



Cloudera Custom Training Hands-On Exercises

General Notes.....	3
Hands-On Exercise: Data Ingest With Hadoop Tools.....	6
Hands-On Exercise: Running Queries from the Shell, Scripts, and Hue	13
Hands-On Exercise: Data Management	17
Hands-On Exercise: Relational Analysis	24
Hands-On Exercise: Working with Impala	26
Hands-On Exercise: Analyzing Text and Complex Data With Hive	29
Hands-On Exercise: Data Transformation with Hive	35
Hands-On Exercise: View the Spark Documentation.....	43
Hands-On Exercise: Use the Spark Shell	44
Hands-On Exercise: Use RDDs to Transform a Dataset.....	46
Hands-On Exercise: Process Data Files with Spark.....	53
Hands-On Exercise: Use Pair RDDs to Join Two Datasets	57

Hands-On Exercise: Write and Run a Spark Application.....	62
Hands-On Exercise: Configure a Spark Application	67
Hands-On Exercise: View Jobs and Stages in the Spark Application UI.....	71
Hands-On Exercise: Persist an RDD.....	77
Hands-On Exercise: Implement an Iterative Algorithm	79
Hands-On Exercise: Use Broadcast Variables	83
Hands-On Exercise: Use Accumulators	84
Hands-On Exercise: Use Spark SQL for ETL	85
Appendix A: Enabling iPython Notebook	89
Data Model Reference	91
Regular Expression Reference	95

General Notes

Cloudera's training courses use a virtual machine (VM) with a recent version of CDH already installed and configured for you. The VM runs in pseudo-distributed mode, a configuration that enables a Hadoop cluster to run on a single machine.

Points to Note While Working in the VM

1. The VM is set to automatically log in as the user `training`. If you log out, you can log back in as the user `training` with the password `training`. The root password is also `training`, though you can prefix any command with `sudo` to run it as root.
2. Exercises often contain steps with commands that look like this:

```
$ hdfs dfs -put accounting_reports_taxyear_2013 \  
/user/training/tax_analysis/
```

The `$` symbol represents the command prompt. Do *not* include this character when copying and pasting commands into your terminal window. Also, the backslash (`\`) signifies that the command continues on the next line. You may either enter the code as shown (on two lines), or omit the backslash and type the command on a single line.

Some commands are to be executed in the Python or Scala Spark Shells; those are color coded and shown with `pyspark>` (blue) or `scala>` (red) prompts, respectively. Linux command steps that apply to only one language or the other are also color coded, but still preceded with the `$` prompt.

3. Although many students are comfortable using UNIX text editors like `vi` or `emacs`, some might prefer a graphical text editor. To invoke the graphical editor from the command line, type `gedit` followed by the path of the file you wish to edit. Appending `&` to the command allows you to type additional commands while the editor is still open. Here is an example of how to edit a file named `myfile.txt`:

```
$ gedit myfile.txt &
```

Class-Specific VM Customization

Your VM is used in several of Cloudera's training classes. This particular class does not require some of the services that start by default, while other services that do not start by default are required for this class. Before starting the course exercises, run the course setup script:

```
$ ~/scripts/analyst/training_setup_da.sh
```

You may safely ignore any messages about services that have already been started or shut down. You only need to run this script once.

Points to Note During the Exercises

Sample Solutions

If you need a hint or want to check your work, the `sample_solution` subdirectory within each exercise directory contains complete code samples.

Catch-up Script

If you are unable to complete an exercise, we have provided a script to catch you up automatically. Each exercise has instructions for running the catch-up script.

\$ADIR Environment Variable

\$ADIR is a shortcut that points to the `/home/training/training_materials/analyst` directory, which contains the code and data you will use in the exercises.

Fewer Step-by-Step Instructions as You Work Through These Exercises

As the exercises progress, and you gain more familiarity with the tools you're using, we provide fewer step-by-step instructions. You should feel free to ask your instructor for assistance at any time, or to consult with your fellow students.

Bonus Exercises

Many of the exercises contain one or more optional “bonus” sections. We encourage you to work through these if time remains after you finish the main exercise and would like an additional challenge to practice what you have learned.

Hands-On Exercise: Data Ingest With Hadoop Tools

In this exercise you will practice using the Hadoop command line utility to interact with Hadoop's Distributed Filesystem (HDFS) and use Sqoop to import tables from a relational database to HDFS.

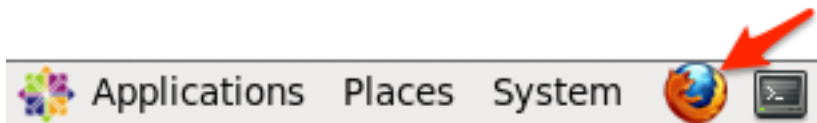
To begin, you must launch the Data Analyst VM.

Be sure you have run the setup script as described in the General Notes section above. If you have not run it yet, do so now:

```
$ ~/scripts/analyst/training_setup_da.sh
```

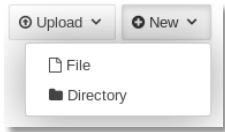
Step 1: Exploring HDFS using the Hue File Browser

1. Start the Firefox Web browser on your VM by clicking on the icon in the system toolbar:

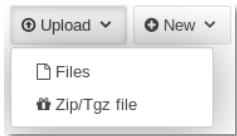


2. In Firefox, click on the **Hue** bookmark in the bookmark toolbar (or type `http://localhost:8888/home` into the address bar and then hit the [Enter] key.)
3. After a few seconds, you should see Hue's home screen. The first time you log in, you will be prompted to create a new username and password. Enter `training` in both the username and password fields, and then click the "Sign In" button.
4. Whenever you log in to Hue a Tips popup will appear. To stop it appearing in the future, check the **Do not show this dialog again** option before dismissing the popup.

5. Click **File Browser** in the Hue toolbar. Your HDFS home directory (`/user/training`) displays. (Since your user ID on the cluster is `training`, your home directory in HDFS is `/user/training`.) The directory contains no files or directories yet.
6. Create a temporary sub-directory: select the **+New** menu and click **Directory**.



7. Enter directory name **test** and click the **Create** button. Your home directory now contains a directory called **test**.
8. Click on **test** to view the contents of that directory; currently it contains no files or subdirectories.
9. Upload a file to the directory by selecting **Upload → Files**.



10. Click **Select Files** to bring up a file browser. By default, the `/home/training/Desktop` folder displays. Click the home directory button (**training**) then navigate to the course data directory: `training_materials/analyst/data`.
11. Choose any of the data files in that directory and click the **Open** button.
12. The file you selected will be loaded into the current HDFS directory. Click the filename to see the file's contents. Because HDFS is design to store very large files, Hue will not display the entire file, just the first page of data. You can click the arrow buttons or use the scrollbar to see more of the data.
13. Return to the `test` directory by clicking **View file location** in the left hand panel.
14. Above the list of files in your current directory is the full path of the directory you are currently displaying. You can click on any directory in the path, or on the first slash (`/`)

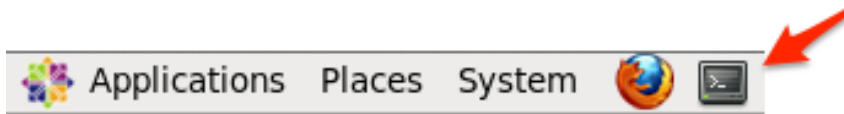
to go to the top level (root) directory. Click **training** to return to your home directory.



15. Delete the temporary `test` directory you created, including the file in it, by selecting the checkbox next to the directory name then clicking the **Move to trash** button. (Confirm that you want to delete by clicking **Yes**.)

Step 2: Exploring HDFS using the command line

4. You can use the `hdfs dfs` command to interact with HDFS from the command line. Close or minimize Firefox, then open a terminal window by clicking the icon in the system toolbar:



16. In the terminal window, enter:

```
$ hdfs dfs
```

This displays a help message describing all subcommands associated with `hdfs dfs`.

17. Run the following command:

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

This lists the contents of the HDFS root directory. One of the directories listed is `/user`. Each user on the cluster has a 'home' directory below `/user` corresponding to his or her user ID.

18. If you do not specify a path, `hdfs dfs` assumes you are referring to your home directory:


```
$ hdfs dfs -ls
```

19. Note the `/dualcore` directory. Most of your work in this course will be in that directory. Try creating a temporary subdirectory in `/dualcore`:

```
$ hdfs dfs -mkdir /dualcore/test1
```

20. Next, add a Web server log file to this new directory in HDFS:

```
$ hdfs dfs -put $ADIR/data/access.log /dualcore/test1/
```

Overwriting Files in Hadoop

Unlike the UNIX shell, Hadoop won't overwrite files and directories. This feature helps protect users from accidentally replacing data that may have taken hours to produce. If you need to replace a file or directory in HDFS, you must first remove the existing one. Please keep this in mind in case you make a mistake and need to repeat a step during the Hands-On Exercises.

To remove a file:

```
$ hdfs dfs -rm /dualcore/example.txt
```

To remove a directory and all its files and subdirectories (recursively):

```
$ hdfs dfs -rm -r /dualcore/example/
```

21. Verify the last step by listing the contents of the `/dualcore/test1` directory. You should observe that the `access.log` file is present and occupies 106,339,468 bytes of space in HDFS:

```
$ hdfs dfs -ls /dualcore/test1
```

22. Remove the temporary directory and its contents:

```
$ hdfs dfs -rm -r /dualcore/test1
```

Step 3: Importing Database Tables into HDFS with Sqoop

Dualcore stores information about its employees, customers, products, and orders in a MySQL database. In the next few steps, you will examine this database before using Sqoop to import its tables into HDFS.

5. In a terminal window, log in to MySQL and select the `dualcore` database:

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

23. Next, list the available tables in the `dualcore` database (**`mysql>`** represents the MySQL client prompt and is not part of the command):

```
mysql> SHOW TABLES;
```

24. Review the structure of the `employees` table and examine a few of its records:

```
mysql> DESCRIBE employees;
mysql> SELECT emp_id, fname, lname, state, salary FROM
employees LIMIT 10;
```

25. Exit MySQL by typing `quit`, and then hit the enter key:

```
mysql> quit
```

Data Model Reference

For your convenience, you will find a reference section depicting the structure for the tables you will use in the exercises at the end of this Exercise Manual.

26. Next, run the following command, which imports the `employees` table into the `/dualcore` directory created earlier using tab characters to separate each field:

```
$ sqoop import \  
  --connect jdbc:mysql://localhost/dualcore \  
  --username training --password training \  
  --fields-terminated-by '\t' \  
  --warehouse-dir /dualcore \  
  --table employees
```

Hiding Passwords

Typing the database password on the command line is a potential security risk since others may see it. An alternative to using the `--password` argument is to use `-P` and let Sqoop prompt you for the password, which is then not visible when you type it.

