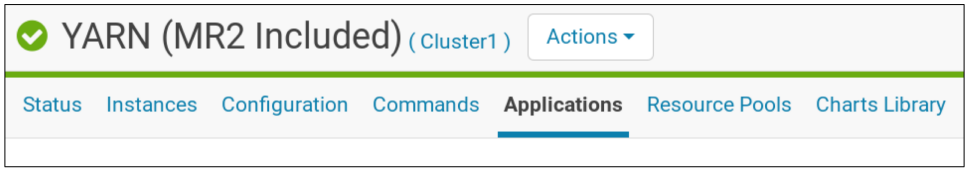
1. In Firefox , go to the Cloudera Manager UI using the provided bookmark.

2. Log into Cloudera Manager with the username **admin** and password **admin**.

3. On the Cloudera Manager home page, open the **Clusters** menu and select **YARN Applications**.



4. On the next page, you should be viewing the **Applications** tab.



Applications that are currently running or have recently run are shown. Confirm that the application you ran above is displayed in the list. (If your application has completed, you can restart it to explore the CM Applications manager working with a running application.)

This is the end of the exercise.

Hands-On Exercise: Exploring DataFrames Using the Apache Spark Shell

Files and Data Used in This Exercise	
Exercise directory	`$DEVSH/exercises/spark-shell`
Data files (local)	`$DEVDATA/devices.json`

In this exercise, you will use the Spark shell to work with DataFrames.

You will start by viewing and bookmarking the Spark documentation in your browser. Then you will start the Spark shell and read a simple JSON file into a DataFrame.

Important: This exercise depends on a previous exercise: "Access HDFS with Command Line and Hue." If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

View the Spark Documentation

1. View the Spark documentation in your web browser by visiting the URI `http://spark.apache.org/docs/2.2.0/`.

2. From the **Programming Guides** menu, select the **DataFrames, Datasets and SQL**. Briefly review the guide and 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 to this documentation.

4. If you are viewing the Scala API, notice that the package names are displayed on the left. Use the search box or scroll down to find the `org.apache.spark.sql` package. This package contains most of the classes and objects you will be working with in this course. In particular, note the `Dataset` class. Although this exercise focuses on DataFrames, remember that DataFrames are simply an alias for Datasets of Row objects. So all the DataFrame operations you will practice using in this exercise are documented on the `Dataset` class.

5. If you are viewing the Python API, locate the `pyspark.sql` module. This module contains most of the classes you will be working with in this course. At the top are some of the key classes in the module. View the API for the DataFrame class; these are the operations you will practice using in this exercise.

Start the Spark Shell

You may choose to do the remaining steps in this exercise using either Scala or Python.

> **Note on Spark Shell Prompt**
>
> To help you keep track of whether a Spark command is Python or Scala, the prompt will be shown here as either **`pyspark>`** or **`scala>`**. Some commands are the same for both Scala and Python. These will be shown with a **`>`** undesignated prompt. The actual prompt displayed in the shell will vary depending on which version of Python or Scala you are using and which command number you are on.

6. If you do not have an open terminal window, start one now.

7. In the terminal window, start the Spark 2 shell. Start either the Python shell or the Scala shell, not both.

   To start the Python shell, use the `pyspark2` command.

```
$ pyspark2
```

   To start the Scala shell, use the `spark2-shell` command.

```
$ spark2-shell
```

   You may get several `WARN` messages, which you can disregard.

8. Spark creates a `SparkSession` object for you called `spark`. Make sure the object exists. Use the first command below if you are using Python, and the second one if you are using Scala. (You only need to complete the exercises in Python *or* Scala.)

```
pyspark> spark
```

```
scala> spark
```

   Python will display information about the `spark` object such as:

   `<pyspark.sql.session.SparkSession at` *address*`>`

   Scala will display similar information in a different format:

```
org.apache.spark.sql.SparkSession =
org.apache.spark.sql.SparkSession@address
```

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>`.

9. Using command completion, you can see all the available Spark session methods: type `spark.` (`spark` followed by a dot) and then the `TAB` key.

   Note: You can exit the Scala shell by typing `sys.exit`. To exit the Python shell, press `Ctrl+D` or type `exit`. However, stay in the shell for now to complete the remainder of this exercise.

Read and Display a JSON File

10. Open a new terminal window (in addition to the terminal running the Spark shell).

11. Review the simple text file you will be using: `$DEVDATA/devices.json`. You can view the file either in gedit, or by starting a new terminal window then using the `less` command. (Do not modify the file.) This file contains records for each of Loudacre's supported devices. For example:

```
{"devnum":1,"release_dt":"2008-10-21T00:00:00.000-07:00",
    "make":"Sorrento","model":"F00L","dev_type":"phone"}
```

   Notice the field names and types of values in the first few records.

12. Upload the data file to the `/loudacre` directory in HDFS:

```
$ hdfs dfs -put $DEVDATA/devices.json /loudacre/
```

13. In the Spark shell, create a new DataFrame based on the `devices.json` file in HDFS.

```
pyspark> devDF = spark.read. \
  json("/loudacre/devices.json")
```

```
scala> val devDF = spark.read.
  json("/loudacre/devices.json")
```

14. Spark has not yet read the data in the file, but it has scanned the file to infer the schema. View the schema, and note that the column names match the record field names in the JSON file.

```
pyspark> devDF.printSchema()
```

```
scala> devDF.printSchema
```

15. Display the data in the DataFrame using the `show` function. If you don't pass an argument to `show`, Spark will display the first 20 rows in the DataFrame. For this step, display the first five rows. Note that the data is displayed in tabular form, using the column names defined in the schema.

```
> devDF.show(5)
```

Note: Like many Spark queries, this command is the same whether you are using Scala or Python.

16. The `show` and `printSchema` operations are actions—that is, they return a value from the distributed DataFrame to the Spark driver. Both functions display the data in a nicely formatted table. These functions are intended for interactive use in the shell, but do not allow you actually work with the data that is returned. Try using the `take` action instead, which returns an array (Scala) or list (Python) of `Row` objects. You can display the data by iterating through the collection.

```
pyspark> rows = devDF.take(5)
pyspark> for row in rows: print row
```

```
scala> val rows = devDF.take(5)
scala> rows.foreach(println)
```

Query a DataFrame

17. Use the `count` action to return the number of items in the DataFrame.

```
> devDF.count()
```

18. DataFrame transformations typically return another DataFrame. Try using a `select` transformation to return a DataFrame with only the `make` and `model`

columns, then display its schema. Note that only the selected columns are in the schema.

```
pyspark> makeModelDF = devDF.select("make","model")
pyspark> makeModelDF.printSchema()
```

```
scala> val makeModelDF = devDF.select("make","model")
scala> makeModelDF.printSchema
```

19. A query is a series of one or more transformations followed by an action. Spark does not execute the query until you call the action operation. Display the first 20 lines of the final DataFrame in the series using the `show` action.

```
pyspark> makeModelDF.show()
```

```
scala> makeModelDF.show
```

20. Transformations in a query can be chained together. Execute a single command to show the results of a query using `select` and `where`. The resulting DataFrame will contain only the columns `devnum`, `make`, and `model`, and only the rows where the make is `Ronin`.

```
pyspark> devDF.select("devnum","make","model"). \
  where("make = 'Ronin'"). \
  show()
```

```
scala> devDF.select("devnum","make","model").
  where("make = 'Ronin'").
  show
```

This is the end of the exercise.

Hands-On Exercise: Working with DataFrames and Schemas

Files and Data Used in This Exercise:	
Exercise directory	`$DEVSH/exercises/dataframes`
Data files (HDFS)	`/loudacre/devices.json`
Hive Tables	`accounts`

In this exercise, you will work with structured account and mobile device data using DataFrames.

You will practice creating and saving DataFrames using different types of data sources, and inferring and defining schemas.

Important: This exercise depends on a previous exercise: "Exploring DataFrames Using the Spark Shell." If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Create a DataFrame Based on a Hive Table

1. This exercise uses a DataFrame based on the `accounts` Hive table. Before you start working in Spark, visit Hue in the web browser on your remote desktop and use the Impala Query Editor to review the schema and data of the `accounts` table in the default database.

2. If you do not have one already, open a terminal , and start the Spark 2 shell (either Scala or Python, as you prefer).

3. Create a new DataFrame using the Hive `accounts` table.

```
pyspark> accountsDF = spark.read.table("accounts")
```

```
scala> val accountsDF = spark.read.table("accounts")
```

4. Print the schema and the first few rows of the DataFrame, and note that the schema and data are the same as the Hive table.

5. Create a new DataFrame with rows from the accounts data where the zip code is 94913, and save the result to CSV files in the `/loudacre/accounts_zip94913`

HDFS directory. You can do this in a single command, as shown below, or with multiple commands.

```
pyspark> accountsDF.where("zipcode = 94913"). \
  write.option("header","true"). \
  csv("/loudacre/accounts_zip94913")
```

```
scala> accountsDF.where("zipcode = '94913'").
  write.option("header","true").
  csv("/loudacre/accounts_zip94913")
```

6. Use Hue or the command line (in a separate terminal window) to view the `/loudacre/accounts_zip94913` directory in HDFS and the data in one of the saved files. Confirm that the CSV file includes a header line, and that only records for the selected zip code are included.

7. *Optional*: Try creating a new DataFrame based on the CSV files you created above. Compare the schema of the original `accountsDF` and the new DataFrame. What's different? Try again, this time setting the `inferSchema` option to `true` and compare again.

Define a Schema for a DataFrame

8. If you have not done so yet, review the data in the HDFS file `/loudacre/devices.json`.

9. Create a new DataFrame based on the `devices.json` file. (This command could take several seconds while it infers the schema.)

```
pyspark> devDF = spark.read. \
  json("/loudacre/devices.json")
```

```
scala> val devDF = spark.read.
  json("/loudacre/devices.json")
```

10. View the schema of the `devDF` DataFrame. Note the column names and types that Spark inferred from the JSON file. In particular, note that the `release_dt` column is of type `string`, whereas the data in the column actually represents a timestamp.

11. Define a schema that correctly specifies the column types for this DataFrame. Start by importing the package with the definitions of necessary classes and types.

```
pyspark> from pyspark.sql.types import *
```

```
scala> import org.apache.spark.sql.types._
```

12. Next, create a collection of `StructField` objects, which represent column definitions. The `release_dt` column should be a timestamp.

```
pyspark> devColumns = [
  StructField("devnum",LongType()),
  StructField("make",StringType()),
  StructField("model",StringType()),
  StructField("release_dt",TimestampType()),
  StructField("dev_type",StringType())]
```

```
scala> val devColumns = List(
  StructField("devnum",LongType),
  StructField("make",StringType),
  StructField("model",StringType),
  StructField("release_dt",TimestampType),
  StructField("dev_type",StringType))
```

13. Create a schema (a `StructType` object) using the column definition list.

```
pyspark> devSchema = StructType(devColumns)
```

```
scala> val devSchema = StructType(devColumns)
```

14. Recreate the `devDF` DataFrame, this time using the new schema.

```
pyspark> devDF = spark.read. \
  schema(devSchema).json("/loudacre/devices.json")
```

```
scala> val devDF = spark.read.
  schema(devSchema).json("/loudacre/devices.json")
```

15. View the schema and data of the new DataFrame, and confirm that the `release_dt` column type is now `timestamp`.

16. Now that the device data uses the correct schema, write the data in Parquet format, which automatically embeds the schema. Save the Parquet data files into an HDFS directory called `/loudacre/devices_parquet`.

17. *Optional:* In a separate terminal window, use `parquet-tools` to view the schema of the saved files.

```
$ parquet-tools schema \
  hdfs://master-1/loudacre/devices_parquet/
```

    Note that the type of the `release_dt` column is noted as `int96`; this is how Spark denotes a timestamp type in Parquet.

    For more information about `parquet-tools`, run `parquet-tools --help`.

18. Create a new DataFrame using the Parquet files you saved in `devices_parquet` and view its schema. Note that Spark is able to correctly infer the `timestamp` type of the `release_dt` column from Parquet's embedded schema.

This is the end of the exercise.

Hands-On Exercise: Analyzing Data with DataFrame Queries

Files and Data Used in This Exercise:	
Exercise directory	`$DEVSH/exercises/analyze`
Data files (local)	`$DEVDATA/accountdevice`
Data files (HDFS)	`/loudacre/devices.json` `/loudacre/base_stations.parquet`
Hive Tables	`accounts`

In this exercise, you will analyze account and mobile device data using DataFrame queries.

First, you will practice using column expressions in queries. You will analyze data in DataFrames by grouping and aggregating data, and by joining two DataFrames. Then you will query multiple sets of data to find out how many of each mobile device model is used in active accounts.

Important: This exercise depends on a previous exercise: "Working with DataFrames and Schemas." If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Query DataFrames Using Column Expressions

1. *Optional*: Review the API docs for the `Column` class (which is in the Python module `pyspark.sql`, and the Scala package `org.apache.spark.sql`). Take note of the various options available.

