201612

cloudera UNIVERSITY

Cloudera Data Analyst Training: Hands-On Exercises

General Notes

Cloudera provides a *hands-on environment* to accompany this training course. This consists of a virtual machine (VM) running Linux, with a recent version of CDH already installed and configured for you. This VM runs in pseudo-distributed mode, a configuration that enables a Hadoop cluster to run on a single machine.

Restarting Required Services

Your hands-on environment is configured to automatically start all required services. Under certain circumstances, services may fail and need to be restarted. If you encounter messages such as `Connection refused`, `Not connected`, or `Could not connect`, or if jobs or queries hang indefinitely and never complete, you can correct the problem by running the restart services script:

```
$ $ADIR/scripts/restart_services.sh
```

This script shuts down then restarts all required services.

Points to Note while Working in the Hands-On Environment

1. The hands-on environment 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_2016 \
/user/training/tax_analysis/
```

   The dollar sign (**$**) 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.

HDFS Warnings

Due to a bug in HDFS, you may see `java.lang.InterruptedException` warnings when running `hdfs dfs -put` commands or other commands that copy data to HDFS. These warnings are of no consequence and can be ignored.

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 &
```

Points to Note during the Exercises

Sample Solutions

If you need a hint or want to check your work, the `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 you would like an additional challenge to practice what you have learned.

Hands-On Exercise: Data Ingest with Hadoop Tools

Exercise directory:	`$ADIR/exercises/data_ingest`

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.

Exploring HDFS using the Hue File Browser

1. Launch the hands-on environment if you haven't already done so, and start the Firefox web browser in the hands-on environment by clicking on the icon in the system toolbar:

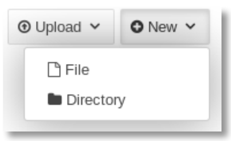
2. In Firefox, click on the **Hue** bookmark in the bookmark toolbar (or type `http://localhost:8888/home` into the address bar and press the `Enter` key.)

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 **Create account** button.

4. Click the **File Browser** icon ( ) on the right side of the Hue top toolbar (maximize the browser window if you do not see it in the toolbar). The File Browser displays your HDFS home directory (since your user ID on the cluster is `training`, your home directory in HDFS is `/user/training`). This directory does not yet contain any files or directories.

> ### In Case of Error
>
> If Hue displays the error message `Cannot access: /user/training/`, this indicates that one or more of the required services has failed and needs to be restarted. Refer to the above section “Restarting Required Services” for instructions about how to restart the required services.

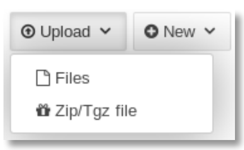
5. Create a temporary sub-directory: select the **+New** menu and click **Directory**.



6. Enter directory name `test` and click the **Create** button. Your home directory now contains a directory called `test`.

7. Click on **`test`** to view the contents of that directory; currently it contains no files or subdirectories.

8. Upload a file to the directory. Start by selecting **Upload → Files**.



9. 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`.

10. Choose any of the data files in that directory and click the **Open** button.

> **In Case of Error**
>
> If Hue displays the error message `Error: IOException: Failed to find datanode`, this indicates that one or more of the required services has failed and needs to be restarted. Refer to the above section "Restarting Required Services" for instructions about how to restart the required services.

11. The file you selected will be loaded into the current HDFS directory. Click the filename to see the file's contents. Because HDFS is designed 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.

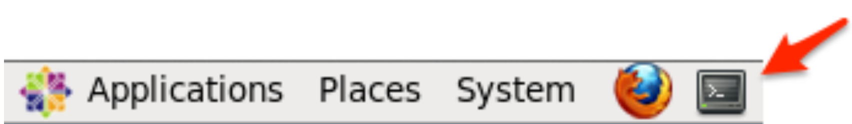
12. Return to the `test` directory by clicking **View file location** in the left hand panel.

13. 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.

14. 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**.

Exploring HDFS Using the Command Line

15. You can use the `hdfs dfs` command to interact with HDFS from the command line. Close or minimize Firefox, and 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
```

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.

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

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

24. 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;
```

25. 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;
```

26. Exit MySQL by typing `quit`, and press 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.

27. Run the following command, which imports the `employees` table into the `/dualcore` directory in HDFS, 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
```

It is normal for Sqoop to produce a lot of log messages, which are shown in your terminal window, during the import process.

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 `employees.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.

28. Revise the previous command and import the `customers` table into HDFS.

29. Revise the previous command and import the `products` table into HDFS.

30. Revise the previous command and import the `orders` table into HDFS.

31. 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 it 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: Using Pig for ETL Processing

Exercise directory:	`$ADIR/exercises/pig_etl`

In this exercise, you will practice using Pig to explore, correct, and reorder data in files from two different ad networks. You will first experiment with small samples of this data using Pig in local mode, and once you are confident that your ETL scripts work as you expect, you will use them to process the complete data sets in HDFS by using Pig in MapReduce mode.