Sqoop Code Generation

After running the `sqoop import` command above, you may notice a new file named `employee.java` in your local directory. This is an artifact of Sqoop's code generation and is really only of interest to Java developers, so you can ignore it.

27. Revise the previous command and import the `customers` table into HDFS.
28. Revise the previous command and import the `products` table into HDFS.
29. Revise the previous command and import the `orders` table into HDFS.

30. Next, import the `order_details` table into HDFS. The command is slightly different because this table only holds references to records in the `orders` and `products` table, and lacks a primary key of its own. Consequently, you will need to specify the `--split-by` option and instruct Sqoop to divide the import work among tasks based on values in the `order_id` field. An alternative is to use the `-m 1` option to force Sqoop to import all the data with a single task, but this would significantly reduce performance.

```
$ sqoop import \  
--connect jdbc:mysql://localhost/dualcore \  
--username training --password training \  
--fields-terminated-by '\t' \  
--warehouse-dir /dualcore \  
--table order_details \  
--split-by=order_id
```

This is the end of the Exercise

Hands-On Exercise: Running Queries from the Shell, Scripts, and Hue

Exercise directory: `$ADIR/exercises/queries`

In this exercise you will practice using the Hue query editor and the Impala and Hive shells to execute simple queries. These exercises use the tables that have been populated with data you imported to HDFS using Sqoop in the “Data Ingest With Hadoop Tools” exercise.

IMPORTANT: In order to prepare the data for this exercise, you must run the following command before continuing:

```
$ ~/scripts/analyst/catchup.sh
```


Step #1: Explore the `customers` table using Hue

One way to run Impala and Hive queries is through your Web browser using Hue’s Query Editors. This is especially convenient if you use more than one computer – or if you use a device (such as a tablet) that isn’t capable of running the Impala or Beeline shells itself – because it does not require any software other than a browser.

6. Start the Firefox Web browser if it isn’t running, then click on the Hue bookmark in the Firefox bookmark toolbar (or type `http://localhost:8888/home` into the address bar and then hit the [Enter] key.)
7. After a few seconds, you should see Hue’s home screen. If you don’t currently have an active session, you will first be prompted to log in. Enter `training` in both the username and password fields, and then click the **Sign In** button.
8. Select the **Query Editors** menu in the Hue toolbar. Note that there are query editors for both Impala and Hive (as well as other tools such as Pig.) The interface is very similar for both Hive and Impala. For these exercises, select the **Impala** query editor.

9. This is the first time we have run Impala since we imported the data using Sqoop. Tell Impala to reload the HDFS metadata for the table by entering the following command in the query area, then clicking **Execute**.

```
INVALIDATE METADATA
```

10. Make sure the `default` database is selected in the database list on the left side of the page.
11. Below the selected database is a list of the tables in that database. Select the **customers** table to view the columns in the table.
12. Click the Preview Sample Data icon () next to the table name to view sample data from the table. When you are done, click to OK button to close the window.

Step #2: Run a Query Using Hue

Dualcore ran a contest in which customers posted videos of interesting ways to use their new tablets. A \$5,000 prize will be awarded to the customer whose video received the highest rating.

However, the registration data was lost due to an RDBMS crash, and the only information we have is from the videos. The winning customer introduced herself only as “Bridget from Kansas City” in her video.

You will need to run a query that identifies the winner’s record in our customer database so that we can send her the \$5,000 prize.

13. All you know about the winner is that her name is Bridget and she lives in Kansas City. In the Impala Query Editor, enter a query in the text area to find the winning customer. Use the `LIKE` operator to do a wildcard search for names such as "Bridget", "Bridgette" or "Bridgitte". Remember to filter on the customer’s city.
14. After entering the query, click the **Execute** button.

While the query is executing, the **Log** tab displays ongoing log output from the query. When the query is complete, the **Results** tab opens, displaying the results of the query.

Question: Which customer did your query identify as the winner of the \$5,000 prize?

Step #3: Run a Query from the Impala Shell

Run a top-N query to identify the three most expensive products that Dualcore currently offers.

15. Start a terminal window if you don't currently have one running.

16. On the Linux command line in the terminal window, start the Impala shell:

```
$ impala-shell
```

Impala displays the URL of the Impala server in the shell command prompt, e.g.:

```
[localhost.localdomain:21000] >
```

17. At the prompt, review the schema of the products table by entering

```
DESCRIBE products;
```

Remember that SQL commands in the shell must be terminated by a semi-colon (;), unlike in the Hue query editor.

18. Show a sample of 10 records from the products table:

```
SELECT * FROM products LIMIT 10;
```

19. Execute a query that displays the three most expensive products. **Hint:** Use `ORDER BY`.

20. When you are done, exit the Impala shell:

```
exit;
```

Step #4: Run a Script in the Impala Shell

The rules for the contest described earlier require that the winner bought the advertised tablet from Dualcore between May 1, 2013 and May 31, 2013. Before we can authorize our accounting department to pay the \$5,000 prize, you must ensure that Bridget is eligible.

Since this query involves joining data from several tables, and we have not yet covered JOIN, you've been provided with a script in the exercise directory.

21. Change to the directory for this hands-on exercise:

```
$ cd $ADIR/exercises/queries
```

22. Review the code for the query:

```
$ cat verify_tablet_order.sql
```

23. Execute the script using the shell's -f option:

```
$ impala-shell -f verify_tablet_order.sql
```

Question: Did Bridget order the advertised tablet in May?

Step #5: Run a Query Using Beeline

24. At the Linux command line in a terminal window, start Beeline:

```
$ beeline -u jdbc:hive2://localhost:10000
```

Beeline displays the URL of the Hive server in the shell command prompt, e.g.:

```
0: jdbc:hive2://localhost:10000>
```

25. Execute a query to find all the Gigabux brand products whose price is less than 1000 (\$10).

26. Exit the Beeline shell by entering

```
!exit
```

This is the end of the Exercise

Hands-On Exercise: Data Management

Exercise directory: `$ADIR/exercises/data_mgmt`

In this exercise you will practice using several common techniques for creating and populating tables.

IMPORTANT: This exercise builds on previous ones. If you were unable to complete any previous exercise or think you may have made a mistake, run the following command to prepare for this exercise before continuing:

```
$ ~/scripts/analyst/catchup.sh
```

Step #1: Review Existing Tables using the Metastore Manager

1. In Firefox, visit the Hue home page, and then choose **Data Browsers** → **Metastore Tables** in the Hue toolbar.
2. Make sure **default** database is selected.
3. Select the **customers** table to display the table browser and review the list of columns.
4. Select the **Sample** tab to view the first hundred rows of data.

Step #2: Create and Load a New Table using the Metastore Manager

Create and then load a table with product ratings data.

5. Before creating the table, review the files containing the product ratings data. The files are in `/home/training/training_materials/analyst/data`. You can use the `head` command in a terminal window to see the first few lines:

```
$ head $ADIR/data/ratings_2012.txt
$ head $ADIR/data/ratings_2013.txt
```

6. Copy the data files to the `/dualcore` directory in HDFS. You may use either the Hue File Browser, or the `hdfs` command in the terminal window:

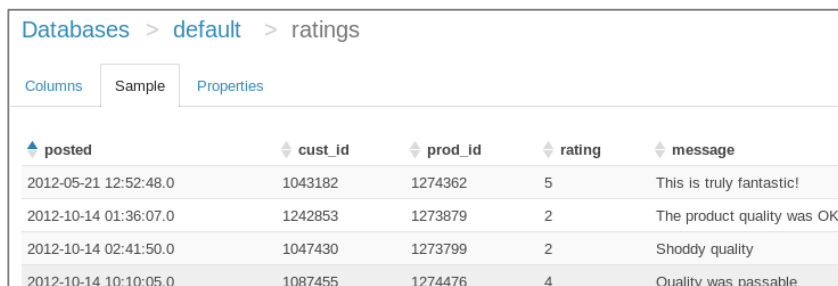
```
$ hdfs dfs -put $ADIR/data/ratings_2012.txt /dualcore/
$ hdfs dfs -put $ADIR/data/ratings_2013.txt /dualcore/
```

7. Return to the Metastore Manager in Hue. Select the **default** database to view the table browser.
8. Click on **Create a new table manually** to start the table definition wizard.
9. The first wizard step is to specify the table's name (required) and a description (optional). Enter table name `ratings`, then click **Next**.
10. In the next step you can choose whether the table will be stored as a regular text file or use a custom Serializer/Deserializer, or SerDe. SerDes will be covered later in the course. For now, select **Delimited**, then click **Next**.
11. The next step allows you to change the default delimiters. For a simple table, only the field terminator is relevant; collection and map delimiters are used for complex data in Hive, and will be covered later in the course. Select **Tab (\t)** for the field terminator, then click **Next**.
12. In the next step, choose a file format. File Formats will be covered later in the course. For now, select **TextFile**, then click **Next**.
13. In the next step, you can choose whether to store the file in the default data warehouse directory or a different location. Make sure the **Use default location** box is checked, then click **Next**.
14. The next step in the wizard lets you add columns. The first column of the ratings table is the timestamp of the time that the rating was posted. Enter column name `posted` and choose column type **timestamp**.

15. You can add additional columns by clicking the **Add a column** button. Repeat the steps above to enter a column name and type for all the columns for the ratings table:

Field Name	Field Type
posted	timestamp
cust_id	int
prod_id	int
rating	tinyint
message	string

16. When you have added all the columns, scroll down and click **Create table**. This will start a job to define the table in the Metastore, and create the warehouse directory in HDFS to store the data.
17. When the job is complete, the new table will appear in the table browser.
18. *Optional:* Use the Hue File Browser or the `hdfs` command to view the `/user/hive/warehouse` directory to confirm creation of the `ratings` subdirectory.
19. Now that the table is created, you can load data from a file. One way to do this is in Hue. Click **Import Table** under Actions.
20. In the Import data dialog box, enter or browse to the HDFS location of the 2012 product ratings data file: `/dualcore/ratings_2012.txt`. Then click **Submit**. (You will load the 2013 ratings in a moment.)
21. Next, verify that the data was loaded by selecting the Sample tab in the table browser for the ratings table. The results should look like this:




The screenshot shows the Hue interface for the 'ratings' table. The breadcrumb path is 'Databases > default > ratings'. There are three tabs: 'Columns', 'Sample', and 'Properties'. The 'Sample' tab is selected, displaying a table with five columns: 'posted', 'cust_id', 'prod_id', 'rating', and 'message'. The data rows show timestamps, customer IDs, product IDs, ratings, and feedback messages.

posted	cust_id	prod_id	rating	message
2012-05-21 12:52:48.0	1043182	1274362	5	This is truly fantastic!
2012-10-14 01:36:07.0	1242853	1273879	2	The product quality was OK
2012-10-14 02:41:50.0	1047430	1273799	2	Shoddy quality
2012-10-14 10:10:05.0	1087455	1274476	4	Quality was passable

22. Trying querying the data in the table. In Hue, switch to the Impala Query Editor.

23. Initially the new table will not appear. You must first reload Impala's metadata cache by entering and executing the command below. (Impala metadata caching will be covered in depth later in the course.)

```
INVALIDATE METADATA;
```

24. If the table does not appear in the table list on the left, click the Reload button: . (This refreshes the page, not the metadata itself.)