2. Start the Spark 2 shell in a terminal if you do not already have it running.

3. Create a new DataFrame called `accountsDF` based on the Hive `accounts` table.

4. Try a simple query with `select`, using both column reference syntaxes.

```
pyspark> accountsDF. \
  select(accountsDF["first_name"]).show()
pyspark> accountsDF.select(accountsDF.first_name).show()
```

```
scala> accountsDF.
  select(accountsDF("first_name")).show
scala> accountsDF.select($"first_name").show
```

5. To explore column expressions, create a column object to work with, based on the `first_name` column in the `accountsDF` DataFrame.

```
pyspark> fnCol = accountsDF.first_name
```

```
scala> val fnCol = accountsDF("first_name")
```

6. Note that the object type is `Column`. To see available methods and attributes, use tab completion—that is, enter `fnCol.` followed by `TAB`.

7. New `Column` objects are created when you perform operations on existing columns. Create a new `Column` object based on a column expression that identifies users whose first name is Lucy using the equality operator on the `fnCol` object you created above.

```
pyspark> lucyCol = (fnCol == "Lucy")
```

```
scala> val lucyCol = (fnCol === "Lucy")
```

8. Use the `lucyCol` column expression in a `select` statement. Because `lucyCol` is based on a boolean expression, the column values will be `true` or `false`

depending on the value of the `first_name` column. Confirm that users named Lucy are identified with the value `true`.

```
pyspark> accountsDF. \
  select(accountsDF.first_name, \
         accountsDF.last_name,lucyCol). \
  show()
```

```
scala> accountsDF.
  select($"first_name",$"last_name",lucyCol).show
```

9. The `where` operation requires a boolean-based column expression. Use the `lucyCol` column expression in a `where` transformation and view the data in the resulting DataFrame. Confirm that only users named Lucy are in the data.

```
> accountsDF.where(lucyCol).show(5)
```

10. Column expressions do not need to be assigned to a variable. Try the same query without using the `lucyCol` variable.

```
pyspark> accountsDF.where(fnCol == "Lucy").show(5)
```

```
scala> accountsDF.where(fnCol === "Lucy").show(5)
```

11. Column expressions are not limited to `where` operations like those above. They can be used in any transformation for which a simple column could be used, such as a `select`. Try selecting the `city` and `state` columns, and the first three characters of the `phone_number` column (in the U.S., the first three digits of a phone number

are known as the area code). Use the `substr` operator on the `phone_number` column to extract the area code.

```
pyspark> accountsDF. \
  select("city", "state", \
    accountsDF.phone_number.substr(1,3)). \
  show(5)
```

```
scala> accountsDF.
  select($"city", $"state",
    $"phone_number".substr(1,3)).
  show(5)
```

12. Notice that in the last step, the values returned by the query were correct, but the column name was `substring(phone_number, 1, 3)`, which is long and hard to work with. Repeat the same query, using the alias operator to rename that column as `area_code`.

```
pyspark> accountsDF. \
  select("city", "state", \
    accountsDF.phone_number. \
    substr(1,3).alias("area_code")). \
  show(5)
```

```
scala> accountsDF.
  select($"city", $"state",
    $"phone_number".substr(1,3).alias("area_code")).
  show(5)
```

13. Perform a query that results in a DataFrame with just `first_name` and `last_name` columns, and only includes users whose first and last names both begin with the same two letters. (For example, the user Robert Roget would be included, because both his first and last names begin with "Ro".)

Group and Count Data by Name

14. Query the `accountsDF` DataFrame using `groupBy` with count to find out the total number people sharing each last name. (Note that the `count` aggregation

transformation returns a DataFrame, unlike the `count` DataFrame action, which returns a single value to the driver.)

```
pyspark> accountsDF.groupBy("last_name").count().show(5)
```

```
scala> accountsDF.groupBy("last_name").count.show(5)
```

15. You can also group by multiple columns. Query `accountsDF` again, this time counting the number of people who share the same last and first name.

```
pyspark> accountsDF. \
  groupBy("last_name","first_name").count().show(5)
```

```
scala> accountsDF.
  groupBy("last_name","first_name").count.show(5)
```

Join Account Data with Cellular Towers by Zip Code

16. In this section, you will join the accounts data that you have been using with data about cell tower base station locations, which is in the `base_stations.parquet` file in the HDFS `/loudacre` directory. Start by reviewing the schema and a few records of the data. Use the `parquet-tools` command in a separate terminal window (not the one running the Spark shell).

```
$ parquet-tools schema \
  hdfs://master-1/loudacre/base_stations.parquet
$ parquet-tools head \
  hdfs://master-1/loudacre/base_stations.parquet
```

17. In your Spark shell, create a new DataFrame called `baseDF` using the base stations data. Review the `baseDF` schema and data to ensure it matches the data in the Parquet file.

18. Some account holders live in zip codes that have a base station. Join `baseDF` and `accountsDF` to find those users, and for each, include their account ID, first name, last name, and the ID and location data for the base station in their zip code.

```
pyspark> accountsDF. \
  select("acct_num","first_name","last_name","zipcode"). \
  join(baseDF, baseDF.zip == accountsDF.zipcode). \
  show()
```

```
scala> accountsDF.
  select("acct_num","first_name","last_name","zipcode").
  join(baseDF,$"zip" === $"zipcode").show()
```

Count Active Devices

19. The `accountdevice` CSV data files contain data lists all the devices used by all the accounts. Each row in the data set includes a row ID, an account ID, a device ID for the type of device, the date the device was activated for the account, and the specific device's ID for that account.

The CSV data files are in the `$DEVDATA/accountdevice` directory . Review the data in the data set, then upload the directory and its contents to the HDFS directory `/loudacre/accountdevice`.

20. Create a DataFrame based on the `accountdevice` data files.

21. Use the account device data and the DataFrames you created previously in this exercise to find the total number of each device model across all *active* accounts (that is, accounts that have not been closed). The new DataFrame should be sorted from most to least common model. Save the data as Parquet files in a directory called `/loudacre/top_devices` with the following columns:

Column Name	Description	Example Value
`device_id`	The ID number of each known device (including those that might not be in use by any account)	`18`
`make`	The manufacturer name for the device	`Ronin`
`model`	The model name for the device	`Novelty Note 2`
`active_num`	The total number of the model used by active accounts	`2092`

Hints:

- Active accounts are those with a null value for `acct_close_dt` (account close date) in the `accounts` table.
- The `account_id` column in the device accounts data corresponds to the `acct_num` column in `accounts` table.
- The `device_id` column in the device accounts data corresponds to the `devnum` column in the list of known devices in the `/loudacre/devices.json` file.
- When you count devices, use `withColumnRenamed` to rename the `count` column to `active_num`. (The `count` column name is ambiguous because it is both a function and a column.)
- The query to complete this exercise is somewhat complicated and includes a sequence of many transformations. You may wish to assign variables to the intermediate DataFrames resulting from the transformations that make up the query to make the code easier to work with and debug.

This is the end of the exercise.

Hands-On Exercise: Working With RDDs

Files and Data Used in This Exercise	
Exercise directory	`$DEVSH/exercises/rdds`

Data files (local)	`$DEVDATA/frostroad.txt`
	`$DEVDATA/makes1.txt`
	`$DEVDATA/makes2.txt`

In this exercise, you will use the Spark shell to work with RDDs.

You will start reading a simple text file into a Resilient Distributed Dataset (RDD) and displaying the contents. You will then create two new RDDs and use transformations to union them and remove duplicates.

Important: This exercise depends on a previous exercise: "Accessing HDFS with Command Line and Hue." If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Review the API Documentation for RDD Operations

1. Review the API docs for the RDD class (which is in the Python module `pyspark`, and the Scala package `org.apache.spark.rdd`). Take note of the various available operations.

Read and Display Data from a Text File

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

3. In a terminal window on your remote desktop, upload the text file to HDFS directory `/loudacre`.

```
$ hdfs dfs -put $DEVDATA/frostroad.txt /loudacre/
```

4. In the Spark shell, define an RDD based on the `frostroad.txt` text file.

```
pyspark> myRDD = sc. \
  textFile("/loudacre/frostroad.txt")
```

```
scala> val myRDD = sc.
  textFile("/loudacre/frostroad.txt")
```

5. Using command completion, you can see all the available transformations and actions you can perform on an RDD. Type `myRDD.` and then the `TAB` key.

6. Spark has not yet read the file. It will not do so until you perform an action on the RDD. Try counting the number of elements in the RDD using the `count` action:

```
pyspark> myRDD.count()
```

```
scala> myRDD.count
```

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

   `Out[2]: 23` (Python) *or*
   `res1: Long = 23` (Scala)

7. Call the `collect` operation to return all data in the RDD to the Spark driver. Take note of the type of the return value; in Python will be a list of strings, and in Scala it will be an array of strings.

   Note: `collect` returns the entire set of data. This is convenient for very small RDDs like this one, but be careful using `collect` for more typical large sets of data.

```
pyspark> lines = myRDD.collect()
```

```
scala> val lines = myRDD.collect
```

8. Display the contents of the collected data by looping through the collection.

```
pyspark> for line in lines: print line
```

```
scala> for (line <- lines) println(line)
```

Transform Data in an RDD

9. In this exercise, you will load two text files containing the names of various cell phone makes, and append one to the other. Review the two text files you will be using by viewing (without editing) the file in a separate window. The files are `makes1.txt` and `makes2.txt` in the `$DEVDATA` directory

10. Upload the two text file to HDFS directory `/loudacre`.

```
$ hdfs dfs -put $DEVDATA/makes*.txt /loudacre/
```

11. In Spark, create an RDD called `makes1RDD` based on the `/loudacre/makes1.txt` file.

```
pyspark> makes1RDD = sc.textFile("/loudacre/makes1.txt")
```

```
scala> val makes1RDD = sc.textFile("/loudacre/makes1.txt")
```

12. Display the contents of the `makes1RDD` data using `collect` and then looping through returned collection.

```
pyspark> for make in makes1RDD.collect(): print make
```

```
scala> for (make <- makes1RDD.collect()) println(make)
```

13. Repeat the previous steps to create and display an RDD called `makes2RDD` based on the second file, `/loudacre/makes2.txt`.

14. Create a new RDD by appending the second RDD to the first using the `union` transformation.

```
pyspark> allMakesRDD = makes1RDD.union(makes2RDD)
```

```
scala> val allMakesRDD = makes1RDD.union(makes2RDD)
```

15. Collect and display the contents of the new `allMakesRDD` RDD.

16. Use the `distinct` transformation to remove duplicates from `allMakesRDD`. Collect and display the contents to confirm that duplicate elements were removed.

17. *Optional*: Try performing different transformations on the RDDs you created above, such as `intersection`, `subtract`, or `zip`. See the RDD API documentation for details.

This is the end of the exercise.

Hands-On Exercise: Transforming Data Using RDDs

Files and Data Used in This Exercise	
Exercise directory	`$DEVSH/exercises/transform-rdds`
Data files (local)	`$DEVDATA/weblogs/` `$DEVDATA/devicestatus.txt`

In this exercise, you will transform data in RDDs.

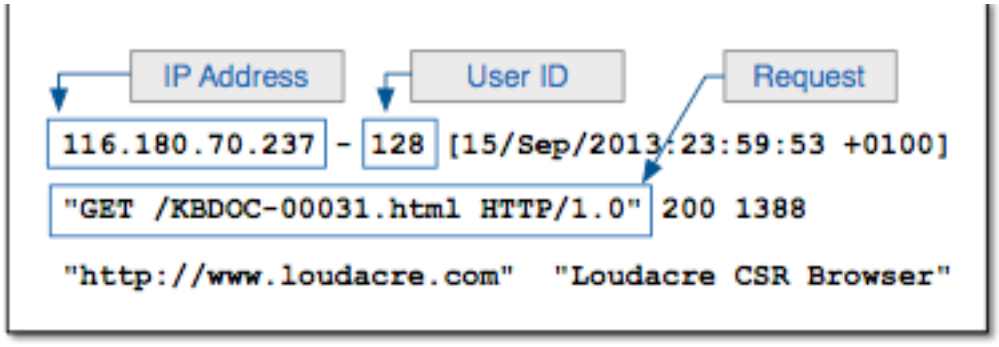
You will start reading a simple text file into a Resilient Distributed Dataset (RDD). Then you create an RDD based on Loudacre's website log data and practice transforming the data.