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

```
$ $ADIR/scripts/catchup.sh
```

Background Information

Dualcore has recently started using online advertisements to attract new customers to our e-commerce site. Each of the two ad networks we use provides data about the ads that have been placed. This includes the site where the ad was placed, the date when it was placed, what keywords triggered its display, whether the user clicked the ad, and the per-click cost.

Unfortunately, the data from each network is in a different format. Each file also contains some invalid records. Before we can analyze the data, we must first correct these problems by using Pig to do the following:

- Filter invalid records
- Reorder fields
- Correct inconsistencies

- Write the corrected data to HDFS

Working in the Grunt Shell

In this section of the exercise, you will practice running Pig commands in the Grunt shell on a small sample of the data.

1. Change to the directory for this exercise:

```
$ cd $ADIR/exercises/pig_etl
```

2. Copy a small number of records from the input file to another file on the local file system. When you start Pig, you will run in local mode. For testing, you can work faster with small local files than large files in HDFS.

It is not essential to choose a random sample here—just a handful of records in the correct format will suffice. Use the command below to capture the first 25 records so you have enough to test your script:

```
$ head -n 25 $ADIR/data/ad_data1.txt > sample1.txt
```

3. Start the Grunt shell in local mode so that you can work with the local `sample1.txt` file.

```
$ pig -x local
```

A prompt indicates that you are now in the Grunt shell:

```
grunt>
```

4. Load the data in the `sample1.txt` file into Pig and dump it:

```
grunt> data = LOAD 'sample1.txt';
grunt> DUMP data;
```

You should see the 25 records that comprise the sample data file.

5. Load the first two columns' data from the sample file as character data, and then dump that data:

```
grunt> first_2_columns = LOAD 'sample1.txt' AS
       (keyword:chararray, campaign_id:chararray);
grunt> DUMP first_2_columns;
```

6. Use the DESCRIBE command to review the schema of first_2_columns:

```
grunt> DESCRIBE first_2_columns;
```

The schema appears in the Grunt shell.

Use the DESCRIBE command while performing these exercises any time you would like to review schema definitions.

7. See what happens if you run the DESCRIBE command on data. Recall that when you loaded data, you did *not* define a schema.

```
grunt> DESCRIBE data;
```

8. End your Grunt shell session:

```
grunt> QUIT;
```

Processing Input Data from the First Ad Network

In this section of the exercise, you will process the input data from the first ad network using a Pig script in a file. Many people find working with a Pig script in a file easier than working directly in the Grunt shell.

9. Edit the first_etl.pig file to complete the LOAD statement and read the data from the sample1.txt file you created earlier. The following table shows the format of the data in the file. For simplicity, you should leave the date and time fields separate, so each will be of type chararray, rather than converting them to a single field of type datetime.

Index	Field	Data Type	Description	Example
0	keyword	chararray	Keyword that triggered ad	tablet
1	campaign_id	chararray	Uniquely identifies our ad	A3
2	date	chararray	Date of ad display	05/29/2013
3	time	chararray	Time of ad display	15:49:21
4	display_site	chararray	Domain where ad shown	www.example.com
5	was_clicked	int	Whether ad was clicked	1
6	cpc	int	Cost per click, in cents	106
7	country	chararray	Name of country in which ad ran	USA
8	placement	chararray	Where on page was ad displayed	TOP

10. Once you have edited the `LOAD` statement, try it by running your script in local mode:

```
$ pig -x local first_etl.pig
```

Ensure that the output shows all fields in the expected order and the values appear similar in format to that shown in the table above before proceeding to the next step. You may find it helpful to also use a `DESCRIBE` statement at the end of the script to display the structure of the data.

11. Make each of the following changes, running your script in local mode after each one to verify that your change is correct:

a. Filter out all records except those where the country field equals `USA`.

b. Use a `FOREACH ... GENERATE` statement to create a new relation containing the fields in the order shown below (the `country` field is not included since all records now have the same value):

Index	Field	Description
0	campaign_id	Uniquely identifies our ad
1	date	Date of ad display
2	time	Time of ad display
3	keyword	Keyword that triggered the ad
4	display_site	Domain where ad shown

5	placement	Where on page was ad displayed
6	was_clicked	Whether ad was clicked
7	cpc	Cost per click, in cents

c. Update your script to convert the `keyword` field to uppercase and to remove any leading or trailing whitespace. (Hint: You can nest calls to the two built-in functions inside the `FOREACH ... GENERATE` statement from the last statement.)

12. Add the complete data file to HDFS:

```
$ hdfs dfs -put $ADIR/data/ad_data1.txt /dualcore/
```

13. Edit `first_etl.pig` and change the path in the `LOAD` statement to match the path of the file you just added to HDFS (`/dualcore/ad_data1.txt`).