25. Try executing a query, such as counting the number of ratings:

```
SELECT COUNT(*) FROM ratings;
```

The total number of records should be 464.

26. Another way to load data into a table is using the `LOAD DATA` command. Load the 2013 ratings data:

```
LOAD DATA INPATH '/dualcore/ratings_2013.txt' INTO TABLE  
ratings;
```

27. The `LOAD DATA INPATH` command *moves* the file to the table's directory. Using the Hue File Browser or `hdfs` command, verify that the file is no longer present in the original directory:

```
$ hdfs dfs -ls /dualcore/ratings_2013.txt
```

28. *Optional:* Verify that the 2013 data is shown alongside the 2012 data in the table's warehouse directory.
29. Finally, count the records in the ratings table to ensure that all 21,997 are available:

```
SELECT COUNT(*) FROM ratings;
```

Step #3: Create an External Table Using CREATE TABLE

You imported data from the `employees` table in MySQL into HDFS in an earlier exercise. Now we want to be able to query this data. Since the data already exists in HDFS, this is a good opportunity to use an external table.

In the last exercise you practiced creating a table using the Metastore Manager; this time, use an Impala SQL statement. You may use either the Impala shell, or the Impala Query Editor in Hue.

30. Write and execute a `CREATE TABLE` statement to create an *external* table for the tab-delimited records in HDFS at `/dualcore/employees`. The data format is shown below:

Field Name	Field Type
<code>emp_id</code>	STRING
<code>fname</code>	STRING
<code>lname</code>	STRING
<code>address</code>	STRING
<code>city</code>	STRING
<code>state</code>	STRING
<code>zipcode</code>	STRING
<code>job_title</code>	STRING
<code>email</code>	STRING
<code>active</code>	STRING
<code>salary</code>	INT

31. Run the following query to verify that you have created the table correctly.

```
SELECT job_title, COUNT(*) AS num
  FROM employees
 GROUP BY job_title
 ORDER BY num DESC
 LIMIT 3;
```

It should show that Sales Associate, Cashier, and Assistant Manager are the three most common job titles at Dualcore.

Bonus Exercise #1: Use Sqoop's Hive Import Option to Create a Table

If you have successfully finished the main exercise and still have time, feel free to continue with this bonus exercise.

You used Sqoop in an earlier exercise to import data from MySQL into HDFS. Sqoop can also create a Hive/Impala table with the same fields as the source table in addition to importing the records, which saves you from having to write a `CREATE TABLE` statement.

32. In a terminal window, execute the following command to import the `suppliers` table from MySQL as a new managed table:

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

33. It is always a good idea to validate data after adding it. Execute the following query to count the number of suppliers in Texas. You may use either the Impala shell or the Hue Impala Query Editor. Remember to invalidate the metadata cache so that Impala can find the new table.

```
INVALIDATE METADATA;  
SELECT COUNT(*) FROM suppliers WHERE state='TX';
```

The query should show that nine records match.

Bonus Exercise #2: Alter a Table

If you have successfully finished the main exercise and still have time, feel free to continue with this bonus exercise. You can compare your work against the files found in the `bonus_02/sample_solution/` subdirectory.

In this exercise you will modify the `suppliers` table you imported using Sqoop in the previous exercise. You may complete these exercises using either the Impala shell or the Impala query editor in Hue.

34. Use `ALTER TABLE` to rename the `company` column to `name`.
35. Use the `DESCRIBE` command on the `suppliers` table to verify the change.
36. Use `ALTER TABLE` to rename the entire table to `vendors`.
37. Although the `ALTER TABLE` command often requires that we make a corresponding change to the data in HDFS, renaming a table or column does not. You can verify this by running a query on the table using the new names, e.g.:

```
SELECT supp_id, name FROM vendors LIMIT 10;
```

This is the end of the Exercise

Hands-On Exercise: Relational Analysis

Exercise directory: `$ADIR/exercises/relational_analysis`

In this exercise you will write queries to analyze data in tables that have been populated with data you imported to HDFS using Sqoop in the “Data Ingest” exercise.

IMPORTANT: In order to prepare the data for this exercise, you must run the following command before continuing:

```
$ ~/scripts/analyst/catchup.sh
```

Several analysis questions are described below and you will need to write the SQL code to answer them. You can use whichever tool you prefer – Impala or Hive – using whichever method you like best, including shell, script, or the Hue Query Editor, to run your queries.

Step #1: Calculate Top N Products

- Which top three products has Dualcore sold more of than any other?
Hint: Remember that if you use a `GROUP BY` clause, you must group by all fields listed in the `SELECT` clause that are not part of an aggregate function.

Step #2: Calculate Order Total

- Which orders had the highest total?

Step #3: Calculate Revenue and Profit

- Write a query to show Dualcore’s revenue (total price of products sold) and profit (price minus cost) by date.
 - Hint: The `order_date` column in the `order_details` table is of type `TIMESTAMP`. Use `TO_DATE` to get just the date portion of the value.

There are several ways you could write these queries. One possible solution for each is in the `sample_solution/` directory.

Bonus Exercise #1: Rank Daily Profits by Month

If you have successfully finished the earlier steps and still have time, feel free to continue with this optional bonus exercise.

- Write a query to show how each day's profit ranks compared to other days within the same year and month.
 - Hint: Use the previous exercise's solution as a sub-query; find the `ROW_NUMBER` of the results within each year and month

There are several ways you could write this query. One possible solution is in the `bonus_01/sample_solution/` directory.

This is the end of the Exercise

Hands-On Exercise: Working with Impala

In this exercise you will explore the query execution plan for various types of queries in Impala.

IMPORTANT: This exercise builds on previous ones. If you were unable to complete any previous exercise or think you may have made a mistake, run the following command to prepare for this exercise before continuing:

```
$ ~/scripts/analyst/catchup.sh
```

Step #1: Review Query Execution Plans

1. Review the execution plan for the following query. You may use either the Impala Query Editor in Hue or the Impala shell command line tool.

```
SELECT * FROM products;
```

2. Note that the query explanation includes a warning that table and column stats are not available for the products table. Compute the stats by executing

```
COMPUTE STATS products;
```

3. Now view the query plan again, this time without the warning.
4. The previous query was a very simple query against a single table. Try reviewing the query plan of a more complex query. The following query returns the top 3 products sold. Before EXPLAINing the query, compute stats on the tables to be queried.

```
SELECT brand, name, COUNT(p.prod_id) AS sold
FROM products p
JOIN order_details d
ON (p.prod_id = d.prod_id)
GROUP BY brand, name, p.prod_id
ORDER BY sold DESC
LIMIT 3;
```

Questions: How many stages are there in this query? What are the estimated per-host memory requirements for this query? What is the total size of all partitions to be scanned?

5. The tables in the queries above are all have only a single partition. Try reviewing the query plan for a partitioned table. Recall that in the “Data Storage and Performance” exercise, you created an `ads` table partitioned on the `network` column. Compare the query plans for the following two queries. The first calculates the total cost of *clicked ads* each ad campaign; the second does the same, but for *all* ads on *one* of ad networks.

```
SELECT campaign_id, SUM(cpc)
FROM ads
WHERE was_clicked=1
GROUP BY campaign_id
ORDER BY campaign_id;
```

```
SELECT campaign_id, SUM(cpc)
FROM ads
WHERE network=1
GROUP BY campaign_id
ORDER BY campaign_id;
```

Questions: What is the estimate per-host memory requirements for the two queries? What explains the difference?

Bonus Exercise #1: Review the Query Summary

If you have successfully finished the earlier steps and still have time, feel free to continue with this optional bonus exercise.

This exercise must be completed in the Impala Shell command line tool, because it uses features not yet available in Hue. Refer to the “Running Queries from the Shell, Scripts, and Hue” exercise for how to use the shell if needed.

6. Try executing one of the queries you examined above, e.g.

```
SELECT brand, name, COUNT(p.prod_id) AS sold
FROM products p
JOIN order_details d
ON (p.prod_id = d.prod_id)
GROUP BY brand, name, p.prod_id
ORDER BY sold DESC
LIMIT 3;
```

7. After the query completes, execute the SUMMARY command:

```
SUMMARY;
```

8. **Questions:** Which stage took the longest average time to complete? Which took the most memory?

This is the end of the Exercise

Hands-On Exercise: Analyzing Text and Complex Data With Hive

Exercise directory: `$ADIR/exercises/complex_data`

In this exercise, you will

- Use Hive's ability to store complex data to work with data from a customer loyalty program
- Use a Regex SerDe to load weblog data into Hive
- Use Hive's text processing features to analyze customers' comments and product ratings, uncover problems and propose potential solutions.

IMPORTANT: This exercise builds on previous ones. If you were unable to complete any previous exercise or think you may have made a mistake, run the following command to prepare for this exercise before continuing:

```
$ ~/scripts/analyst/catchup.sh
```

Step #1: Create, Load and Query a Table with Complex Data

Dualcore recently started a loyalty program to reward our best customers. A colleague has already provided us with a sample of the data that contains information about customers who have signed up for the program, including their phone numbers (as a map), a list of past order IDs (as an array), and a struct that summarizes the minimum, maximum, average, and total value of past orders. You will create the table, populate it with the provided data, and then run a few queries to practice referencing these types of fields.

You may use either the Beeline shell or Hue's Hive Query Editor to complete these exercises.

1. Create a table with the following characteristics:

Name: `loyalty_program`

Type: EXTERNAL

Columns:

Field Name	Field Type
<code>cust_id</code>	STRING
<code>fname</code>	STRING
<code>lname</code>	STRING
<code>email</code>	STRING
<code>level</code>	STRING
<code>phone</code>	MAP<STRING, STRING>
<code>order_ids</code>	ARRAY<INT>
<code>order_value</code>	STRUCT<min:INT, max:INT, avg:INT, total:INT>

Field Terminator: `|` (vertical bar)

Collection item terminator: `,` (comma)

Map Key Terminator: `:` (colon)

Location: `/dualcore/loyalty_program`

2. Examine the data in `$ADIR/data/loyalty_data.txt` to see how it corresponds to the fields in the table.
3. Load the data file by placing it into the HDFS data warehouse directory for the new table. You can use either the Hue File Browser, or the `hdfs` command:

```
$ hdfs dfs -put $ADIR/data/loyalty_data.txt \  
  /dualcore/loyalty_program/
```

4. Run a query to select the `HOME` phone number (hint: map keys are case-sensitive) for customer ID 1200866. You should see 408-555-4914 as the result.
5. Select the third element from the `order_ids` array for customer ID 1200866 (hint: elements are indexed from zero). The query should return 5278505.
6. Select the `total` attribute from the `order_value` struct for customer ID 1200866. The query should return 401874.