Important: This exercise depends on a previous exercise: "Accessing HDFS with Command Line and Hue." If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Explore the Loudacre Web Log Files

1. In this section you will be using data in `$DEVDATA/weblogs`. Review one of the `.log` files in the directory. Note the format of the lines:



2. Copy the `weblogs` directory from the local filesystem to the `/loudacre` HDFS directory.

```
$ hdfs dfs -put $DEVDATA/weblogs /loudacre/
```

3. In spark, create an RDD from the uploaded web logs data files in the `/loudacre/weblogs/` directory in HDFS.

```
pyspark> logsRDD = sc.textFile("/loudacre/weblogs/")
```

```
scala> val logsRDD = sc.textFile("/loudacre/weblogs/")
```

4. Create an RDD containing only those lines that are requests for JPG files. Use the `filter` operation with a transformation function that takes a string RDD element and returns a boolean value.

```
pyspark> jpglogsRDD = \
  logsRDD.filter(lambda line: ".jpg" in line)
```

```
scala> val jpglogsRDD =
  logsRDD.filter(line => line.contains(".jpg"))
```

5. Use `take` to return the first five lines of the data in `jpglogsRDD`. The return value is a list of strings (in Python) or array of strings (in Scala).

```
pyspark> jpgLines = jpglogsRDD.take(5)
```

```
scala> val jpgLines = jpglogsRDD.take(5)
```

6. Loop through and display the strings returned by `take`.

```
pyspark> for line in jpgLines: print line
```

```
scala> jpgLines.foreach(println)
```

7. Now try using the `map` transformation to define a new RDD. Start with a simple map function that returns the length of each line in the log file. This results in an RDD of integers.

```
pyspark> lineLengthsRDD = \
  logsRDD.map(lambda line: len(line))
```

```
scala> val lineLengthsRDD =
  logsRDD.map(line => line.length)
```

8. Loop through and display the first five elements (integers) in the RDD.

9. Calculating line lengths is not very useful. Instead, try mapping each string in `logsRDD` by splitting the strings based on spaces. The result will be an RDD in which each element is a list of strings (in Python) or an array of strings (in Scala). Each string represents a "field" in the web log line.

```
pyspark> lineFieldsRDD = \
  logsRDD.map(lambda line: line.split(' '))
```

```
scala> val lineFieldsRDD =
  logsRDD.map(line => line.split(' '))
```

10. Return the first five elements of `lineFieldsRDD`. The result will be a list of lists of strings (in Python) or an array of arrays of strings (in Scala).

```
pyspark> lineFields = lineFieldsRDD.take(5)
```

```
scala> val lineFields = lineFieldsRDD.take(5)
```

11. Display the contents of the return from `take`. Unlike in examples above, which returned collections of simple values (strings and ints), this time you have a set of compound values (arrays or lists containing strings). Therefore, to display them properly, you will need to loop through the arrays/lists in `lineFields`, and then loop through each string in the array/list. To make it easier to read the output, use `-------` to separate each set of field values.

If you choose to copy and paste the Pyspark code below into the shell, it may not automatically indent properly; be sure the indentation is correct before executing the command.

```
pyspark> for fields in lineFields:
  print "-------"
  for field in fields: print field
```

```
scala> for (fields <- lineFields) {
  println("-------")
  fields.foreach(println)
}
```

12. Now that you know how `map` works, create a new RDD containing just the IP addresses from each line in the log file. (The IP address is the first space-delimited field in each line.)

```
pyspark> ipsRDD = \
  logsRDD.map(lambda line: line.split(' ')[0])
pyspark> for ip in ipsRDD.take(5): print ip
```

```
scala> val ipsRDD =
  logsRDD.map(line => line.split(' ')(0))
scala> ipsRDD.take(5).foreach(println)
```

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

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

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

- **Note:** If you re-run this command, you will not be able to save to the same directory because it already exists. Be sure to first delete the directory using either the `hdfs` command (in a separate terminal window) or the Hue file browser.

14. In a separate terminal window or the Hue file browser, list the contents of the `/loudacre/iplist` folder. Review the contents of one of the files to confirm that they were created correctly.

Map weblog entries to IP address/user ID pairs

15. Use RDD transformations to create a dataset consisting of the IP address and corresponding user ID for each request for an HTML file. (Filter for files with the `.html` extension; disregard requests for other file types.) The user ID is the third field in each log file line. Save the data into a comma-separated text file in the directory `/loudacre/userips_csv`. Make sure the data is saved in the form of comma-separated strings:

```
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
…
```

16. Now that the data is in CSV format, it can easily be used by Spark SQL. Load the new CSV files in `/loudacre/userips_csv` created above into a DataFrame, then view the data and schema.

Bonus Exercise 1: Clean device status data

If you have more time, attempt this extra bonus exercise.

One common use of core Spark RDDs is data scrubbing—converting the data into a format that can be used in Spark SQL. 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 `$DEVDATA/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 providers' 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 the following:

17. Upload the `devicestatus.txt` file to HDFS.

18. Determine which delimiter to use (the 20th character—position 19—is the first use of the delimiter).

19. Filter out any records which do not parse correctly (hint: each record should have exactly 14 values).

20. Extract the date (first field), model (second field), device ID (third field), and latitude and longitude (13th and 14th fields respectively).

21. 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`). Keep just the manufacturer name.

22. Save the extracted data to comma-delimited text files in the `/loudacre/devicestatus_etl` directory on HDFS.

23. Confirm that the data in the file(s) was saved correctly. The lines in the file should all look similar to this, with all fields delimited by commas.

```
2014-03-15:10:10:20,Sorrento,8cc3b47e-bd01-4482-
b500-28f2342679af,33.6894754264,-117.543308253
```

The solutions to the bonus exercise are in `$DEVSH/exercises/transform-rdds/bonus-dataclean/solution`.

Bonus Exercise 2: Convert Multi-line XML files to CSV files

One of the common uses for RDDs in core Spark is to transform data from unstructured or semi-structured sources or formats that aren't supported by Spark SQL to structured formats you can use with Spark SQL. In this bonus exercise, you will convert a whole-file XML record to a CSV file that can be read into a DataFrame.

24. Review the data on the local Linux filesystem in the directory `$DEVDATA/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>
```

25. Copy the entire `activations` directory to `/loudacre` in HDFS.

```
$ hdfs dfs -put $DEVDATA/activations /loudacre/
```

Follow the steps below to write code to 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
…
```

26. Start with the `ActivationModels` stub script in the bonus exercise directory: `$DEVSH/exercises/transform-rdds/bonus-xml`. (Stubs are 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 of your choice.

27. 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.

28. 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.

29. 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.

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

The solutions to the bonus exercise are in `$DEVSH/exercises/transform-rdds/bonus-xml/solution`.

This is the end of the exercise.

Hands-On Exercise: Joining Data Using Pair RDDs

Files and Data Used in This Exercise:	
Exercise directory	`$DEVSH/exercises/pair-rdds`
Data files (HDFS)	`/loudacre/weblogs` `/user/hive/warehouse/accounts`

In this exercise, you will explore the Loudacre web server log files, as well as the Loudacre user account data, using key-value pair RDDs.

Important: This exercise depends on a previous exercise: "Transforming Data Using RDDs." If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Explore Web Log Files

Create a pair RDD based on data in the `weblogs` data files, and use the pair RDD to explore the data.

Tip: In this exercise, you will be reducing and joining large datasets, which can take a lot of time and may result in memory errors resulting from the limited resources available in the course exercise environment. Perform these exercises with a subset of the the web log files by using a wildcard: `textFile("/loudacre/weblogs/*2.log")` includes only filenames ending with `2.log`.

1. Using map-reduce logic, 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 this:

   ```
   (userid,5)
   ```

(*userid*,7)
(*userid*,2)
…

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

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

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

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

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

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

(*userid*,20.1.34.55)
(*userid*,245.33.1.1)
(*userid*,65.50.196.141)
…

(*userid*,[20.1.34.55, 74.125.239.98])
(*userid*,[75.175.32.10, 245.33.1.1, 66.79.233.99])
(*userid*,[65.50.196.141])
…

Join Web Log Data with Account Data

Review the accounts data located in `/user/hive/warehouse/accounts`, which contains the data in the Hive `accounts` table. 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…])

```
(9012,[9012,2008-11-24 10:04:08,\N,Cheryl,West, 4905 Olive Street,…])
(2312,[2312,2008-11-23 14:05:07,\N,Elizabeth,Kerns, 4703 Eva Pearl Street,Richmond,CA,…])
(1195,[1195,2008-11-02 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.

```
(9012,([9012,2008-11-24 10:04:08,\N,Cheryl,West, 4905 Olive Street,San Francisco,CA,…],4))
(2312,([2312,2008-11-23 14:05:07,\N,Elizabeth,Kerns, 4703 Eva Pearl Street,Richmond,CA,…],8))
(1195,([1195,2008-11-02 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 five elements. The output should look similar to this:

```
9012 6 Rick Hopper
1123 8 Lucio Arnold
1093 2 Brittany Parrott
…
```

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: Querying Tables and Views with SQL

Files and Data Used in This Exercise:	
Exercise directory	`$DEVSH/exercises/spark-sql`
Data files (local)	`$DEVDATA/accountdevice`
Hive Tables	`accounts`

In this exercise, you will use the Catalog API to explore Hive tables, and create DataFrames by executing SQL queries.

Use the Catalog API to list the tables in the default Hive database, and view the schema of the `accounts` table. Perform queries on the accounts table, and review the resulting DataFrames. Create a temporary view based on the `accountdevice` CSV files, and use SQL to join that table with the `accounts` table.

Important: This exercise depends on a previous exercise: "Analyzing Data with DataFrame Queries." If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Show tables and columns using the Catalog API

1. View the list of current Hive tables and temporary views in the default database.

```
pyspark> for table in spark.catalog.listTables():
  print table
```

```
scala> spark.catalog.listTables.show
```

The list should include the `accounts` table.

2. List the schema (column definitions) of the `accounts` table.

```
pyspark> for column in \
 spark.catalog.listColumns("accounts"):
   print column
```

```
scala> spark.catalog.listColumns("accounts").show
```

3. Create a new DataFrame based on the `accounts` table, and confirm that its schema matches that of the column list above.

Perform a SQL query on a table

4. Create a new DataFrame by performing a simple SQL query on the `accounts` table. Confirm that the schema and data is correct.

```
pyspark> firstLastDF = spark. \
  sql("SELECT first_name,last_name FROM accounts")
```

```
scala> val firstLastDF = spark.
  sql("SELECT first_name,last_name FROM accounts")
```

5. *Optional*: Perform the equivalent query using the DataFrame API, and compare the schema and data in the results to those of the query above.

Create and query a view

6. Create a DataFrame called `accountDeviceDF` based on the CSV files in `/loudacre/accountdevice`. (Be sure to use the headers and inferred schema to determine the column names and types.)

7. Create a temporary view on the `accountDeviceDF` DataFrame called `account_dev`.

8. Confirm the view was created correctly by listing the tables and views in the default database as you did earlier. Notice that the `account_dev` table type is `TEMPORARY`.

9. Using a SQL query, create a new DataFrame based on the first five rows of the `account_dev` table, and display the results.

```
> spark.sql("SELECT * FROM account_dev LIMIT 5").show()
```

Use SQL to join two tables

10. Join the `accounts` and `account_dev` tables using the account ID, and display the results. (Note that the SQL string in the command below must be entered on a single line in the Spark shell.)

```
pyspark> nameDevDF = spark.sql("SELECT acct_num,
 first_name, last_name, account_device_id
 FROM accounts JOIN account_dev ON acct_num =
 account_id")
pyspark> nameDevDF.show()
```

```
scala> val nameDevDF = spark.sql("SELECT acct_num,
 first_name, last_name, account_device_id
 FROM accounts JOIN account_dev ON acct_num =
 account_id")
scala> nameDevDF.show
```

11. Save `nameDevDF` as a table called `name_dev` (with the file path as `/loudacre/name_dev`).

12. Use the Catalog API to confirm that the table was created correctly with the right schema.

13. *Optional*: If you are familiar with using Hive or Impala, verify that the `name_dev` table now exists in the Hive metastore. If you use Impala, be sure to invalidate Impala's local store of the metastore using the `INVALIDATE METADATA` command or the refresh icon in the Hue Impala Query Editor.

14. *Optional*: Exit and restart the shell and confirm that the temporary view is no longer available.

This is the end of the exercise.

Hands-On Exercise: Using Datasets in Scala

Files and Data Used in This Exercise:

Exercise directory	`$DEVSH/exercises/datasets`
Data files (HDFS)	`/loudacre/weblogs`

In this exercise, you will explore Datasets using web log data.

Create an RDD of account ID/IP address pairs, and then create a new Dataset of products (case class objects) based on that RDD. Compare the results of typed and untyped transformations to better understand the relationship between DataFrames and Datasets.

Note: These exercises are in Scala only, because Datasets are not defined in Python.

Important: This exercise depends on a previous exercise: "Transforming Data Using RDDs." If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Explore Datasets using web log data

Find all the account IDs and the IP addresses from which those accounts logged in to the web site from Loudacre's web log data.