14. Next, replace `DUMP` with a `STORE` statement that will write the output of your processing as tab-delimited records to the `/dualcore/ad_data1` directory.

15. Run this script in Pig's cluster mode to analyze the entire file in HDFS:

```
$ pig first_etl.pig
```

If your script fails, check your code carefully, fix the error, and then try running it again. Don't forget that you must remove output in HDFS from a previous run before you execute the script again.

16. Check the first 20 output records that your script wrote to HDFS and ensure they look correct:

```
$ hdfs dfs -cat /dualcore/ad_data1/part* | head -20
```

You can ignore the message that says `cat` is unable to write to the output stream; this simply happens because you are writing more data with the `hdfs dfs -cat` command than you are reading with the `head` command.

a. Are the fields in the correct order?

b. Are all the keywords now in uppercase?

Processing Input Data from the Second Ad Network

Now that you have successfully processed the data from the first ad network, continue by processing data from the second one.

17. Create a small sample of the data from the second ad network that you can test locally while you develop your script:

```
$ head -n 25 $ADIR/data/ad_data2.txt > sample2.txt
```

18. Edit the `second_etl.pig` file to complete the `LOAD` statement and read the data from the sample you just created. (Hint: The fields are comma-delimited.) The following table shows the order of fields in this file, again with each *row* of the table representing the contexts of one *column* of the data:

Index	Field	Data Type	Description	Example
0	campaign_id	chararray	Uniquely identifies our ad	A3
1	date	chararray	Date of ad display	05/29/2013
2	time	chararray	Time of ad display	15:49:21
3	display_site	chararray	Domain where ad shown	www.example.com
4	placement	chararray	Ad display location on page	TOP
5	was_clicked	int	Whether ad was clicked	Y
6	cpc	int	Cost per click, in cents	106
7	keyword	chararray	Keyword that triggered ad	tablet

19. Once you have edited the `LOAD` statement, use the `DESCRIBE` keyword and then run your script in local mode to check that the schema matches the table above:

```
$ pig -x local second_etl.pig
```

20. Replace `DESCRIBE` with a `DUMP` statement and then make each of the following changes to `second_etl.pig`, running this script in local mode after each change to verify what you've done before you continue with the next step:

a. This ad network sometimes logs a given record twice. Add a statement to the `second_etl.pig` file so that you remove any duplicate records. If you have done this correctly, you should only see one record where the `display_site` field has a value of `siliconwire.example.com`.

b. As before, you need to store the fields in a different order than you received them. Use a `FOREACH ... GENERATE` statement to create a new relation containing the fields in the same order you used to write the output from first ad network (shown again in the table below). Also, convert the `keyword` field to uppercase and remove any leading or trailing whitespace, as you did with the data from the first ad network:

Index	Field	Description
0	`campaign_id`	Uniquely identifies our ad
1	`date`	Date of ad display
2	`time`	Time of ad display
3	`keyword`	Keyword that triggered ad
4	`display_site`	Domain where ad shown
5	`placement`	Where on page was ad displayed
6	`was_clicked`	Whether ad was clicked
7	`cpc`	Cost per click, in cents

c. The date field in this dataset is in the format `MM-DD-YYYY`, while the data you previously wrote is in the format `MM/DD/YYYY`. Edit the `FOREACH ... GENERATE` statement to call the `REPLACE(date, '-', '/')` function to correct this.

21. Once you are sure the script works locally, add the full dataset to HDFS:

```
$ hdfs dfs -put $ADIR/data/ad_data2.txt /dualcore/
```

22. Edit the script to have it `LOAD` the file you just added to HDFS, and then replace the `DUMP` statement with a `STORE` statement to write your output as tab-delimited records to the `/dualcore/ad_data2` directory.

23. Run your script against the data you added to HDFS:

```
$ pig second_etl.pig
```

24. Check the first 15 output records written in HDFS by your script:

```
$ hdfs dfs -cat /dualcore/ad_data2/part* | head -15
```

a. Do you see any duplicate records?

b. Are the fields in the correct order?

c. Are all the keywords in uppercase?

d. Is the date field in the correct (`MM/DD/YYYY`) format?

This is the end of the exercise

Hands-On Exercise: Analyzing Ad Campaign Data with Pig

Exercise directory:	`$ADIR/exercises/analyze_ads`

During the previous exercise, you performed ETL processing on datasets from two online ad networks. In this exercise, you will write Pig scripts that analyze this data to optimize our advertising, helping Dualcore to save money and attract new customers.

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

```
$ $ADIR/scripts/catchup.sh
```

Finding Low-Cost Websites

Both ad networks charge us only when a user clicks on our ad. This is ideal for Dualcore since our goal is to bring new customers to our site. However, some sites and keywords are more effective than others at attracting people interested in the new tablet we advertise. With this in mind, you will begin by identifying which sites have the lowest total cost.