Step #2: Create and Populate the Web Logs Table

Many interesting analyses can be done on data from the usage of a web site. The first step is to load the semi-structured data in the web log files into a Hive table. Typical log file formats are not delimited, so you will need to use the RegexSerDe and specify a pattern Hive can use to parse lines into individual fields you can then query.

7. Examine the `create_web_logs.hql` script to get an idea of how it uses a RegexSerDe to parse lines in the log file (an example log line is shown in the comment at the top of the file). When you have examined the script, run it to create the table. You can paste the code into the Hive Query Editor, or use HCatalog:

```
$ hcat -f $ADIR/exercises/complex_data/create_web_logs.hql
```

8. Populate the table by adding the log file to the table's directory in HDFS:

```
$ hdfs dfs -put $ADIR/data/access.log /dualcore/web_logs/
```

9. Verify that the data is loaded correctly by running this query to show the top three items users searched for on our Web site:

```
SELECT term, COUNT(term) AS num FROM
  (SELECT LOWER(REGEXP_EXTRACT(request,
    '/search\\?phrase=(\\S+)', 1)) AS term
   FROM web_logs
   WHERE request REGEXP '/search\\?phrase=') terms
GROUP BY term
ORDER BY num DESC
LIMIT 3;
```

You should see that it returns `tablet (303)`, `ram (153)` and `wifi (148)`.

Note: The REGEXP operator, which is available in some SQL dialects, is similar to LIKE, but uses regular expressions for more powerful pattern matching. The REGEXP operator is synonymous with the RLIKE operator.

Bonus Exercise #1: Analyze Numeric Product Ratings

If you have successfully finished the earlier steps and still have time, feel free to continue with this optional bonus exercise.

Customer ratings and feedback are great sources of information for both customers and retailers like Dualcore.

However, customer comments are typically free-form text and must be handled differently. Fortunately, Hive provides extensive support for text processing.

Before delving into text processing, you'll begin by analyzing the numeric ratings customers have assigned to various products. In the next bonus exercise, you will use these results in doing text analysis.

10. Review the `ratings` table structure using the Hive Query Editor or using the `DESCRIBE` command in the Beeline shell.
11. We want to find the product that customers like most, but must guard against being misled by products that have few ratings assigned. Run the following query to find the product with the highest average among all those with at least 50 ratings:

```
SELECT prod_id, FORMAT_NUMBER(avg_rating, 2) AS avg_rating
FROM (SELECT prod_id, AVG(rating) AS avg_rating,
      COUNT(*) AS num
      FROM ratings
      GROUP BY prod_id) rated
WHERE num >= 50
ORDER BY avg_rating DESC
LIMIT 1;
```

12. Rewrite, and then execute, the query above to find the product with the *lowest* average among products with at least 50 ratings. You should see that the result is product ID 1274673 with an average rating of 1.10.

Bonus Exercise #2: Analyze Rating Comments

We observed earlier that customers are very dissatisfied with one of the products we sell. Although numeric ratings can help identify *which* product that is, they don't tell us *why* customers don't like the product. Although we could simply read through all the comments associated with that product to learn this information, that approach doesn't scale. Next, you will use Hive's text processing support to analyze the comments.

13. The following query normalizes all comments on that product to lowercase, breaks them into individual words using the `SENTENCES` function, and passes those to the `NGRAMS` function to find the five most common bigrams (two-word combinations). Run the query:

```
SELECT EXPLODE (NGRAMS (SENTENCES (LOWER (message)) , 2 , 5))
      AS bigrams
FROM ratings
WHERE prod_id = 1274673;
```

14. Most of these words are too common to provide much insight, though the word “expensive” does stand out in the list. Modify the previous query to find the five most common *trigrams* (three-word combinations), and then run that query in Hive.
15. Among the patterns you see in the result is the phrase “ten times more.” This might be related to the complaints that the product is too expensive. Now that you've identified a specific phrase, look at a few comments that contain it by running this query:

```
SELECT message
FROM ratings
WHERE prod_id = 1274673
      AND message LIKE '%ten times more%'
LIMIT 3;
```

You should see three comments that say, “Why does the red one cost ten times more than the others?”

16. We can infer that customers are complaining about the price of this item, but the comment alone doesn't provide enough detail. One of the words ("red") in that comment was also found in the list of trigrams from the earlier query. Write and execute a query that will find all distinct comments containing the word "red" that are associated with product ID 1274673.

17. The previous step should have displayed two comments:

18. "What is so special about red?"

19. "Why does the red one cost ten times more than the others?"

The second comment implies that this product is overpriced relative to similar products. Write and run a query that will display the record for product ID 1274673 in the `products` table.

20. Your query should have shown that the product was a "16GB USB Flash Drive (Red)" from the "Orion" brand. Next, run this query to identify similar products:

```
SELECT *  
  FROM products  
 WHERE name LIKE '%16 GB USB Flash Drive%'  
    AND brand='Orion';
```

The query results show that we have three almost identical products, but the product with the negative reviews (the red one) costs about ten times as much as the others, just as some of the comments said.

Based on the cost and price columns, it appears that doing text processing on the product ratings has helped us uncover a pricing error.

This is the end of the Exercise

Hands-On Exercise: Data Transformation with Hive

Exercise directory: \$ADIR/exercises/transform

In this exercise you will explore the data from Dualcore’s Web server that you loaded in the “Analyzing Text and Complex Data” exercise. Queries on that data will reveal that many customers abandon their shopping carts before completing the checkout process. You will create several additional tables, using data from a TRANSFORM script and a supplied UDF, which you will use later to analyze how Dualcore could turn this problem into an opportunity.

IMPORTANT: This exercise builds on previous ones. If you were unable to complete any previous exercise or think you may have made a mistake, run the following command to prepare for this exercise before continuing:

```
$ ~/scripts/analyst/catchup.sh
```

Step #1: Analyze Customer Checkouts

As on many Web sites, Dualcore’s customers add products to their shopping carts and then follow a “checkout” process to complete their purchase. We want to figure out if customers who start the checkout process are completing it. Since each part of the four-step checkout process can be identified by its URL in the logs, we can use a regular expression to identify them:

Step	Request URL	Description
1	/cart/checkout/step1-viewcart	View list of items added to cart
2	/cart/checkout/step2-shippingcost	Notify customer of shipping cost
3	/cart/checkout/step3-payment	Gather payment information
4	/cart/checkout/step4-receipt	Show receipt for completed order

Note: Because the `web_logs` table uses a Regex SerDes, which is a feature not supported by Impala, this step must be completed in Hive. You may use either the Beeline shell or the Hive Query Editor in Hue.

1. Run the following query in Hive to show the number of requests for each step of the checkout process:

```
SELECT COUNT(*), request
FROM web_logs
WHERE request REGEXP '/cart/checkout/step\\d.+'
GROUP BY request;
```

The results of this query highlight a major problem. About one out of every three customers abandons their cart after the second step. This might mean millions of dollars in lost revenue, so let's see if we can determine the cause.

2. The log file's `cookie` field stores a value that uniquely identifies each user session. Since not all sessions involve checkouts at all, create a new table containing the session ID and number of checkout steps completed for just those sessions that do:

```
CREATE TABLE checkout_sessions AS
SELECT cookie, ip_address, COUNT(request) AS steps_completed
FROM web_logs
WHERE request REGEXP '/cart/checkout/step\\d.+'
GROUP BY cookie, ip_address;
```

3. Run this query to show the number of people who abandoned their cart after each step:

```
SELECT steps_completed, COUNT(cookie) AS num
FROM checkout_sessions
GROUP BY steps_completed;
```

You should see that most customers who abandoned their order did so after the second step, which is when they first learn how much it will cost to ship their order.

4. *Optional:* Because the new `checkout_sessions` table does not use a SerDes, it can be queried in Impala. Try running the same query as in the previous step in Impala. What happens?

Step #2: Use TRANSFORM for IP Geolocation

Based on what you've just seen, it seems likely that customers abandon their carts due to high shipping costs. The shipping cost is based on the customer's location and the weight of the items they've ordered. Although this information isn't in the database (since the order wasn't completed), we can gather enough data from the logs to estimate them.

We don't have the customer's address, but we can use a process known as "IP geolocation" to map the computer's IP address in the log file to an approximate physical location. Since this isn't a built-in capability of Hive, you'll use a provided Python script to `TRANSFORM` the `ip_address` field from the `checkout_sessions` table to a ZIP code, as part of HiveQL statement that creates a new table called `cart_zipcodes`.

Regarding TRANSFORM and UDF Examples in this Exercise

During this exercise, you will use a Python script for IP geolocation and a UDF to calculate shipping costs. Both are implemented merely as a simulation – compatible with the fictitious data we use in class and intended to work even when Internet access is unavailable. The focus of these exercises is on how to *use* external scripts and UDFs, rather than how the code for the examples works internally.

5. Examine the `create_cart_zipcodes.hql` script and observe the following:
 - a. It creates a new table called `cart_zipcodes` based on select statement.
 - b. That select statement transforms the `ip_address`, `cookie`, and `steps_completed` fields from the `checkout_sessions` table using a Python script.
 - c. The new table contains the ZIP code instead of an IP address, plus the other two fields from the original table.
6. Examine the `ipgeolocator.py` script and observe the following:

- a. Records are read from Hive on standard input.
 - b. The script splits them into individual fields using a tab delimiter.
 - c. The `ip_addr` field is converted to `zipcode`, but the `cookie` and `steps_completed` fields are passed through unmodified.
 - d. The three fields in each output record are delimited with tabs and are printed to standard output.
7. Copy the Python file to HDFS so that the Hive Server can access it. You may use the Hue File Browser or the `hdfs` command:

```
$ hdfs dfs -put $ADIR/exercises/transform/ipgeolocator.py \
/dualcore/
```

8. Run the script to create the `cart_zipcodes` table. You can either paste the code into the Hive Query Editor, or use Beeline in a terminal window:

```
$ beeline -u jdbc:hive2://localhost:10000 \
-f $ADIR/exercises/transform/create_cart_zipcodes.hql
```

Step #3: Extract List of Products Added to Each Cart

As described earlier, estimating the shipping cost also requires a list of items in the customer's cart. You can identify products added to the cart since the request URL looks like this (only the product ID changes from one record to the next):

```
/cart/additem?productid=1234567
```

9. Write a HiveQL statement to create a table called `cart_items` with two fields: `cookie` and `prod_id` based on data selected from the `web_logs` table. Keep the following in mind when writing your statement:
- a. The `prod_id` field should contain only the seven-digit product ID (hint: use the `REGEXP_EXTRACT` function)

- b. Use a `WHERE` clause with `REGEXP` using the same regular expression as above, so that you only include records where customers are adding items to the cart.
- c. If you need a hint on how to write the statement, look at the `create_cart_items.hql` file in the exercise's `sample_solution` directory.