1. Create a case class for account ID/IP address pairs.

```
scala> case class AccountIP (id: Int, ip: String)
```

2. Create an RDD of `AccountIP` objects by using the web log data in `/loudacre/weblogs`. Split the data by spaces and use the first field as IP address and the third as account ID.

```
scala> val accountIPRDD = sc.
  textFile("/loudacre/weblogs").
  map(line => line.split(' ')).
  map(fields =>
        new AccountIP(fields(2).toInt,fields(0)))
```

3. Create a Dataset of `AccountIP` objects using the new RDD.

```
scala> val accountIPDS = spark.createDataset(accountIPRDD)
```

4. View the schema and data in the new Dataset.

5. Compare the result types of a typed transformation—`distinct`—and an untyped transformation—`groupBy/count`.

```
scala> val distinctIPDS = accountIPDS.distinct
scala> val accountIPCountDS = distinctIPDS.
  groupBy("id","ip").count
```

6. Save the `accountIPDS` Dataset as a Parquet file, then read the file back into a DataFrame. Note that the type of the original Dataset (`AccountIP`) is not preserved, but the types of the columns are.

Bonus Exercises

1. Try creating a new Dataset of `AccountIP` objects based on the DataFrame you created above.

2. Create a view on the `AccountIPDS` Dataset, and perform a SQL query on the view. What is the return type of the SQL query? Were column types preserved?

This is the end of the exercise.

Hands-On Exercise: Writing, Configuring, and Running a Spark Application

Files and Data Used in This Exercise:	
Exercise directory	`$DEVSH/exercises/spark-application`
Hive Tables	`accounts`
Scala project	`$DEVSH/exercises/spark-application/accounts-by-state_project`
Scala classes	`stubs.AccountsByState` `solution.AccountsByState`
Python stub	`accounts-by-state.py`

In this exercise, you will write your own Spark application instead of using the interactive Spark shell application.

Write a simple Spark application that takes a single argument, a state code (such as `CA`). The program should read the data from the `accounts` Hive table and save the rows whose `state` column value matches the specified state code. Write the results to `/loudacre/accounts_by_state/`*`state-code`* (such as `accounts_by_state/CA`).

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

Important: This exercise depends on a previous exercise: "Accessing HDFS with the Command Line and Hue." If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Editing Scala and Python Files

You may use any text editor you wish to work on your application code. If you do not have an editor preference, you may wish to use gedit, which includes language-specific support for Scala.
You can start gedit on your remote desktop using the desktop shortcut.

Write and Run a Spark Application in Python

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

2. A simple stub file to get you started has been provided in the exercise directory: `$DEVSH/exercises/spark-application/python-stubs/accounts-by-state.py`. This stub imports the required Spark classes and sets up your main code block. Open the stub file in an editor.

3. Create a `SparkSession` object using the following code:

```
spark = SparkSession.builder.getOrCreate()
```

4. In the body of the program, load the `accounts` Hive table into a DataFrame. Select accounts where the `state` column value matches the string provided as the first argument to the application. Save the results to a directory called `/loudacre/accounts_by_state/`*`state-code`* (where *`state-code`* is a string such as `CA`.) Use `overwrite` mode when saving the file so that you can re-run the application without needing to delete the directory.

5. At the end of the application, be sure to stop the Spark session:

```
spark.stop()
```

6. If you have a Spark shell running in any terminal, exit the shell before running your application.

7. Run your application. In a terminal window, change to the exercise working directory, then run the program, passing the state code to select. For example, to select accounts in California, use the following command:

```
$ cd $DEVSH/exercises/spark-application/
$ spark2-submit python-stubs/accounts-by-state.py CA
```

Note: To run the solution application, use **`python-solution`**`/accounts-by-state.py`.

8. After the program completes, use `parquet-tools` to verify that the file contents are correct. For example, if you used the state code `CA`, you would use the command below:

```
$ parquet-tools head \
  hdfs://master-1/loudacre/accounts_by_state/CA
```

For more information about `parquet-tools`, run `parquet-tools --help`.

9. Skip the section below on writing and running a Spark application in Scala and continue with View the Spark Application UI.

Write and Run a Spark Application in Scala

A Maven project to get you started has been provided in the exercise directory: `$DEVSH/exercises/spark-application/accounts-by-state_project.`

The first time you build a Scala Spark application in your environment, Maven must first download the Spark libraries from the Maven central repository. This can take several minutes, so before starting work on the exercise steps, compile the exercise project following the steps below. (Maven only needs to download the libraries the first time. It caches the libraries locally, and uses the cache for subsequent builds.)

10. In a terminal window, change to the project directory. Be sure to enter the command line shown below as a single line.

```
$ cd \
$DEVSH/exercises/spark-application/accounts-by-state_project
```

11. Build the exercise project using Maven.

```
$ mvn package
```

Maven will begin to download the required Spark libraries. Next time you build the project, Maven will used the libraries in its local cache. While Maven downloads the libraries, continue with the exercise steps below.

12. Edit the Scala class defined in `stubs` package in `src/main/scala/stubs/AccountsByState.scala`.

13. Create a `SparkSession` object using the following code:

```
val spark = SparkSession.builder.getOrCreate()
```

14. In the body of the program, load the `accounts` Hive table into a DataFrame. Select accounts where the `state` column value matches the string provided as the first argument to the application. Save the results to a Parquet file called `/loudacre/accounts_by_state/`*`state-code`* (where *`state-code`* is a string such as `CA`). Use `overwrite` mode when saving the file so that you can re-run the application without needing to delete the save directory.

15. At the end of the application, be sure to stop the Spark session:

```
spark.stop
```

16. Return to the terminal in which you ran Maven earlier in order to cache Spark libraries locally. If the Maven command is still running, wait for it to finish. When it finishes, rebuild the project, this time including the code you added above. This time, the Maven command should take much less time.

```
$ mvn package
```

    If the build is successful, Maven will generate a JAR file called `accounts-by-state-1.0.jar` in the `target` directory.

17. If you have a Spark shell running in any terminal, exit the shell before running your application.

18. Run the program in the compiled JAR file, passing the state code to select. For example, to select accounts in California, use the following command:

```
$ spark2-submit \
  --class stubs.AccountsByState \
  target/accounts-by-state-1.0.jar CA
```

    Note: To run the solution application, use `--class solution.AccountsByState`.

19. After the program completes, use `parquet-tools` to verify that the file contents are correct. For example, if you used the state code CA, you would use the command below:

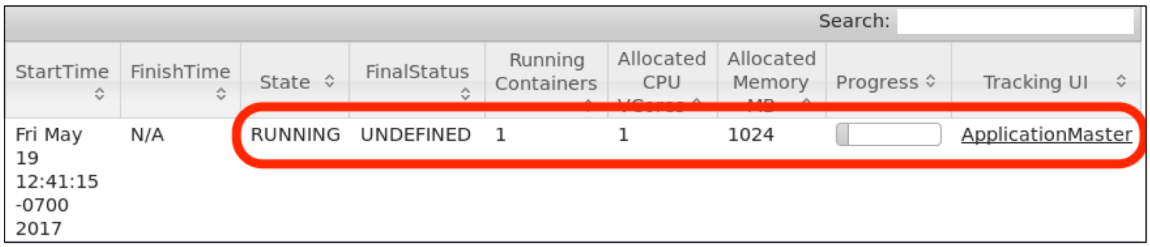
```
$ parquet-tools head \
  hdfs://master-1/loudacre/accounts_by_state/CA/
```

View the Spark Application UI

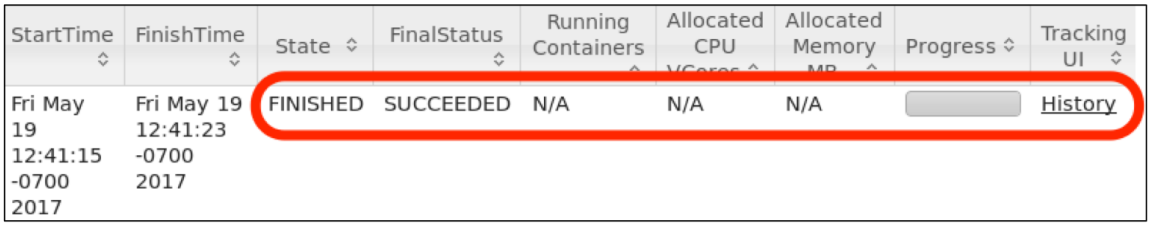
In the previous section, you ran a Python or Scala Spark application using `spark2-submit`. Now view that application's Spark UI (or history server UI if the application is complete).

20. Open Firefox on your remote desktop and visit the YARN Resource Manager UI using the provided **RM** bookmark (or go to URI `http://master-1:8088/`).

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:



To view the Spark Application UI if your application is still running, select the **ApplicationMaster** link. To view the History Service UI if your application has completed, select the **History** link.

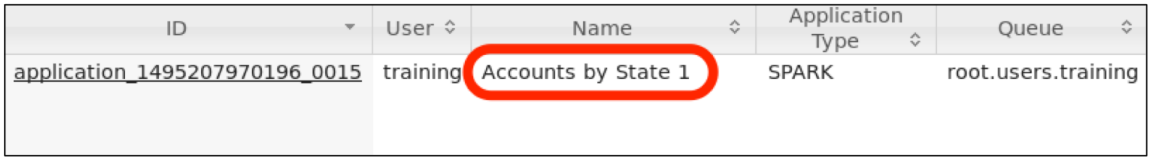
Set Configuration Options Using Submit Script Flags

21. Change to the correct directory (if necessary) and re-run the Python or Scala program you wrote in the previous section, this time specifying an application name. For example:

```
$ spark2-submit --name "Accounts by State 1" \
  python-stubs/accounts-by-state.py CA
```

```
$ spark2-submit --name "Accounts by State 1" \
  --class stubs.AccountsByState \
  target/accounts-by-state-1.0.jar CA
```

22. Go back to the YARN RM UI in your browser, and confirm that the application name was set correctly in the list of applications.



23. Follow the **ApplicationMaster** or **History** link. View the **Environment** tab. Take note of the `spark.*` properties such as `master` and `app.name`.

24. You can set most of the common application properties using submit script flags such as `name`, but for others you need to use `conf`. Use `conf` to set the `spark.default.parallelism` property, which controls how many partitions result after a "wide" RDD operation like `reduceByKey`.

```
$ spark2-submit --name "Accounts by State 2" \
  --conf spark.default.parallelism=4 \
   python-stubs/accounts-by-state.py CA
```

```
$ spark2-submit --name "Accounts by State 2" \
  --conf spark.default.parallelism=4 \
  --class stubs.AccountsByState \
  target/accounts-by-state-1.0.jar CA
```

25. View the application history for this application to confirm that the `spark.default.parallelism` property was set correctly. (You will need to view the YARN RM UI again to view the correct application's history.)

Optional Review Property Setting Overrides

26. Rerun the previous submit command with the `verbose` option. This will display your application property default and override values.

```
$ spark2-submit --verbose --name "Accounts by State 3" \
  --conf spark.default.parallelism=4 \
  python-stubs/accounts-by-state.py CA
```

```
$ spark2-submit --verbose --name "Accounts by State 3" \
  --conf spark.default.parallelism=4 \
  --class stubs.AccountsByState \
  target/accounts-by-state-1.0.jar CA
```

27. Examine the extra output displayed when the application starts up.

a. The first section starts with `Using properties file`, and shows the file name and the default property settings the application loaded from that properties file.

- What is the system properties file?
- What properties are being set in the file?

b. The second section starts with `Parsed arguments`. This lists the arguments—that is, the flags and settings—you set when running the submit script (except for `conf`). Submit script flags that you didn't pass use their default values, if defined by the script, or are shown as `null`.

- Does the list correctly include the value you set with `--name`?
- Which arguments (flags) have defaults set in the script and which do not?

c. Scroll down to the section that starts with `System properties`. This list shows *all* the properties set—those loaded from the system properties file, those you set using submit script arguments, and those you set using the `conf` flag.

- Is `spark.default.parallelism` included and set correctly?

Bonus Exercise: Set Configuration Properties Programmatically

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

1. Edit the Python or Scala application you wrote above, and use the builder function `appName` to set the application name.
2. Re-run the application without using script options to set properties.
3. View the YARN UI to confirm that the application name was correctly set.

You can find the Python bonus solution in `$DEVSH/exercises/spark-application/python-bonus`. The Scala solution is in the `bonus` package in the exercise project.

This is the end of the exercise.

Hands-On Exercise: Exploring Query Execution

Files and Data Used in This Exercise:

Exercise directory	`$DEVSH/exercises/query-execution`

Data files (HDFS)	`/user/hive/warehouse/accounts`
	`/loudacre/weblogs`
	`/loudacre/devices.json`
	`/loudacre/accountdevice`

Hive tables	`accounts`

In this exercise, you will explore how Spark executes RDD and DataFrame/ Dataset queries.