10. Verify the contents of the new table by running this query:

```
SELECT COUNT(DISTINCT cookie) FROM cart_items
WHERE prod_id=1273905;
```

If this doesn't return 47, then compare your statement to the `create_cart_items.hql` file, make the necessary corrections, and then re-run your statement (after dropping the `cart_items` table).

Step #4: Create Tables to Join Web Logs with Product Data

You now have tables representing the ZIP codes and products associated with checkout sessions, but you'll need to join these with the products table to get the weight of these items before you can estimate shipping costs. In order to do some more analysis later, we'll also include total selling price and total wholesale cost in addition to the total shipping weight for all items in the cart.

11. Run the following HiveQL to create a table called `cart_orders` with the information:

```
CREATE TABLE cart_orders AS
  SELECT z.cookie, steps_completed, zipcode,
         SUM(shipping_wt) AS total_weight,
         SUM(price) AS total_price,
         SUM(cost) AS total_cost
  FROM cart_zipcodes z
  JOIN cart_items i
    ON (z.cookie = i.cookie)
  JOIN products p
    ON (i.prod_id = p.prod_id)
  GROUP BY z.cookie, zipcode, steps_completed;
```

Step #5: Create a Table Using a UDF to Estimate Shipping Cost

We finally have all the information we need to estimate the shipping cost for each abandoned order. One of the developers on our team has already written, compiled, and packaged a Hive UDF that will calculate the shipping cost given a ZIP code and the total weight of all items in the order.

12. Before you can use a UDF, you must make it available to Hive. First, copy the file to HDFS so that the Hive Server can access it. You may use the Hue File Browser or the `hdfs` command:


```
$ hdfs dfs -put \  
$ADIR/exercises/transform/geolocation_udf.jar \  
/dualcore/
```

13. Next, register the function with Hive and provide the name of the UDF class as well as the alias you want to use for the function. Run the Hive command below to associate the UDF with the alias `CALC_SHIPPING_COST`:

```
CREATE TEMPORARY FUNCTION CALC_SHIPPING_COST  
AS 'com.cloudera.hive.udf.UDFCalcShippingCost'  
USING JAR 'hdfs://dualcore/geolocation_udf.jar';
```

14. Now create a new table called `cart_shipping` that will contain the session ID, number of steps completed, total retail price, total wholesale cost, and the estimated shipping cost for each order based on data from the `cart_orders` table:

```
CREATE TABLE cart_shipping AS  
SELECT cookie, steps_completed, total_price, total_cost,  
CALC_SHIPPING_COST(zipcode, total_weight) AS shipping_cost  
FROM cart_orders;
```

15. Finally, verify your table by running the following query to check a record:

```
SELECT * FROM cart_shipping WHERE cookie='100002920697';
```

This should show that session as having two completed steps, a total retail price of \$263.77, a total wholesale cost of \$236.98, and a shipping cost of \$9.09.

Note: The `total_price`, `total_cost`, and `shipping_cost` columns in the `cart_shipping` table contain the number of cents as integers. Be sure to divide results containing monetary amounts by 100 to get dollars and cents.

You may now shut down the Data Analyst VM and launch the Spark VM. Run the following script:

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

This is the end of the Exercise

Hands-On Exercise: View the Spark Documentation

In this exercise, you will familiarize yourself with the Spark documentation.

You must now shut down the Data Analyst VM and launch the Spark VM, if you have not already done so.

IMPORTANT: In order to prepare for this exercise, you must run the following command before continuing:

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

1. Start Firefox in your Virtual Machine and visit the Spark documentation on your local machine, using the provided bookmark or opening the URL
`file:/usr/lib/spark/docs/_site/index.html`
2. From the **Programming Guides** menu, select the **Spark Programming Guide**. Briefly review the guide. You may wish to bookmark the page for later review.
3. From the **API Docs** menu, select either **Scala** or **Python**, depending on your language preference. Bookmark the API page for use during class. Later exercises will refer you to this documentation.

This is the end of the Exercise

Hands-On Exercise: Use the Spark Shell

In this exercise, you will start the Spark Shell and view the Spark Context object.

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

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

Using the Python Spark Shell

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

```
$ pyspark
```

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

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

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

```
pyspark> sc
```

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

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

5. Using command completion, you can see all the available `SparkContext` methods: type `sc.` (`sc` followed by a dot) and then the [TAB] key.
6. You can exit the shell by hitting Ctrl-D or by typing `exit`.

Using the Scala Spark Shell

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

```
$ spark-shell
```

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

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

```
scala> sc
```

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

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

9. Using command completion, you can see all the available `SparkContext` methods: type `sc.` (`sc` followed by a dot) and then the [TAB] key.
10. You can exit the shell by hitting Ctrl-D or typing `exit`.

This is the end of the Exercise

Hands-On Exercise: Use RDDs to Transform a Dataset

Files and Data Used in This Exercise:

Data files (local):

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

Solutions:

```
~/training_materials/sparkdev/solutions/LogIPs.pyspark  
~/training_materials/sparkdev/solutions/LogIPs.scalaspark
```

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

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

Loading and Viewing a Text File

1. Review the simple text file we will be using by viewing (without editing) the file in a text editor. The file is located at:
`~/training_materials/sparkdev/data/frostroad.txt.`
2. Start the Spark Shell if you exited it from the previous exercise. You may use either Scala (`spark-shell`) or Python (`pyspark`).
3. Define an RDD to be created by reading in a simple test file. For **Python**, enter:

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

Or for **Scala**, enter:

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

- **Note:** For the remainder of the Hands-On Exercises, note the color coding and prompt in exercise text snippets to follow the instructions for whichever language you are using.)

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

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

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

The `count` operation causes the RDD to be materialized (created and populated), after which the result (23) is displayed. The example below shows the output for Pyspark (Scala produces the same result, but the output format will differ slightly):

```
Out[2]: 23
```

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

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

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

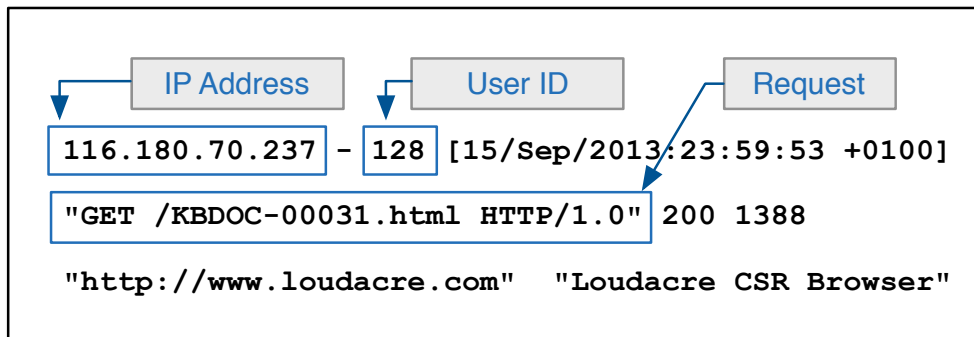
6. Using command completion, you can see all the available transformations and operations you can perform on an RDD. Type `mydata .` and then the [TAB] key.

Exploring the Loudacre Web Log Files

In this exercise, you will be using data in

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

7. Review one of the `.log` files in the directory. Note the format of the lines:



8. In the previous example you used a local datafile. In the real world, you will almost always be working with data on the HDFS cluster instead. Create an HDFS directory for the course, and then copy the dataset to your HDFS home directory. In a separate terminal window (not your Spark shell) execute:

```
$ hdfs dfs -mkdir /loudacre
$ hdfs dfs -put \
~/training_materials/sparkdev/data/weblogs/ \
/loudacre/
```

9. In your Spark Shell, set a variable for the data file so you do not have to retype it each time.

```
pyspark> logfile="/loudacre/weblogs/2013-09-15.log"
```

```
scala> val logfile="/loudacre/weblogs/2013-09-15.log"
```

10. Create an RDD from the data file.


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

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

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

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

```
scala> val jpglogs = logs.\n    filter(line => line.contains(".jpg"))
```

12. View the first 10 lines of the data using take:

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

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

13. Sometimes you do not need to store intermediate data in a variable, in which case you can combine the steps into a single line of code. For instance, if all you need is to count the number of JPG requests, you can execute this in a single command:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
pyspark> for x in ips.take(5): print x
```

```
scala> ips.take(5).foreach(println)
```

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

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

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

19. In a terminal window, list the contents of the HDFS `iplist` directory (in your HDFS home directory):

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

20. You should see multiple files. The one you care about is `part-00000`, which should contain the list of IP addresses. “Part” (partition) files are numbered because there may be results from multiple tasks running on the cluster; you will learn more about this later.

If You Have More Time

If you have more time, attempt the following challenges:

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

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

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

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

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

This is the end of the Exercise

Hands-On Exercise: Process Data Files with Spark

Files and Data Used in This Exercise:

Data files (local):

```
~/training_materials/sparkdev/data/activations/*  
~/training_materials/sparkdev/data/devicestatus.txt (Bonus)
```

Stubs:

```
stubs/ActivationModels.pyspark  
stubs/ActivationModels.scalaspark
```

Solutions:

```
solutions/ActivationModels.pyspark  
solutions/ActivationModels.scalaspark  
solutions/DeviceStatusETL.pyspark (Bonus)  
solutions/DeviceStatusETL.scalaspark (Bonus)
```

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

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

Reviewing the API Documentation for RDD Operations

Visit the Spark API page you bookmarked previously. Follow the link at the top for the RDD class and review the list of available methods.

The Data

1. Review the data in `activations` (in the course data directory). 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>
```

2. Copy this data to HDFS:

```
$ hdfs dfs -put \
~/training_materials/sparkdev/data/activations \
/loudacre/
```

The Task

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

The output will look something like:

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

3. Start with the `ActivationModels` stub script. (A stub is provided for Scala and Python; use whichever language you prefer.) Note that for convenience you have been provided with functions to parse the XML, as that is not the focus of this exercise. Copy the stub code into the Spark Shell.
4. 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.
5. 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.
6. 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.
7. Save the formatted strings to a text file in the directory `/loudacre/account-models`.

Bonus Exercise

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

Review the contents of the data file `devicestatus.txt`. This file contains data collected from mobile devices on Loudacre's network, including device ID, current status, location and so on. Because Loudacre previously acquired other mobile provider's networks, the data from different subnetworks has a different format. Note that the records in this file

have different field delimiters: some use commas, some use pipes (|) and so on. Your tasks are to:

- Load the dataset
- Determine which delimiter to use (hint: the character at position 19 is the first use of the delimiter)
- Filter out any records which do not parse correctly (hint: each record should have exactly 14 values)
- Extract the date (first field), model (second field), device ID (third field), and latitude and longitude (13th and 14th fields respectively)
- The second field contains the device manufacturer and model name (e.g. `Ronin S2`.) Split this field by spaces to separate the manufacturer from the model (e.g., manufacturer `Ronin`, model `S2`.)
- Save the extracted data to comma delimited text files in the `/loudacre/devicestatus_etl` directory on HDFS.
- Confirm that the data in the file(s) was saved correctly.

This is the end of the Exercise

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

Files Used in This Exercise:

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

Data files (local):

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

Solution: `solutions/UserRequests.pyspark`
 `solutions/UserRequests.scalaspark`

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

Exploring Web Log Files

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

Tip: In this exercise you will be reducing and joining large datasets, which can take a lot of time. You may wish to perform the exercises below using a smaller dataset, consisting of only a few of the web log files, rather than all of them. Remember that you can specify a wildcard; `textFile("/loudacre/weblogs/*6.log")` would include only filenames ending with the digit 6 and having a `log` file extension.

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

(<i>userid</i> , 1)
(<i>userid</i> , 1)
(<i>userid</i> , 1)
...

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

(<i>userid</i> , 5)
(<i>userid</i> , 7)
(<i>userid</i> , 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, <i>userid</i>)
(7, <i>userid</i>)
(2, <i>userid</i>)
...

- 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`.

(<i>userid</i> , 20.1.34.55)
(<i>userid</i> , 245.33.1.1)
(<i>userid</i> , 65.50.196.141)
...



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

Joining Web Log Data with Account Data

4. Copy the file `accounts.csv` data file to HDFS:

```
$ hdfs dfs -put \
~/training_materials/sparkdev/data/accounts.csv \
/loudacre/
```

This data set consists of information about Loudacre’s user accounts. The first field in each line is the user ID, which corresponds to the user ID in the web server logs. The other fields include account details such as creation date, first and last name and so on.

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

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

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

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

<code>(userid1, ([userid1, 2008-11-24 10:04:08, \N, Cheryl, West, 4905 Olive Street, San Francisco, CA, ...], 4))</code>
<code>(userid2, ([userid2, 2008-11-23 14:05:07, \N, Elizabeth, Kerns, 4703 Eva Pearl Street, Richmond, CA, ...], 8))</code>
<code>(userid3, ([userid3, 2008-11-02 17:12:12, 2013-07-18 16:42:36, Melissa, Roman, 3539 James Martin Circle, Oakland, CA, ...], 1))</code>
...

- c. Display the user ID, hit count, and first name (3rd value) and last name (4th value) for the first 5 elements, e.g.:

```
userid1 4 Cheryl West
userid2 8 Elizabeth Kerns
userid3 1 Melissa Roman
```

Bonus Exercises

If you have more time, attempt the following challenges:

- Challenge 1: Use `keyBy` to create an RDD of account data with the postal code (9th field in the CSV file) as the key.
 - Tip: Assign this new RDD to a variable for use in the next challenge
- Challenge 2: Create a pair RDD with postal code as the key and a list of names (Last Name, First Name) in that postal code as the value.
 - Hint: First name and last name are the 4th and 5th fields respectively
 - Optional: Try using the `mapValues` operation
- Challenge 3: Sort the data by postal code, then for the first five postal codes, display the code and list the names in that postal zone, such as:

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

This is the end of the Exercise

Hands-On Exercise: Write and Run a Spark Application

Files and Directories Used in This Exercise:

Data files (HDFS):	<code>/loudacre/weblogs/*</code>
Scala Project Directory:	<code>projects/countjpgs</code>
Scala Classes:	<code>stubs.CountJPGs</code> <code>solution.CountJPGs</code>
Python Stub:	<code>stubs/CountJPGs.py</code>
Python Solution:	<code>solutions/CountJPGs.py</code>

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

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

This is the same task you did earlier in the “Getting Started With RDDs” exercise. The logic is the same, but this time you will need to set up the `SparkContext` object yourself.

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

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

Writing a Spark Application in Python

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

1. A simple stub file to get started has been provided:
`~/training_materials/sparkdev/stubs/countjpgs.py`. This stub imports the required Spark class and sets up your main code block. Copy this stub to your work area and edit it to complete this exercise.

2. Set up a `SparkContext` using the following code:

```
sc = SparkContext()
```

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

4. At the end of the program, be sure to call:

```
sc.stop()
```

5. Run the program locally, passing the name of the log file to process, such as:

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

6. Skip the Scala exercises below, and skip to the section "Start the Spark Standalone Cluster."

Writing a Spark Application in Scala

You may use any text editor you wish. If you don't have an editor preference, you may wish to use gedit, which includes language-specific support for Scala. If you are familiar with the Idea IntelliJ IDE, you may choose to use that; the provided project directories include IntelliJ configuration.

A Maven project to get started has been provided in the `projects/countjpgs` directory.

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

8. Set up a `SparkContext` using the following code:

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

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

10. At the end of the program, be sure to call:

```
sc.stop
```

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

```
$ mvn package
```

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

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

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

Starting the Spark Standalone Cluster

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


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

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

14. View the Spark Standalone Cluster UI: Start Firefox on your VM and visit the Spark Master UI by using the provided bookmark or visiting `http://localhost:18080/`.
15. You should not see any applications in the Running Applications or Completed Applications areas because you have not run any applications on the cluster yet.
16. A real-world Spark cluster would have several workers configured. In this class we have just one, running locally, which is named by the date it started, the host it is running on, and the port it is listening on. For example:

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

17. Click on the worker ID link to view the Spark Worker UI and note that there are no executors currently running on the node.
18. In the previous section, you ran your application locally, because you did not specify a master when starting it. Re-run the program, specifying the cluster master in order to run it on the cluster.

For Python:

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

For Scala:

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

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

This is the end of the Exercise

Hands-On Exercise: Configure a Spark Application

Files Used in This Exercise:

Data files (HDFS): /loudacre/weblogs

Properties files (local): spark.conf
 log4j.properties

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

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

Setting Configuration Options at the Command Line

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

```
$ spark-submit --master spark://localhost:7077 \  
  --name 'Count JPGs' \  
  CountJPGs.py /loudacre/weblogs/*
```

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

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

3. *Optional:* While the application is running, visit the Spark Application UI and view the **Environment** tab. Take note of the `spark.*` properties such as `master`, `appName`, and `driver` properties.

Setting Configuration Options in a Configuration File

4. Change directories to your working directory. (If you are working in Scala, that is the `countjpgs` project directory.)
5. Using a text editor, create a file in the working directory called `myspark.conf`, containing settings for the properties shown below:

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

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

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

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

7. While the application is running, view the Standalone Spark Master UI to confirm application name is correctly displayed as “My Spark App,” such as:

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

Setting Logging Levels

- Copy the template file `/etc/spark/conf/log4j.properties.template` to `log4j.properties` in your working directory.
- Edit `log4j.properties`. The first line currently reads:

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

Replace `INFO` with `DEBUG`:

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

- Re-run your Spark application. Because the current directory is on the Java classpath, your `log4j.properties` file will set the logging level to `DEBUG`.
- Notice that the output now contains both the `INFO` messages it did before and `DEBUG` messages, similar to what is shown below:

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

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

- Edit the `log4j.properties` file to replace `DEBUG` with `WARN` and try again. This time notice that no `INFO` or `DEBUG` messages are displayed; only `WARN` messages.
- You can also set the log level for the Spark Shell by placing the `log4j.properties` file in your working directory before starting the shell. Try starting the shell from the directory in which you placed the file and note that only `WARN` messages now appear.

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

This is the end of the Exercise

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

Files and Data Used in This Exercise:

Data files (HDFS):	<code>/loudacre/weblogs/*</code> <code>/loudacre/accounts.csv</code>
Solutions:	<code>solutions/SparkStages.pyspark</code> <code>solutions/SparkStages.scalaspark</code>

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

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

Exploring Partitioning of File-Based RDDs

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

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

or for Scala:

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

2. Create an RDD based on the accounts datafile (`/loudacre/accounts.csv`) and then call `toDebugString` on the RDD, which displays the number of partitions in parentheses () before the RDD ID. How many partitions are in the resulting RDD?

```
pyspark> accounts=sc.textFile("/loudacre/accounts.csv")
pyspark> print accounts.toDebugString()
```

```
scala> val accounts=sc.
      textFile("/loudacre/accounts.csv")
scala> accounts.toDebugString
```

3. Repeat this process, but specify a minimum of three of partitions:
`textFile(filename, 3)`. Does the RDD correctly have three partitions?
4. Create another RDD based on all the weblogs dataset files (`/loudacre/weblogs/*`) and then call `toDebugString` on the RDD. How many partitions are in the weblogs RDD?

```
pyspark> weblogs=sc.textFile("/loudacre/weblogs/*")
pyspark> print weblogs.toDebugString()
```

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

How does the number of files in the dataset compare to the number of partitions in the RDD?

5. Repeat this process, but specify only a subset of the files: those for the month of October in 2013, `/loudacre/weblogs/2013-10-*.log`.
6. Bonus: use `foreachPartition` to print out the first record of each partition.