First, you will explore RDD partitioning and lineage-based execution plans using the Spark shell and the Spark Application UI. Then you will explore how Catalyst executes DataFrame and Dataset queries.

Important: This exercise depends on a previous exercise: "Transforming Data Using RDDs". If you did not complete those exercises, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Explore Partitioning of File-Based RDDs

1. Review the accounts data files (`/user/hive/warehouse/accounts/`) using Hue or the command line. Take note of the number of files.

2. In the Spark shell, create an RDD called `accountsRDD` by reading the accounts data, splitting it by commas, and keying it by account ID, which is the first field of each line.

```
pyspark> accountsRDD = sc. \
  textFile("/user/hive/warehouse/accounts"). \
  map(lambda line: line.split(',')). \
  map(lambda account: (account[0],account))
```

```
scala> val accountsRDD = sc.
  textFile("/user/hive/warehouse/accounts").
  map(line => line.split(',')).
  map(account => (account(0),account))
```

3. Find the number of partitions in the new RDD.

```
pyspark> accountsRDD.getNumPartitions()
```

```
scala> accountsRDD.getNumPartitions
```

4. Use `toDebugString` to view the lineage and execution plan of `accountsRDD`. How many partitions are in the resulting RDD? How may stages does the query have?

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

```
scala> accountsRDD.toDebugString
```

Explore Execution of RDD Queries

5. Call `count` on `accountsRDD` to count the number of accounts. This will trigger execution of a job.

6. In the browser, view the application in the YARN RM UI using the provided bookmark (or `http://master-1:8088`) and click through to view the Spark Application UI.

7. Make sure the **Jobs** tab is selected, and review the list of completed jobs. The most recent job, which you triggered by calling `count`, should be at the top of the list. (Note that the job description is usually based on the action that triggered the job execution.) Confirm that the number of stages is correct, and the number of tasks completed for the job matches the number of RDD partitions you noted when you used `toDebugString`.

8. Click on the job description to view details of the job. This will list all the stages in the job, which in this case is one.

9. Click on **DAG Visualization** to see a diagram of the execution plan based on the RDD's lineage. The main diagram displays only the stages, but if you click on a stage, it will show you the tasks within that stage.

10. *Optional*: Explore the partitioning and DAG of a more complex query like the one below. Before you view the execution plan or job details, try to figure out how many stages the job will have.

This query loads Loudacre's web log data, and calculates how many times each user visited. Then it joins that user count data with account data for each user.

```
pyspark> logsRDD = sc.textFile("/loudacre/weblogs")
pyspark> userReqsRDD = logsRDD. \
  map(lambda line: line.split(' ')). \
  map(lambda words: (words[2],1)).  \
  reduceByKey(lambda v1,v2: v1 + v2)
pyspark> accountHitsRDD = accountsRDD.join(userReqsRDD)
```

```
scala> val logs = sc.textFile("/loudacre/weblogs")
scala> val userReqsRDD = logs.map(line => line.split(' ')).
  map(words => (words(2),1)).
  reduceByKey((v1,v2) => v1 + v2)
scala> val accountHitsRDD = accountsRDD.join(userReqsRDD)
```

Note: If you execute the query multiple times, you may note that some tasks within a stage are marked as "skipped." This is because whenever a shuffle operation is executed, Spark temporarily caches the data that was shuffled. Subsequent executions of the same query re-use that data if it's available to save some steps and increase performance.

Explore Execution of DataFrame Queries

11. Create a DataFrame of active accounts from the `accounts` table.

```
pyspark> accountsDF = spark.read.table("accounts")
pyspark> activeAccountsDF = accountsDF. \
  select("acct_num"). \
  where(accountsDF.acct_close_dt.isNull())
```

```
scala> val accountsDF = spark.read.table("accounts")
scala> val activeAccountsDF = accountsDF.
  select("acct_num").
  where($"acct_close_dt".isNull)
```

12. View the full execution plan for the new DataFrame.

```
pyspark> activeAccountsDF.explain(True)
```

```
scala> activeAccountsDF.explain(true)
```

    Can you locate the line in the physical plan corresponding to the command to load the `accounts` table into a DataFrame?

    How many stages do you think this query has?

13. Call the DataFrame's `show` function to execute the query.

14. View the Spark Application UI and choose the **SQL** tab. This displays a list of DataFrame and Dataset queries you have executed, with the most recent query at the top.

15. Click the description for the top query to see the visualization of the query's execution. You can also see the query's full execution plan by opening the **Details** panel below the visualization graph.

16. The first step in the execution is a `HiveTableScan`, which loaded the account data into the DataFrame. Hover your mouse over the step to show the step's execution plan. Compare that to the physical plan for the query. Note that it is the same as the last line in the physical execution plan, because it is the first step to execute. Did you correctly identify this line in the execution plan as the one corresponding to the `DataFrame.read.table` operation?

17. The **Succeeded Jobs** label provides links to the jobs that executed as part of this query execution. In this case, there is just a single job. Click the job's ID (in the **Jobs** column) to view the job details. This will display a list of stages that were completed for the query.

    How many stages executed? Is that the number of stages you predicted it would be?

18. *Optional*: Click the description of the stage to view metrics on the execution of the stage and its tasks.

19. The previous query was very simple, involving just a single data source with a `where` to return only active accounts. Try executing a more complex query that joins data from two different data sources.

    This query reads in the `accountdevice` data file, which maps that maps account IDs to associated device IDs. Then it joins that data with the DataFrame of active

accounts you created above. The result is DataFrame consisting of all device IDs in use by currently active accounts.

```
pyspark> accountDeviceDF = spark.read. \
  option("header","true"). \
  option("inferSchema","true"). \
  csv("/loudacre/accountdevice")
pyspark> activeAcctDevsDF = activeAccountsDF. \
  join(accountDeviceDF,
    accountsDF.acct_num == accountDeviceDF.account_id). \
  select("device_id")
```

```
scala> val accountDeviceDF = spark.read.
  option("header","true").
  option("inferSchema","true").
  csv("/loudacre/accountdevice")
scala> val activeAcctDevsDF = activeAccountsDF.
  join(accountDeviceDF,$"acct_num" === $"account_id").
  select($"device_id")
```

20. Review the full execution plan using `explain`, as you did with the previous DataFrame.

Can you identify which lines in the execution plan load the two different data sources?

How many stages do you think this query will execute?

21. Execute the query and review the execution visualization in the Spark UI.

What differences do you see between the execution of the earlier query and this one?

How many stages executed? Is this what you expected?

22. *Optional*: Explore an even more complex query that involves multiple joins with three data sources. You can use the last query in the solutions file for this exercise (in the `$DEVSH/exercises/query-execution/solution/` directory). That query creates a list of device IDs, makes, and models, and the number of active accounts that use that type of device, sorted in order from most popular device type to least.

This is the end of the exercise.

Hands-On Exercise: Persisting Data

Files and Data Used in This Exercise:

Exercise directory	`$DEVSH/exercises/persist`
Data files (HDFS)	`/loudacre/accountdevice`
Hive tables	`accounts`

In this exercise, you will explore DataFrame persistence.

Important: This exercise depends a on previous exercise: "Analyzing Data with DataFrame Queries." If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Compare the Execution Plans of Persisted and Unpersisted Queries

1. Create a DataFrame that joins account data for active accounts with their associated devices. To save time and effort, copy the query code from the `persist.pyspark` or `.scalaspark` file from the `$DEVSH/exercises/persist/stubs` directory into the Spark shell.

2. The query code you pasted above defines a new DataFrame called `accountsDevsDF`, which joins account data and device data for all active accounts. Try executing a query starting with the `accountsDevsDF` DataFrame that displays the account number, first name, last name and device ID for each row.

```
pyspark> accountsDevsDF. \
  select("acct_num","first_name","last_name","device_id"). \
  show(5)
```

```
scala> accountsDevsDF.
  select("acct_num","first_name","last_name","device_id").
  show(5)
```

3. In your browser, go to the **SQL** tab of your application's Spark UI, and view the execution visualization of the query you just executed. Take note of the complexity so that you can compare it to later executions when using persistence.

   Remember that queries are listed in the **SQL** tab in the order they were executed, starting with the most recent. The descriptions of multiple executions of the same

action will not distinguish one query from another, so make sure you choose the correct one for the query you are looking at.

4. In your Spark shell, persist the `accountsDevsDF` DataFrame using the default storage level.

```
pyspark> accountsDevsDF.persist()
```

```
scala> accountsDevsDF.persist
```

5. Repeat the final steps of the query you executed above.

```
pyspark> accountsDevsDF. \
  select("acct_num","first_name","last_name","device_id"). \
  show(5)
```

```
scala> accountsDevsDF.
  select("acct_num","first_name","last_name","device_id").
  show(5)
```

6. In the browser, reload the Spark UI **SQL** tab, and view the execution diagram for the query just just executed. Notice that it has far fewer steps. Instead of reading, filtering, and joining the data from the two sources, it reads the persisted data from memory. If you hover your mouse over the memory scan step, you will see that the only operation it performs on the data in memory is the last step of the query: the unpersisted `select` transformation. Compare the diagram for this query with the first one you executed above, before persisting.

7. The first time you execute a query on a persisted DataFrame, Dataset, or RDD, Spark has to execute the full query in order to materialize the data that gets saved in memory or on disk. Compare the difference between the first and second queries after executing `persist` by re-executing the query one final time. Then use the Spark UI to compare both queries executed after the `persist` operation, and consider these questions.

   - Do the execution diagrams differ? Why or why not?

   - Did one query take longer than the other? If so, which one, and why?

View Storage for Persisted DataFrames

8. View the **Storage** tab in the Spark UI to see currently persisted data. The list shows the RDD identified by the execution plan for the query that generated the data. Consider these questions.

   - What is the storage level of the RDD?
   - How many partitions of the RDD were persisted and how much space do those partitions take up in memory and on disk?
   - Note that only a small percentage of the data is cached. Why is that? How could you cache more of the data?
   - Click the RDD name to view the storage details. Which executors are storing data for this RDD?

9. Execute the same query as above using the `write` action instead of `show`.

```
pyspark> accountsDevsDF.write.mode("overwrite"). \
  save("/loudacre/accounts_devices")
```

```
scala> accountsDevsDF.write.mode("overwrite").
  save("/loudacre/accounts_devices")
```

10. Reload the Spark UI **Storage** tab.

    - What percentage of the data is cached? Why? How does this compare to the last time you persisted the data?
    - How much memory is the data taking up? How much disk space?

Change the Storage Level for the Persisted DataFrame

11. Unpersist the `accountsDevsDF` DataFrame.

```
> accountsDevsDF.unpersist()
```

12. View the Spark UI **Storage** to verify that the cache for `accountsDevsDF` has been removed.

13. Repersist the same DataFrame, setting the storage level to save the data to files on disk, replicated twice.

```
pyspark> from pyspark import StorageLevel
pyspark> accountsDevsDF.persist(StorageLevel.DISK_ONLY_2)
```

```
scala> import org.apache.spark.storage.StorageLevel
scala> accountsDevsDF.persist(StorageLevel.DISK_ONLY_2)
```

14. Reexecute the previous query.

15. Reload the **Storage** tab to confirm that the storage level for the RDD is set correctly. Also consider these questions:

- How much memory is the data taking up? How much disk space?
- Which executors are storing the RDD's data files?
- How many partitions are stored? Are they replicated? Where?

This is the end of the exercise.

Hands-On Exercise: Implement an Iterative Algorithm with Apache Spark

Files and Data Used in This Exercise:

Exercise directory	`$DEVSH/exercises/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

1. If you completed the bonus section of the "Process Data Files with Apache Spark" 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 complete that bonus exercise, upload the solution file to HDFS now. (If you have run the course catch-up script, this has already done for you.)

```
$ hdfs dfs -put $DEVDATA/static_data/devicestatus_etl \
  /loudacre/
```

2. 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

3. Start by copying the code provided in the `KMeansCoords` stub file into your Spark shell. The starter code defines 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 equal to `5`—this is the number of means to calculate.

4. 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`.

5. Parse the input file—which is delimited by commas—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 the resulting RDD because you will access it each time through the iteration.

6. 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:

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

   Or in Scala:

```
val kPoints = data.takeSample(false, K, 42)
```

7. 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 that 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 also find 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 the centers "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.

8. When all iterations are complete, display the final K center points.

This is the end of the exercise.

Hands-On Exercise: Writing a Streaming Application

Files and Directories Used in This Exercise:	
Exercise directory	`$DEVSH/exercises/streaming-dstreams`
Python stub	`stubs-python/StreamingLogs.py`
Python solution	`solution-python/StreamingLogs.py`
Scala project directory	`streaminglogs_project`
Scala stub class	`stubs.StreamingLogs`
Scala solution class	`solution.StreamingLogs`
Test data (local)	`$DEVDATA/weblogs/`
Test script	`streamtest.py`

In this exercise, you will write a Spark Streaming application to count Knowledge Base article requests.