Setting up the Job

7. First, create an RDD of accounts, keyed by ID and with first name, last name for the value.

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

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

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

```
pyspark> userreqs = weblogs \
    .map(lambda line: line.split()) \
    .map(lambda words: (words[2],1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```

```
scala> val userreqs = weblogs.
    map(line => line.split(' ')).
    map(words => (words(2),1)).
    reduceByKey((v1,v2) => v1 + v2)
```

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

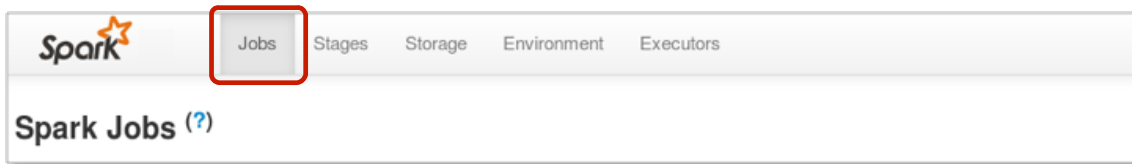
```
pyspark> accounthits = accountsByID.join(userreqs)\
    .values()
```

```
scala> val accounthits =  
  accountsByID.join(userreqs).values
```

10. Print the results of `accounthits.toDebugString` and review the output. Based on this, see if you can determine:
- How many stages are in this job?
 - Which stages are dependent on which?
 - How many tasks will each stage consist of?

Runing and Reviewing the Job in the Spark Application UI

11. In your browser, visiting the Spark Application UI by using the provided toolbar bookmark, or visiting `http://localhost:4040/` in your browser.
12. In the Spark UI, make sure the **Jobs** tab is selected. No jobs are yet running so the list will be empty.



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

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

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

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

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

Things to note:

- a. How many stages are in the job? Does it match the number you expected from the RDD's `toDebugString` output?
- b. The stages are numbered, but numbers do not relate to the order of execution. Note the times the stages were submitted to determine the order. Does the order match what you expected based on RDD dependency?
- c. How many tasks are in each stage? The number of tasks in the first stages corresponds to the number of partitions.
- d. The Shuffle Read and Shuffle Write columns indicate how much data was copied between tasks. This is useful to know because copying too much data across the network can cause performance issues.

16. Click on the stages to view details about that stage.

Things to note:

- a. The Summary Metrics area shows you how much time was spend on various steps. This can help you narrow down performance problems.
- b. The Tasks area lists each task. The Locality Level column indicates whether the process ran on the same node where the partition was physically stored or not. Remember that Spark will attempt to always run tasks where the data is, but may not always be able to, if the node is busy.
- c. In a real-world cluster, the executor column in the Task area would display the different worker nodes that ran the tasks. In this single-node cluster, all tasks run on the same host: `localhost`.

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

18. *Optional:* Try re-running the last action. You will need to either delete the `saveAsTextFile` output directory in HDFS, or specify a different directory name. You

will probably find that the job completes much faster, and that several stages (and the tasks in them) show as “skipped.”

Bonus question: Which tasks were skipped and why?

Leave the Spark Shell running for the next exercise.

This is the end of the Exercise

Hands-On Exercise: Persist an RDD

Files and Data Used in This Exercise:

Data files (HDFS):	/loudacre/weblogs/* /loudacre/accounts.csv
Job Setup:	solutions/SparkStages.pyspark solutions/SparkStages.scalaspark

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

1. Make sure the Spark Shell is still running from the last exercise. If it isn't, restart it (in local mode with two threads) and paste in the job setup code from the solution file or the previous exercise.
2. This time to start the job you are going to perform a slightly different action than last time: count the number of user accounts with a total hit count greater than five:

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

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

3. Cache (persist to memory) the RDD by calling `accounthits.persist()`.
4. In your browser, view the Spark Application UI and select the **Storage** tab. At this point, you have marked your RDD to be persisted, but have not yet performed an action that would cause it to be materialized and persisted, so you will not yet see any persisted RDDs.
5. In the Spark Shell, execute the count again.

6. View the RDD's `toDebugString`. Notice that the output indicates the persistence level selected.
7. Reload the Storage tab in your browser, and this time note that the RDD you persisted is shown. Click on the RDD ID to see details about partitions and persistence.
8. Click on the **Executors** tab and take note of the amount of memory used and available for your one worker node.

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

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

This is the end of the Exercise

Hands-On Exercise: Implement an Iterative Algorithm

Files and Data Used in This Exercise:

Data files (HDFS):	<code>/loudacre/devicestatus_etl/</code>
Stubs:	<code>stubs/KMeansCoords.pyspark</code> <code>stubs/KMeansCoords.scalaspark</code>
Solutions:	<code>solutions/KMeansCoords.pyspark</code> <code>solutions/KMeansCoords.scalaspark</code>

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

Reviewing the Data

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

If you did not have time to complete that bonus exercise, run the solution script now following the two steps below.

- Copy `~/training_materials/sparkdev/data/devicestatus.txt` to the `/loudacre/` directory in HDFS.
- Run the Spark script
`~/training_materials/solutions/DeviceStatusETL`
(either `.pyspark` or `.scalaspark` depending on which language you are using)

Examine the data in the dataset. Note that the latitude and longitude are the 4th and 5th fields, respectively, such as:

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

Calculate k-means for Device Location

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

1. Start by copying the provided `KMeansCoords` stub file, which contains the following convenience functions used in calculating k-means:
 - `closestPoint`: given a (latitude/longitude) point and an array of current center points, returns the index in the array of the center closest to the given point
 - `addPoints`: given two points, return a point which is the sum of the two points – that is, $(x1+x2, y1+y2)$
 - `distanceSquared`: given two points, returns the squared distance of the two. This is a common calculation required in graph analysis.
2. Set the variable `K` (the number of means to calculate). For this use `K=5`.
3. Set the variable `convergeDist`. This will be used to decide when the k-means calculation is done – when the amount the locations of the means changes between iterations is less than `convergeDist`. A “perfect” solution would be 0; this number represents a “good enough” solution. For this exercise, use a value of `0.1`.
4. Parse the input file, which is delimited by a comma character within (latitude, longitude) pairs (the 4th and 5th fields in each line). Only include known locations (that is, filter out (0,0) locations). Be sure to persist (cache) the resulting RDD because you will access it each time through the iteration.

5. Create a k-length array called `kPoints` by taking a random sample of `k` location points from the RDD as starting means (center points); for example:

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

6. Iteratively calculate a new set of `K` means until the total distance between the means calculated for this iteration and the last is smaller than `convergeDist`. For each iteration:
- For each coordinate point, use the provided `closestPoint` function to map each point to the index in the `kPoints` array of the location closest to that point. The resulting RDD should be keyed by the index, and the value should be the pair: `(point, 1)`. The value "1" will later be used to count the number of points closest to a given mean; for example:

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

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

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

- The reduced RDD should have (at most) `k` members. Map each to a new center point by calculating the average latitude and longitude for each set of closest

points: that is, `map (index, (totalX, totalY), n) to (index, (totalX/n, totalY/n))`

- d. Collect these new points into a local map or array keyed by index.
 - e. Use the provided `distanceSquared` method to calculate how much each center “moved” between the current iteration and the last. That is, for each center in `kPoints`, calculate the distance between that point and the corresponding new point, and sum those distances. That is the delta between iterations; when the delta is less than `convergeDist`, stop iterating.
 - f. Copy the new center points to the `kPoints` array in preparation for the next iteration.
7. When the iteration is complete, display the final `k` center points.

This is the end of the Exercise

Hands-On Exercise: Use Broadcast Variables

Files Used in This Exercise:

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

Data files (local):

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

Stubs: `stubs/TargetModels.pyspark`
 `stubs/TargetModels.scalaspark`

Solutions: `solutions/TargetModels.pyspark`
 `solutions/TargetModels.scalaspark`

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

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

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

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

This is the end of the Exercise

Hands-On Exercise: Use Accumulators

Files Used in This Exercise:

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

Solutions: `solutions/RequestAccumulator.pyspark`
 `solutions/RequestAccumulator.scalaspark`

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

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

Hint: use the file extension string to determine the type of request, such as `.html`, `.css`, or `.jpg`.

This is the end of the Exercise

Hands-On Exercise: Use Spark SQL for ETL

Files and Data Used in this Exercise

MySQL table: `loudacre.webpage`

Output HDFS directory: `/loudacre/webpage_files`

Solutions: `solutions/SparkSQL-webpage-files.pyspark`
`solutions/SparkSQL-webpage-files.scalaspark`

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

Reviewing the Data in MySQL

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

1. List the columns and types in the table:

```
$ mysql -utrainig -ptraining loudacre \  
-e"DESCRIBE webpage"
```

2. View the first few rows from the table:

```
$ mysql -utrainig -ptraining loudacre \  
-e"SELECT * FROM webpage LIMIT 5"
```

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

Loading the Data from MySQL

3. If necessary, start the Spark Shell.
4. Import the `SQLContext` class definition, and define a SQL context:

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

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

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

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

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

6. Examine the schema of the new `DataFrame` by calling `webpages.printSchema()`.
7. Create a new `DataFrame` by selecting the `web_page_num` and `associated_files` columns from the existing `DataFrame`:

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

```
scala> val assocfiles =  
webpages.select(webpages("web_page_num"),webpages("associated  
_files"))
```

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

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

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

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

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

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

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

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

10. Create a new `DataFrame` from the `RDD`:

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

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

11. Call `printSchema` on the new `DataFrame`. Note that Spark SQL gave the columns generic names: `_1` and `_2`.
12. Create a new `DataFrame` by renaming the columns to reflect the data they hold.

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

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

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

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

13. Call `printSchema` to confirm that the new `DataFrame` has the correct column names.
14. Your final `DataFrame` contains the processed data, so call `finaldf.collect()` to confirm the data is correct.
15. Optional: Save the final `DataFrame` in Parquet format (the default) in `/loudacre/webpage_files`. The code is the same in Scala and Python.

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

Confirm that the data was saved in HDFS is correct. Note that the data will be in Parquet file format, which is a binary format. This means that when you view the file, only some of the content will be in readable string form. This is expected behavior.

This is the end of the Exercise

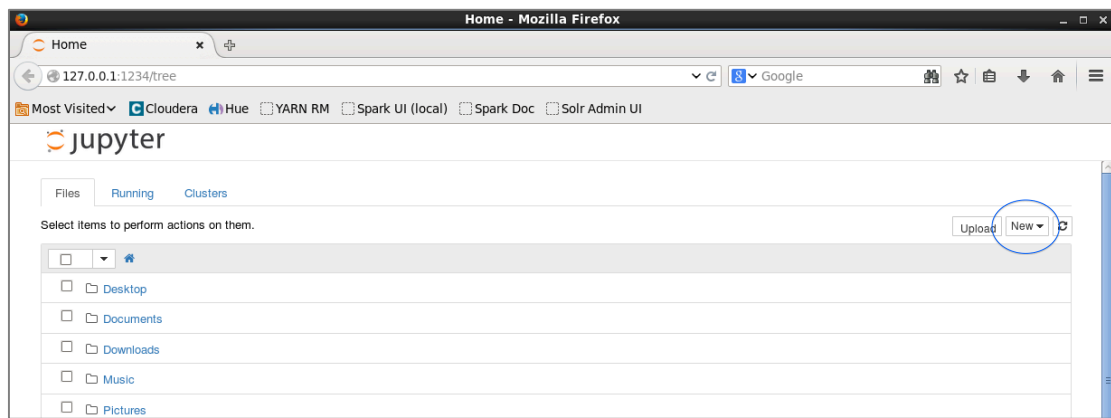
Appendix A: Enabling iPython Notebook

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

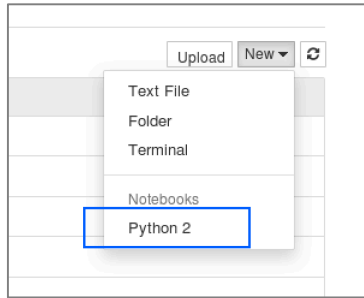
1. Open the following file for editing: `/home/training/.bashrc`
2. Uncomment out the following line (remove the leading #).