This exercise has two parts. First, you will review the Spark Streaming documentation. Then you will write and test a Spark Streaming application to read streaming web server log data and count the number of requests for Knowledge Base articles.

Review the Spark Streaming Documentation

1. View the Spark Streaming API in the Spark API documentation for either Scala or Python and then:

   For Scala:

   - Scroll down and select the `org.apache.spark.streaming` package in the package pane on the left.
   - Follow the links at the top of the package page to view the `DStream` and `PairDStreamFunctions` classes— these will show you the methods available on a DStream of regular RDDs and pair RDDs respectively.

   For Python:

   - Go to the `pyspark.streaming` module.
   - Scroll down to the `pyspark.streaming.DStream` class and review the available methods.

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

Simulate Streaming Web Logs

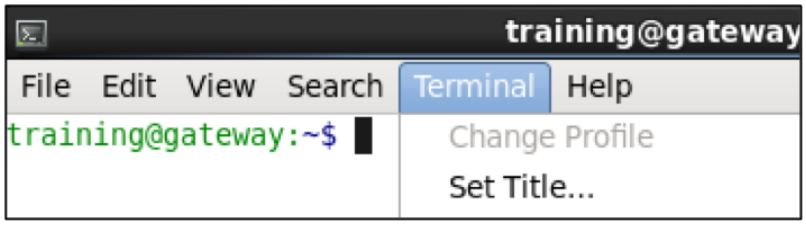
To simulate a streaming data source, you will use the provided `streamtest.py` Python script, which waits for a connection on the host and port specified and, once it receives a connection, sends the contents of the file(s) specified to the client (which will be your Spark Streaming application). You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

3. Stream the Loudacre web log files at a rate of 20 lines per second using the provided test script.

```
$ python $DEVSH/scripts/streamtest.py gateway 1234 20 \
  $DEVDATA/weblogs/*
```

This script will exit after the client disconnects, so you will need to restart the script when you restart your Spark application.

Tip: This exercise involves using multiple terminal windows on your remote desktop. To avoid confusion, set a different title for each one by selecting **Set Title...** on the **Terminal** menu:



Set the title for this window to "Test Streaming."

Write a Spark Streaming Application

4. To help you get started writing a Spark Streaming application, stub files have been provided for you.

For Python, start with the stub file `StreamingLogs.py` in the `$DEVSH/exercises/streaming-dstreams/stubs-python` directory, which imports the necessary classes for the application.

For Scala, a Maven project directory called `streaminglogs_project` has been provided in the exercise directory (`$DEVSH/exercises/streaming-`

`dstreams`). To complete the exercise, start with the stub code in `src/main/scala/stubs/StreamingLogs.scala`, which imports the necessary classes for the application.

5. Define a Streaming context with a one-second batch duration.
6. Create a DStream by reading the data from the host and port provided as input parameters.
7. Filter the DStream to only include lines containing the string `KBDOC`.
8. To confirm that your application is correctly receiving the streaming web log data, display the first five records in the filtered DStream for each one-second batch. (In Scala, use the DStream `print` function; in Python, use `pprint`.)
9. For each RDD in the filtered DStream, display the number of items—that is, the number of requests for KB articles.

   Tip: Python does not allow calling `print` within a lambda function, so create a named defined function to print.
10. Save the filtered logs to text files in HDFS. Use the base directory name `/loudacre/streamlog/kblogs`.
11. Finally, start the Streaming context, and then call `awaitTermination()`.

Test the Application

After a few batches of data have been processed and displayed, you can end the test.

12. Open a new terminal window, and change to the correct directory for the language you are using for your application.

    For Python, change to the exercise directory:

```
$ cd $DEVSH/exercises/streaming-dstreams
```

    For Scala, change to the project directory for the exercise:

```
$ cd \
  $DEVSH/exercises/streaming-dstreams/streaminglogs_project
```

13. If you are using Scala, build your application JAR file using Maven.

```
$ mvn package
```

Note: If this is your first time compiling a Spark Scala application, it may take several minutes for Maven to download the required libraries to package the application.

14. Use `spark2-submit` to run your application. The `StreamingLogs` application takes two parameters: the host name and the port number to which to connect the DStream. Specify the same host and port at which the test script you started earlier is listening.

Spark Streaming applications running on a cluster must have two worker nodes available. However, only a single node is available in the exercise environment. To work around this, you will need to run the application locally, rather than on the cluster. You must specify at least two threads, as shown below.

For Python:

```
$ spark2-submit --master local[2] \
  stubs-python/StreamingLogs.py gateway 1234
```

Note: Use `solution-python/StreamingLogs.py` to run the solution application instead.

For Scala:

```
$ spark2-submit --master local[2] \
  --class stubs.StreamingLogs \
  target/streamlog-1.0.jar gateway 1234
```

Note: Use `--class solution.StreamingLogs` to run the solution class instead.

15. After a few moments, the application will connect to the test script's simulated stream of web server log output. Confirm that for every batch of data received (every second), the application displays the first few Knowledge Base requests and the count of requests in the batch. Review the HDFS files the application saved in `/loudacre/streamlog`.

16. Return to the terminal window in which you started the `streamtest.py` test script earlier. Stop the test script by typing `Ctrl+C`.

17. Return to the terminal window in which your application is running. Stop your application by typing `Ctrl+C`. (You may see several error messages resulting from the interruption of the job in Spark; you may disregard these.)

This is the end of the exercise.

Hands-On Exercise: Processing Multiple Batches of Streaming Data

Files and Data Used in This Exercise	
Exercise directory	`$DEVSH/exercises/streaming-multi`
Python stub	`stubs-python/StreamingLogsMB.py`
Python solution	`solution-python/StreamingLogsMB.py`
Scala project directory	`streaminglogsMB_project`
Scala stub class	`stubs.StreamingLogsMB`
Scala solution class	`solution.StreamingLogsMB`
Data (local)	`$DEVDATA/weblogs`

In this exercise, you will write a Spark Streaming application to count web page requests over time.

Simulate Streaming Web Logs

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

1. Open a new terminal window. This exercise uses multiple terminal windows. To avoid confusion, set a different title for the new window, such as "Test Stream."

2. Stream the Loudacre Web log files at a rate of 20 lines per second using the provided test script.

```
$ python $DEVSH/scripts/streamtest.py gateway 1234 20 \
  $DEVDATA/weblogs/*
```

   This script exits after the client disconnects, so you will need to restart the script when you restart your Spark application.

Display the Total Request Count

3. A stub file for this exercise has been provided for you in the exercise directory. The stub code creates a Streaming context for you, and creates a DStream called `logs` based on web log request messages received on a network socket.

   For Python, start with the stub file `StreamingLogsMB.py` in the `stubs-python` directory.

   For Scala, a Maven project directory called `streaminglogsMB_project` has been provided in the exercise directory. To complete the exercise, start with the stub code in `src/main/scala/stubs/StreamingLogsMB.scala`.

4. Enable checkpointing to a directory called `logcheckpt`.

5. Count the number of page requests over a window of five seconds. Print out the updated five-second total every two seconds.

   - Hint: Use the `countByWindow` function.

Build and Run Your Application

After a few batches of data have been processed and displayed, you can end the test.

6. In a different terminal window than the one in which you started the `streamtest.py` script, change to the correct directory for the language you are using for your application. To avoid confusion, you might wish to set a different title for the new window such as "Application".

   For Python, change to the exercise directory:

```
$ cd $DEVSH/exercises/streaming-multi
```

   For Scala, change to the project directory for the exercise:

```
$ cd \
  $DEVSH/exercises/streaming-multi/streaminglogsMB_project
```

7. If you are using Scala, build your application JAR file using the `mvn package` command.

8. Use `spark2-submit` to run your application. Your application takes two parameters: the host name and the port number to connect the DStream to. Specify the same host and port at which the test script you started earlier is listening.

For Python:

```
$ spark2-submit --master local[2] \
  stubs-python/StreamingLogsMB.py gateway 1234
```

Note: Use `solution-python/StreamingLogsMB.py` to run the solution application instead.

For Scala:

```
$ spark2-submit --master local[2] \
  --class stubs.StreamingLogsMB \
  target/streamlogmb-1.0.jar gateway 1234
```

Note: Use `--class solution.StreamingLogsMB` to run the solution class instead.

9. After a few moments, the application should connect to the test script's simulated stream of web server log output. Confirm that for every batch of data received (every second), the application displays the count of requests in the batch. Review the files.

10. Return to the terminal window in which you started the `streamtest.py` test script earlier. Stop the test script by typing `Ctrl+C`.

11. Return to the terminal window in which your application is running. Stop your application by typing `Ctrl+C`. (You may see several error messages resulting from the interruption of the job in Spark; you may disregard these.)

Bonus Exercise

Extend the application you wrote above to also count the total number of page requests by user from the start of the application, and then display the top ten users with the highest number of requests.

Follow the steps below to implement a solution for this bonus exercise:

1. Use map-reduce to count the number of times each user made a page request in each batch (a hit-count).

   - Hint: Remember that the User ID is the 3rd field in each line.

2. Define a function called `updateCount` that takes an array (in Python) or sequence (in Scala) of hit-counts and an existing hit-count for a user. The function should return the sum of the new hit-counts plus the existing count.

   - Hint: An example of an `updateCount` function is in the course material and the code can be found in `$DEVSH/examples/spark/spark-streaming`.

3. Use `updateStateByKey` with your `updateCount` function to create a new DStream of users and their hit-counts over time.

4. Use `transform` to call the `sortByKey` transformation to sort by hit-count.

   - Hint: You will have to swap the key (user ID) with the value (hit-count) to sort.

Note: The solution files for this bonus exercise are in the `bonus` package in the exercise Maven project directory (Scala) and in `solution-python/bonus` in the exercise directory (Python).

This is the end of the exercise.

Hands-On Exercise: Processing Streaming Apache Kafka Messages

Files and Data Used in This Exercise	
Exercise directory	`$DEVSH/exercises/streaming-kafka`
Python stub	`stubs-python/StreamingLogsKafka.py`
Python solution	`stubs-solution/StreamingLogsKafka.py`
Scala project directory	`streaminglogskafka_project`
Scala stub class	`stubs.StreamingLogsKafka`
Scala solution class	`solution.StreamingLogsKafka`
Data (local)	`$DEVDATA/weblogs/*`

In this exercise, you will write an Apache Spark Streaming application to handle web logs received as messages on a Kafka topic.

Write a Spark Streaming Application Using a Direct Kafka DStream

Write an application to consume Kafka messages in the `weblogs` topic using direct Kafka DStream.

In this exercise you will work in the exercise directory: `$DEVSH/exercises/streaming-kafka`. Within that directory are separate language-specific directories containing stub files that import the necessary libraries for your application.

- For Python, start with the stub file `StreamingLogsKafka.py` in the `stubs-python` directory.
- For Scala, a Maven project directory called `streaminglogskafka_project` has been provided. To complete the exercise, start with the stub code in `src/main/scala/stubs/StreamingLogsKafka.scala`.

1. Your application should accept two input arguments that the user will set when starting the application:
   - First argument: the Kafka topic from which to retrieve messages
   - Second argument: a comma-separated list of Kafka brokers and ports, such as `broker1:port,broker2:port,broker3:port`. (This could be any number of brokers, depending on the environment.)

2. Create a DStream using `KafkaUtils.createDirectStream`. The topic list and broker list should use the arguments passed by the user when submitting the application.

   Refer to the course materials for the details of creating a Kafka stream.

3. Kafka messages are in (key, value) form, but for this application, the key is null and only the value is needed. (The value is the web log line.) Map the DStream to remove the key and use only the value.

4. To verify that the DStream is correctly receiving messages, display the first 10 elements in each batch.

5. For each RDD in the DStream, display the number of items—that is, the number of requests.

   Tip: Python does not allow calling `print` within a lambda function, so define a named function to print.

6. Save the filtered logs to text files in HDFS. Use the base directory name `/loudacre/streamlog/kafkalogs`.

Build Your Scala Application

If you are using Scala, you need to build your application before running it. If you are using Python, you can skip these steps.

7. Change to the project directory for the exercise.

```
$ cd \
$DEVSH/exercises/streaming-kafka/streaminglogskafka_project
```

8. Build your application JAR file using the `mvn package` command.

Create a Kafka Topic

9. Open a new terminal window. This exercise uses multiple terminal windows. To avoid confusion, set a different title for the new window, such as “Kafka”.

10. Use the `kafka-topics` script to create a Kafka topic called `weblogs` from which your application will consume messages.

```
$ kafka-topics --create --zookeeper master-1:2181 \
  --replication-factor 1 --partitions 2 \
  --topic weblogs
```

11. Confirm your topic was created correctly by listing topics. Make sure `weblogs` is displayed.

```
$ kafka-topics --list --zookeeper master-1:2181
```

Produce Kafka Messages to Test Your Application

12. Run the test script to generate Kafka messages to the `weblogs` topic using the data files in `$DEVDATA/weblogs` at a rate of 20 messages per second.

```
$ $DEVSH/scripts/streamtest-kafka.sh \
  weblogs worker-1:9092 20 $DEVDATA/weblogs/*
```

The script will begin displaying the messages it is sending to the `weblogs` Kafka topic. (You may disregard any `SLF4J` messages.)

Run Your Application

13. Open a new terminal window. This exercise uses multiple terminal windows. To avoid confusion, set a different title for the new window, such as "Application".

14. Change to the correct directory for the language you are using for your application.

- For Python, change to the main exercise directory:

```
$ cd $DEVSH/exercises/streaming-kafka
```

- For Scala, change to the project directory for the exercise:

```
$ cd \
$DEVSH/exercises/streaming-kafka/streaminglogskafka_project
```

15. Use `spark2-submit` to run your application. Your application takes two parameters: the name of the Kafka topic from which the DStream will read messages, `weblogs`, and a comma-separated list of broker hosts and ports.

    - For Python:

    ```
    $ spark2-submit --master local[2] \
      stubs-python/StreamingLogsKafka.py weblogs \
      worker-1:9092
    ```

    Note: Use `solution-python/StreamingLogsKafka.py` to run the solution application instead.

    - For Scala:

    ```
    $ spark2-submit --master local[2] \
      --class stubs.StreamingLogsKafka \
      target/streamlogkafka-1.0.jar weblogs \
      worker-1:9092
    ```

    Note: Use `--class solution.StreamingLogsKafka` to run the solution class instead.

16. Confirm that your application is correctly displaying the Kafka messages it receives, as well as displaying the number of received messages, every second.

    Note: It may take a few moments for your application to start receiving messages. Occasionally you might find that after 30 seconds or so, it is still not receiving any messages. If that happens, press `Ctrl+C` to stop the application, then restart it.

Clean Up

17. Stop the Spark application in the first terminal window by pressing `Ctrl+C`. (You might see several error messages resulting from the interruption of the job in Spark; you may disregard these.)

18. Stop the `streamtest-kafka.sh` script in the second terminal window by pressing `Ctrl+C`.

This is the end of the exercise.

Appendix Hands-On Exercise: Producing and Consuming Apache Kafka Messages

Files and Data Used in This Exercise	
Exercise directory	`$DEVSH/exercises/kafka`

In this exercise, you will use Kafka's command line tool to create a Kafka topic. You will also use the command line producer and consumer clients to publish and read messages.

Creating a Kafka Topic

1. Open a new terminal and create a Kafka topic named `weblogs` that will contain messages representing lines in Loudacre's web server logs.

```
$ kafka-topics --create \
  --zookeeper master-1:2181 \
  --replication-factor 1 \
  --partitions 2 \
  --topic weblogs
```

   You will see the message: `Created topic "weblogs".`

   Note: If you previously worked on an exercise that used Kafka, you may get an error here indicating that this topic already exists. You may disregard the error.

2. Display all Kafka topics to confirm that the new topic you just created is listed:

```
$ kafka-topics --list \
  --zookeeper master-1:2181
```

3. Review the details of the `weblogs` topic.

```
$ kafka-topics --describe weblogs \
  --zookeeper master-1:2181
```

Producing and Consuming Messages

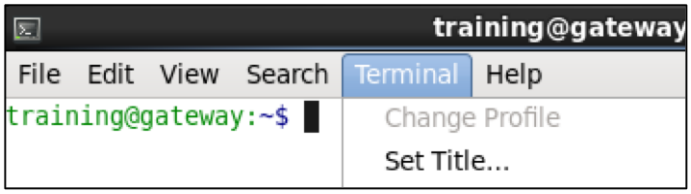
You will now use Kafka command line utilities to start producers and consumers for the topic created earlier.

4. Start a Kafka producer for the `weblogs` topic:

```
$ kafka-console-producer \
  --broker-list worker-1:9092 \
  --topic weblogs
```

You will see a few `SLF4J` messages, at which point the producer is ready to accept messages on the command line.

Tip: This exercise involves using multiple terminal windows. To avoid confusion, set a different title for each one by selecting **Set Title…** on the **Terminal** menu:



Set the title for this window to "Kafka Producer."

5. Publish a test message to the `weblogs` topic by typing the message text and then pressing `Enter`. For example:

```
test weblog entry 1
```

6. Open a new terminal window and adjust it to fit on the window beneath the producer window. Set the title for this window to "Kafka Consumer."

7. In the new terminal window, start a Kafka consumer that will read from the beginning of the `weblogs` topic:

```
$ kafka-console-consumer \
  --zookeeper master-1:2181 \
  --topic weblogs \
  --from-beginning
```

After a few `SLF4J` messages, you should see the status message you sent using the producer displayed on the consumer's console, such as:

```
test weblog entry 1
```

8. Press `Ctrl+C` to stop the weblogs consumer, and restart it, but this time omit the `--from-beginning` option to this command. You should see that no messages are displayed.

9. Switch back to the producer window and type another test message into the terminal, followed by the `Enter` key:

```
test weblog entry 2
```

10. Return to the consumer window and verify that it now displays the alert message you published from the producer in the previous step.

Cleaning Up

11. Press `Ctrl+C` in the consumer terminal window to end its process.

12. Press `Ctrl+C` in the producer terminal window to end its process.

This is the end of the exercise.

Appendix Hands-On Exercise: Collecting Web Server Logs with Apache Flume

Files and Data Used in This Exercise	
Exercise directory	`$DEVSH/exercises/flume`
Data files (local)	`$DEVDATA/weblogs`

In this exercise, you will run a Flume agent 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.

Create an HDFS Directory for Flume-Ingested Data

1. In a terminal window create a directory in HDFS called `/loudacre/weblogs_flume` to hold the data files that Flume ingests:

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

Create a Local Directory for Web Server Log Output

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

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

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

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

Configure Flume

A Flume agent configuration file has been provided for you: `$DEVSH/exercises/flume/spooldir.conf`.

Review the configuration file. *You do not need to edit this file.* Take note in particular of the following:

- The *source* is a spooling directory source that pulls from the local `/flume/weblogs_spooldir` directory.
- The *sink* is an HDFS sink that writes files to the HDFS `/loudacre/weblogs_flume` directory.
- The *channel* is a memory channel.

Run the Flume Agent

Next, start the Flume agent and copy the files to the spooling directory.

4. Start the Flume agent using the configuration you just reviewed:

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

5. 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`

Simulate Apache Web Server Output

6. Open a separate terminal window. Run the script to place the web log files in the `/flume/weblogs_spooldir` directory:

```
$ $DEVSH/scripts/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.

7. 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.

8. Once the Flume agent has finished, enter `Ctrl+C` to terminate the process.

9. Using the command line or Hue File Browser, list the files that were added by the Flume agent in the HDFS directory `/loudacre/weblogs_flume`.

   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`.

This is the end of the exercise.

Appendix Hands-On Exercise: Sending Messages from Flume to Kafka

Files and Data Used in This Exercise	
Exercise directory	`$DEVSH/exercises/flafka`
Data files (local)	`$DEVDATA/weblogs`

In this exercise, you will run a Flume agent that ingests web logs from a local spool directory and sends each line as a message to a Kafka topic.

The Flume agent is configured to send messages to the `weblogs` topic you created earlier.

Important: This exercise depends on two prior exercises: "Collect Web Server Logs with Flume" and "Produce and Consume Kafka Messages." If you did not complete both of these exercises, run the catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Configure a Flume Agent with a Kafka Sink

A Flume agent configuration file has been provided for you : `$DEVSH/exercises/flafka/spooldir_kafka.conf`

1. Review the configuration file. *You do not need to edit this file.* Take note in particular of the following points:

   - The *source* and *channel* configurations are identical to the ones in the "Collect Web Server Logs with Flume" exercise: a spooling directory source that pulls from the local `/flume/weblogs_spooldir` directory, and a memory channel.
   - Instead of an HDFS sink, this configuration uses a Kafka sink that publishes messages to the `weblogs` topic.

Run the Flume Agent

2. Start the Flume agent using the configuration you just reviewed:

```
$ flume-ng agent --conf /etc/flume-ng/conf \
  --conf-file $DEVSH/exercises/flafka/spooldir_kafka.conf \
  --name agent1 -Dflume.root.logger=INFO,console
```

3. Wait a few moments for the Flume agent to start up. You will see a message like: `Component type: SINK, name: kafka-sink started`

   Tip: This exercise involves using multiple terminal windows. To avoid confusion, set a different title for each window. Set the title of the current window to "Flume Agent."

Test the Flume Agent Kafka Sink

4. In a new terminal window, start a Kafka consumer that will read from the `weblogs` topic:

```
$ kafka-console-consumer \
  --zookeeper master-1:2181 \
  --topic weblogs
```

   Tip: Set the title of this window to "Kafka Consumer."

5. In a separate new terminal window, run the script to place the web log files in the `/flume/weblogs_spooldir` directory:

```
$ $DEVSH/scripts/copy-move-weblogs.sh \
  /flume/weblogs_spooldir
```

   Note: If you completed an earlier Flume exercise or ran `catchup.sh`, the script will prompt you whether you want to clear out the `spooldir` directory. Be sure to enter `y` when prompted.

6. In the terminal that is running the Flume agent, watch the logging output. The output will give information about the files Flume is ingesting from the source directory.

7. In the terminal that is running the Kafka consumer, confirm that the consumer tool is displaying each message (that is, each line of the web log file Flume is ingesting).

8. Once the Flume agent has finished, enter `Ctrl+C` in both the Flume agent terminal and the Kafka consumer terminal to end their respective processes.

This is the end of the exercise.

Appendix Hands-On Exercise: Import Data from MySQL Using Apache Sqoop

Files and Data Used in This Exercise	
Exercise directory	`$DEVSH/exercises/sqoop`
MySQL database	`loudacre`
MySQL table	`basestations`
HDFS paths	`/loudacre/basestations_import` `/loudacre/basestations_import_parquet`

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

Important: This exercise depends on a previous exercise: "Accessing HDFS with Command Line and Hue." If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Import 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. If you do not already have one started, open a terminal window.

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://gateway/loudacre \
  --username training --password training
```

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

```
$ sqoop help import
```

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

```
$ sqoop import \
  --connect jdbc:mysql://gateway/loudacre \
  --username training --password training \
  --table basestations \
  --target-dir /loudacre/basestations_import \
  --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. Single quotes must be used around the `\\N` character. Using double quotes will result in an "illegal escape character" error from Sqoop.

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

View the Imported Data

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

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

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

   - **Note:** Output of Hadoop jobs is saved as one or more "partition" files.

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

Import a Table Using an Alternate File Format

9. Import the `basestations` table to a Parquet data format rather than the default file form (text file).

```
$ sqoop import \
  --connect jdbc:mysql://gateway/loudacre \
  --username training --password training \
  --table basestations \
  --target-dir /loudacre/basestations_import_parquet \
  --as-parquetfile
```

10. View the results of the import command by listing the contents of the `basestations_import_parquet` directory in HDFS, using either Hue or the `hdfs` command. Note that the Parquet files are each given unique names, such as `e8f3424e-230d-4101-abba-66b521bae8ef.parquet`.

11. You can't directly view the contents of the Parquet files because they are binary files rather than text. Use the `parquet-tools head` command to view the first few records in the set of data files imported by Sqoop.