```
# export PYSARK_DRIVER_PYTHON_OPTS='notebook .....jax'
```

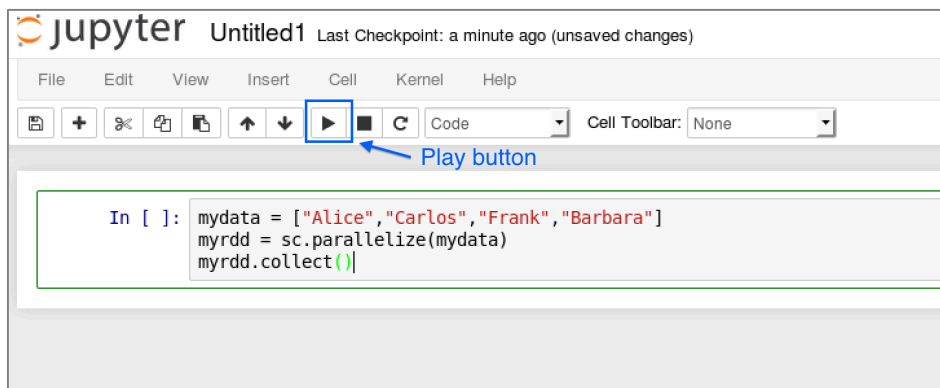
3. Save the file.
4. Open a new terminal window. (Must be a new terminal so it loads your edited `.bashrc` file).
5. Enter `pyspark` in the terminal. This will cause a browser window to open, and you should see the following web page:



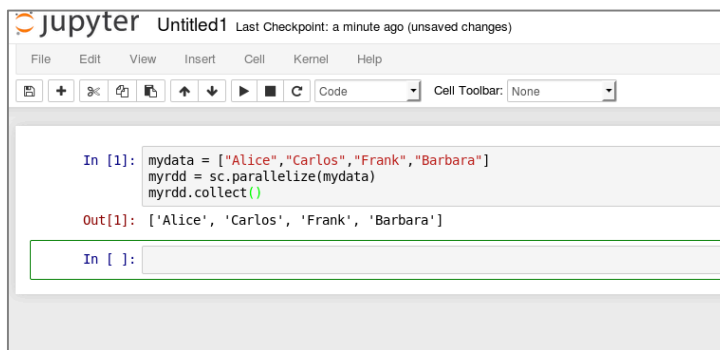
6. On the right hand side of the page select **Python 2** from the **New** menu



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



8. Notice the output displayed.



This is the end of the Appendix

Data Model Reference

Note that not all of the information below applies to this customized version of the course.

Tables Imported from MySQL

The following depicts the structure of the MySQL tables imported into HDFS using Sqoop. The primary key column from the database, if any, is denoted by bold text:

customers: 201,375 records (imported to /dualcore/customers)

Index	Field	Description	Example
0	cust_id	Customer ID	1846532
1	fname	First name	Sam
2	lname	Last name	Jones
3	address	Address of residence	456 Clue Road
4	city	City	Silicon Sands
5	state	State	CA
6	zipcode	Postal code	94306

employees: 61,712 records (imported to /dualcore/employees and later used as an external table in Hive)

Index	Field	Description	Example
0	emp_id	Employee ID	BR5331404
1	fname	First name	Betty
2	lname	Last name	Richardson
3	address	Address of residence	123 Shady Lane
4	city	City	Anytown
5	state	State	CA
6	zipcode	Postal Code	90210
7	job_title	Employee's job title	Vice President
8	email	e-mail address	br5331404@example.com
9	active	Is actively employed?	Y
10	salary	Annual pay (in dollars)	136900

orders: 1,662,951 records (imported to /dualcore/orders)

Index	Field	Description	Example
0	order_id	Order ID	3213254

1	cust_id	Customer ID	1846532
2	order_date	Date/time of order	2013-05-31 16:59:34

order_details: 3,333,244 records (imported to /dualcore/order_details)

Index	Field	Description	Example
0	order_id	Order ID	3213254
1	prod_id	Product ID	1754836

products: 1,114 records (imported to /dualcore/products)

Index	Field	Description	Example
0	prod_id	Product ID	1273641
1	brand	Brand name	Foocorp
2	name	Name of product	4-port USB Hub
3	price	Retail sales price, in cents	1999
4	cost	Wholesale cost, in cents	1463
5	shipping_wt	Shipping weight (in pounds)	1

suppliers: 66 records (imported to /dualcore/suppliers)

Index	Field	Description	Example
0	supp_id	Supplier ID	1000
1	fname	First name	ACME Inc.
2	lname	Last name	Sally Jones
3	address	Address of office	123 Oak Street
4	city	City	New Athens
5	state	State	IL
6	zipcode	Postal code	62264
7	phone	Office phone number	(618) 555-5914

Hive/Impala Tables

The following is a record count for tables that are created or queried during the hands-on exercises. Use the `DESCRIBE tablename` command to see the table structure.

Table Name	Record Count
ads	788,952
cart_items	33,812
cart_orders	12,955
cart_shipping	12,955
cart_zipcodes	12,955
checkout_sessions	12,955
customers	201,375
employees	61,712
loyalty_program	311
order_details	3,333,244
orders	1,662,951
products	1,114
ratings	21,997
web_logs	412,860

Other Data Added to HDFS

The following describes the structure of other important data sets added to HDFS.

Combined Ad Campaign Data: (788,952 records total), stored in two directories:

- /dualcore/ad_data1 (438,389 records)
- /dualcore/ad_data2 (350,563 records).

Index	Field	Description	Example
0	campaign_id	Uniquely identifies our ad	A3
1	date	Date of ad display	05/23/2013
2	time	Time of ad display	15:39:26
3	keyword	Keyword that triggered ad	tablet
4	display_site	Domain where ad shown	news.example.com
5	placement	Location of ad on Web page	INLINE
6	was_clicked	Whether ad was clicked	1
7	cpc	Cost per click, in cents	106

access.log: 412,860 records (uploaded to /dualcore/access.log)

This file is used to populate the `web_logs` table in Hive. Note that the RFC 931 and Username fields are seldom populated in log files for modern public Web sites and are ignored in our RegexSerDe.

Index	Field / Description	Example
0	IP address	192.168.1.15
1	RFC 931 (Ident)	-
2	Username	-
3	Date/Time	[22/May/2013:15:01:46 -0800]
4	Request	"GET /foo?bar=1 HTTP/1.1"
5	Status code	200
6	Bytes transferred	762
7	Referer	"http://dualcore.com/"
8	User agent (browser)	"Mozilla/4.0 [en] (WinNT; I) "
9	Cookie (session ID)	"SESSION=8763723145"

Regular Expression Reference

The following is a brief tutorial intended for the convenience of students who don't have experience using regular expressions or may need a refresher. A more complete reference can be found in the documentation for Java's Pattern class:

<http://tiny.cloudera.com/regexpattern>

Introduction to Regular Expressions

Regular expressions are used for pattern matching. There are two kinds of patterns in regular expressions: literals and metacharacters. Literal values are used to match precise patterns while metacharacters have special meaning; for example, a dot will match any single character. Here's the complete list of metacharacters, followed by explanations of those that are commonly used:

< ([{ \ ^ - = \$! |] }) ? * + . >

Literal characters are any characters not listed as a metacharacter. They're matched exactly, but if you want to match a metacharacter, you must escape it with a backslash. Since a backslash is itself a metacharacter, it must also be escaped with a backslash. For example, you would use the pattern `\\.` to match a literal dot.

Regular expressions support patterns much more flexible than simply using a dot to match any character. The following explains how to use *character classes* to restrict which characters are matched.

Character Classes

[057]	Matches any single digit that is either 0, 5, or 7
[0-9]	Matches any single digit between 0 and 9
[3-6]	Matches any single digit between 3 and 6
[a-z]	Matches any single lowercase letter
[C-F]	Matches any single uppercase letter between C and F

For example, the pattern `[C-F][3-6]` would match the string D3 or F5 but would fail to match G3 or C7.

There are also some built-in character classes that are shortcuts for common sets of characters.

Predefined Character Classes

<code>\\d</code>	Matches any single digit
<code>\\w</code>	Matches any word character (letters of any case, plus digits or underscore)
<code>\\s</code>	Matches any whitespace character (space, tab, newline, etc.)

For example, the pattern `\\d\\d\\d\\w` would match the string `314d` or `934X` but would fail to match `93X` or `Z871`.

Sometimes it's easier to choose what you don't want to match instead of what you do want to match. These three can be negated by using an uppercase letter instead.

Negated Predefined Character Classes

<code>\\D</code>	Matches any single non-digit character
<code>\\W</code>	Matches any non-word character
<code>\\S</code>	Matches any non-whitespace character

For example, the pattern `\\D\\D\\W` would match the string `ZX#` or `@ P` but would fail to match `93X` or `36_`.

The metacharacters shown above match each exactly one character. You can specify them multiple times to match more than one character, but regular expressions support the use of quantifiers to eliminate this repetition.

Matching Quantifiers

<code>{5}</code>	Preceding character may occur exactly five times
<code>{0,6}</code>	Preceding character may occur between zero and six times
<code>?</code>	Preceding character is optional (may occur zero or one times)
<code>+</code>	Preceding character may occur one or more times
<code>*</code>	Preceding character may occur zero or more times

By default, quantifiers try to match as many characters as possible. If you used the pattern `ore.+a` on the string `Dualcore has a store in Florida`, you might be surprised to learn that it matches `ore has a store in Florida` rather than `ore ha` or `ore in Florida` as you might have expected. This is because matches a "greedy" by default. Adding a question mark makes the quantifier match as few characters as possible instead, so the pattern `ore.+?a` on this string would match `ore ha`.

Finally, there are two special metacharacters that match zero characters. They are used to ensure that a string matches a pattern only when it occurs at the beginning or end of a string.

Boundary Matching Metacharacters

<code>^</code>	Matches only at the beginning of a string
<code>\$</code>	Matches only at the ending of a string

NOTE: When used inside square brackets (which denote a character class), the `^` character is interpreted differently. In that context, it negates the match. Therefore, specifying the pattern `[^0-9]` is equivalent to using the predefined character class `\d` described earlier.