```
$ parquet-tools head \
  hdfs://master-1/loudacre/basestations_import_parquet/
```

This is the end of the exercise.

Appendix: Enabling Jupyter Notebook for PySpark

The Jupyter Notebook is an application that allows you to run Python Spark code interactively in your browser. (For more information on Jupyter, see http://jupyter.org.) Jupyter is a third-party project and is not supported by Cloudera nor included with CDH.

However, if you are used to working in a browser-based notebook environment, you may use Jupyter Notebook in this course. Although your instructor cannot provide any addition assisstance with Jupyter, the notebook is installed In the course exercise environment. To use it instead of the command-line version of PySpark, follow these steps:

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

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

```
# export PYSPARK_DRIVER_PYTHON_OPTS='notebook --ip gateway
 --no-browser'
```

3. Save the file.

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

5. Confirm the changes with the following Linux command:

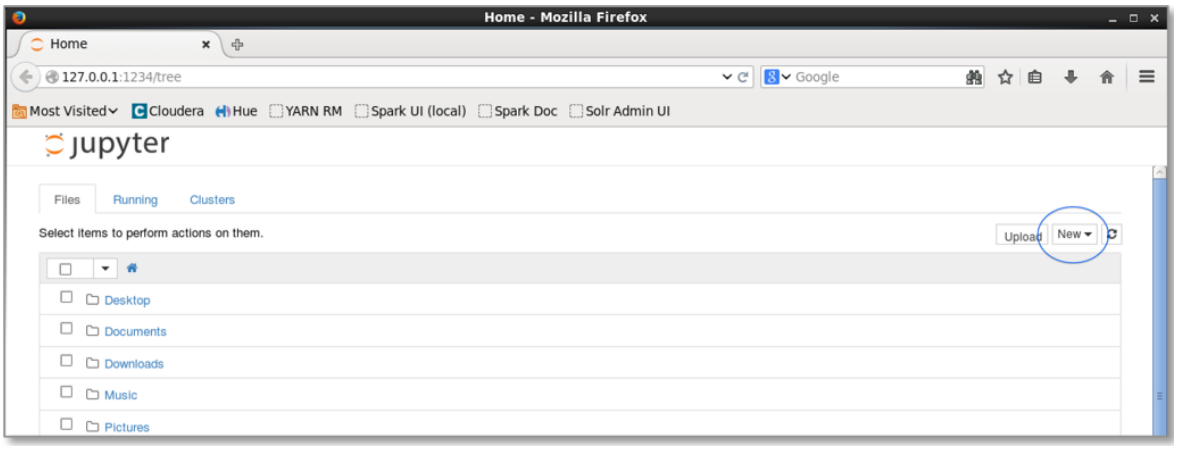
```
$ env | grep PYSPARK
```

   The output should include the setting below. If not, the `.bashrc` file was not edited or saved properly.

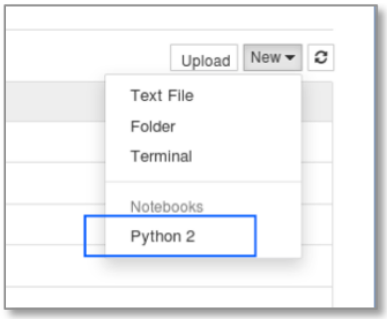
```
PYSPARK_DRIVER_PYTHON_OPTS='notebook --ip gateway --no-
browser'
```

6. Enter `pyspark2` in the terminal. This will start a notebook server.

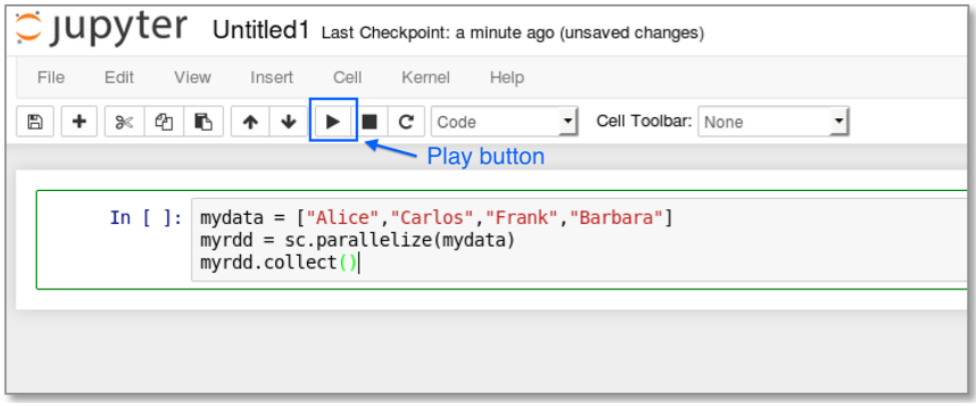
7. In a browser on your remote desktop, go to `http://gateway:8888/`. This will cause a browser window to open, and you should see the following web page:



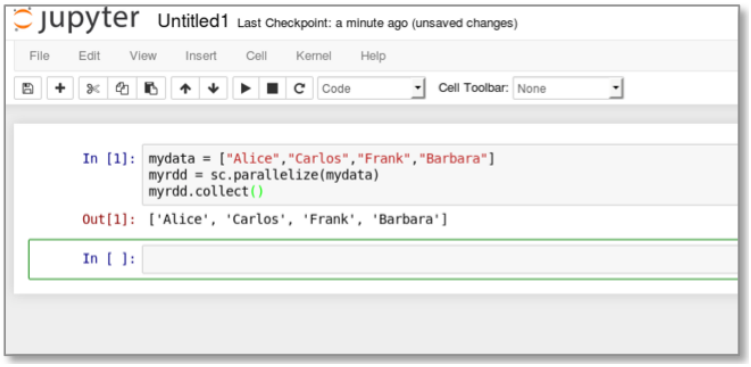
8. On the right hand side of the page select **Python 2** from the **New** menu.



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



10. Notice the output displayed.



11. To stop the Spark notebook server, enter `Ctrl+C` in the terminal running Spark.

This is the end of the exercise.

Appendix: Troubleshooting Tips

This appendix offers some tips to handle problems you may encounter while completing exercises.

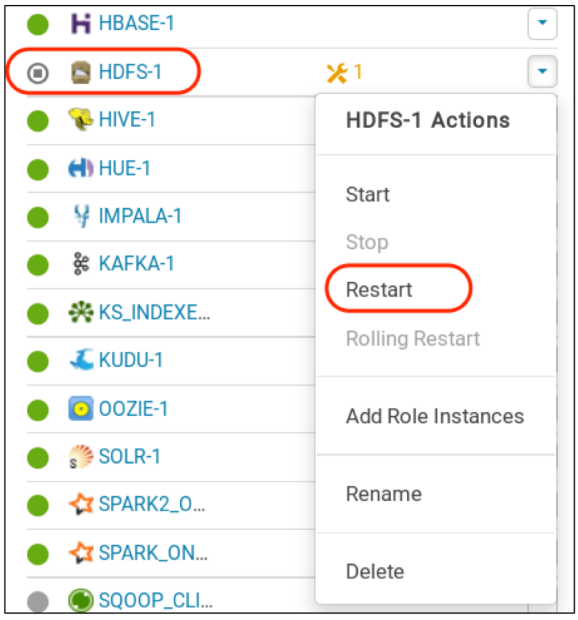
Services on the cluster are unavailable

You may get errors in Hue or on the command line about being unable to connect to services such as Hue, YARN, HDFS, ZooKeeper, Kafka, Hive, or other required services. In this case, the services may need to be restarted using Cloudera Manager.

1. Use the Cloudera Manager bookmark on your remote desktop browser to view the Cloudera Manager web UI, and log in using username **admin** with password **admin**.

2. If any of the cluster services you need for the exercises are shown with anything *other* than a green dot (such as a gray or red dot), restart the service by clicking on the dropdown menu next to the service name and selecting **Restart**.

   This screenshot shows an example in which the **HDFS-1** service is stopped, and how you would restart it.



3. After restarting the service, you may find that other services that depend on the restarted service also need restarting, which is indicated by an icon next to the service name. For example, Hue depends on HDFS, so after restarting HDFS, you would need to restart Hue following the same steps. The screenshot below shows the icon indicating that a service restart is required.

Cloudera Manager displays unhealthy status of services

If you just restarted the cluster from a stopped state, it can take a few minutes for bad health status indicators to resolve themselves. However, if health issues persist, here are some ways to determine what might be causing the health issues to appear, and how to resolve them.

Click on **All Health Issues**, then click on **Organize by Health Test**.

- If you have **Process Status** issues where the Cloudera Manager agent is not responding (as indicated by **Hosts** with unhealthy status), try restarting the cluster services. From the desktop, select **Applications** > **Training** > **Start Cluster** to restart your cluster services. Give the script time to complete.
- If you have any type of **Canary** issues, these typically clear up on their own, given time.
- If any other issues still exist after solving any process status issues:
  - In Cloudera Manager, note the name of one of the services reporting the issue (such as HDFS).
  - From the **Clusters** menu, choose that service (such as HDFS).
  - From the **Actions** menu, choose **Restart**.

You Forgot Hue Username or Password for `training` user

If you are unable to log in to Hue, it may be due to a misremembered username or password.

To determine if the username is correctly set to `training`:

1. Open a new terminal window.
2. Display the name of the administrative user, which should be `training`.

```
$ mysql -uroot -ptraining -e 'SELECT username from
 hue.auth_user'
```

If the username is incorrect, reset the password as below if you do not remember it, then go the following section to create a new user called `training`.

3. Reset the password for all users to `training`. The entire command must be entered on a single line. Be sure to type the value of the password as shown. That is the secret key to use for the `training` password.

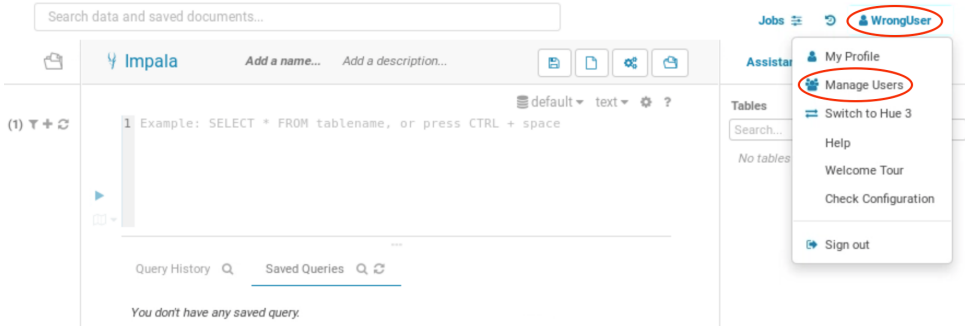
```
$ mysql -uroot -e 'UPDATE hue.auth_user SET
 password="pbkdf2_sha256$12000$P5tN0qCcUksb
$HlQ3tWzlrPHRU94oUgwOVbyVxCTwvxyfRlDDGSZAjUw="'
```

You created the wrong user name in Hue

The exercise environment setup script created system user called **training** for use in the exercises. Your Hue username must also be called **training**. If you created a different user when you first logged into Hue, you will need to create another user with the correct name to complete the exercises.

1. Log in to Hue using your existing username and password.

2. Select the **Manage Users** item on the user menu in the upper-right hand corner of the screen.



3. Click the **Manage Users** button.

4. Next, click the **Add User** button on the right-hand side of the screen.

5. In the **Step 1** tab, enter the correct credentials: Username: `training` and Password: `training`. Uncheck the box labeled **Create home directory**.

6. Skip step 2 by clicking on the **Step 3** tab. Check the box labeled **Superuser status**.

7. Click the **Add User** button at the bottom of the page.

8. Log out of Hue, and log back in as user **training**.

Hue hangs while loading a page

When you connect to Hue, you may occasionally find that the home page displays, but a spinner indicates that Hue is having trouble confirming that all services are correctly configured and running, or you may get a **Check config failed** error message.

Workaround 1: Try waiting for a few moments. Sometimes the configuration check takes a while but eventually completes, either with a confirmation that all services are working, or that one or more services might be misconfigured. If the misconfigured services are not ones that are required for the exercises (such as HBase or Oozie), you can continue with the exercise steps.

Workaround 2: Usually when the configuration check has not completed or warns of a misconfigured service, the rest of Hue will still work correctly. Try completing the exercise by going to the Hue page you want to use, such as the **Query Editor** or **File Browser**. If those functions work, you can continue with the exercise steps.

Solution: If the workarounds above are not helpful, you may need to restart the Hue service using Cloudera Manager. Refer to the section above called "Services on the cluster are unavailable". Follow those steps to restart the **HUE-1** service, even if the service is displayed with a healthy (green dot) indicator in Cloudera Manager. When the service is restarted, reload the Hue page in your browser.

HDFS error running catchup script

```
WARN hdfs.DFSClient: Caught exception
java.lang.InterruptedException
```

You may get this warning from the catchup script. This is just a warning, and you can disregard it.

`BindException` When Starting the Spark Shell

You may see a Spark warning like this when starting either the Python or Scala Spark shell:

```
FAILED org.spark-project.jetty.server.Server@69419d59:
 java.net.BindException: Address already in use
```

This is usually because you are attempting to run two instances of the Spark shell at the same time.

To fix this issue, exit one of your two running Spark shells.

If you do not have a terminal window running a second Spark shell, you may have one running in the background. View the applications running on the YARN cluster using the Hue Job Browser. Check the start times to determine which application is the one you want to keep running. Select the other one and click **kill**.

Spark job is accepted but never runs

When you execute a Spark application on YARN, you may see several messages showing that the job is ACCEPTED:

```
INFO yarn.Client: Application report for application_xxxx
 (state: ACCEPTED)
```

After a few seconds, you should be notified that the job status is now RUNNING. If the ACCEPTED message keeps displaying and the application or query never executes, this means that YARN has scheduled the job to run when cluster resources are availble, but none (or too few) resources are available.

Cause: This usually happens if you are running multiple Spark applications (such as two Spark shells, or a shell and an application) at the same time. It can also mean that a Spark application has crashed or exited without releasing its cluster resources.

Fix: Stop running one of the Spark applications. If you cannot finding a running application, use the Hue job browser to see what applications are running on the YARN cluster, and kill the one you do not need.