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

```
$ cd $ADIR/exercises/analyze_ads
```

2. Obtain a local subset of the input data by running the following command:

```
$ hdfs dfs -cat /dualcore/ad_data1/part* \
| head -n 100 > test_ad_data.txt
```

You can ignore the message that says `cat` is unable to write to the output stream; this simply happens because you are writing more data with the `hdfs dfs -cat` command than you are reading with the `head` command:

Note: As mentioned in the previous exercise, it is faster to test Pig scripts by using a local subset of the input data. You can use local subsets of data when testing Pig scripts throughout this course. Although explicit steps are not provided for creating local data subsets in upcoming exercises, doing so will help you perform the exercises more quickly.

3. Open the `low_cost_sites.pig` file in your editor, and then make the following changes:

   a. Modify the `LOAD` statement to read the sample data in the `test_ad_data.txt` file.

   b. Add a line that creates a new relation to include only records where `was_clicked` has a value of `1`.

   c. Group this filtered relation by the `display_site` field.

   d. Create a new relation that includes two fields: the `display_site` and the total cost of all clicks on that site.

   e. Sort that new relation by cost (in ascending order).

   f. Display just the first three records to the screen.

4. Once you have made these changes, try running your script against the sample data:

```
$ pig -x local low_cost_sites.pig
```

5. In the `LOAD` statement, replace the `test_ad_data.txt` file with a file glob (pattern) that will load both the `/dualcore/ad_data1` and `/dualcore/ad_data2` directories (and does *not* load any other data, such as the text files from the previous exercise).

6. Once you have made these changes, try running your script against the data in HDFS:

```
$ pig low_cost_sites.pig
```

Question: Which three sites have the lowest overall cost?

Finding High-Cost Keywords

The terms users type when doing searches may prompt the site to display a Dualcore advertisement. Since online advertisers compete for the same set of keywords, some of them cost more than others. You will now write some Pig Latin to determine which keywords have been the most expensive for us overall.

7. Since this will be a slight variation on the code you have just written, copy that file as `high_cost_keywords.pig`:

```
$ cp low_cost_sites.pig high_cost_keywords.pig
```

8. Edit the `high_cost_keywords.pig` file and make the following three changes:

    a. Group by the `keyword` field instead of `display_site`.

    b. Sort in descending order of cost.

    c. Display the top five results to the screen instead of the top three as before.

9. Once you have made these changes, try running your script against the data in HDFS:

```
$ pig high_cost_keywords.pig
```

Question: Which five keywords have the highest overall cost?

Bonus Exercise 1: Counting Ad Clicks

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

One important statistic we haven't yet calculated is the total number of clicks our ads have received. Doing so will help our Marketing Director plan her next ad campaign budget.

1. Change to the `bonus_01` subdirectory of the current exercise:

```
$ cd bonus_01
```

2. Edit the `total_click_count.pig` file and implement the following:

   a. Group the records (filtered by `was_clicked == 1`) so that you can call the aggregate function in the next step.

   b. Invoke the `COUNT` function to calculate the total of clicked ads. (Hint: Because we shouldn't have any null records, you can use the `COUNT` function instead of `COUNT_STAR`, and the choice of field you supply to the function is arbitrary.)

   c. Display the result to the screen.

3. Once you have made these changes, try running your script against the data in HDFS:

```
$ pig total_click_count.pig
```

Question: How many clicks did we receive?

Bonus Exercise 2: Estimating the Maximum Cost of the Next Ad Campaign

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

When you reported the total number of clicks to our Marketing Director, she said that her goal is to get about three times that amount during the next campaign. Unfortunately, because the cost is based on the site and keyword, she doesn't know how much to budget for that campaign. She asked you to help by estimating the worst case (most expensive) cost based on 50,000 clicks. You will do this by finding the most expensive ad and then multiplying it by the number of clicks she wants to achieve in the next campaign.

1. Change to the `bonus_02` subdirectory of the current exercise:

```
$ cd ../bonus_02
```

2. Because this code will be similar to the code you wrote in the previous step, start by copying that file as `project_next_campaign_cost.pig`:

```
$ cp ../bonus_01/total_click_count.pig \
project_next_campaign_cost.pig
```

3. Edit the `project_next_campaign_cost.pig` file and make the following modifications:

   a. Since you are trying to determine the highest possible cost, you should not limit your calculation to the cost for ads actually clicked. Remove the `FILTER` statement so that you consider the possibility that any ad might be clicked.

   b. Change the aggregate function to the one that returns the maximum value in the `cpc` field. (Hint: Don't forget to change the name of the relation in the preceding `GROUP` statement, and the name of the relation the `cpc` field comes from, to account for the removal of the `FILTER` statement in the previous instruction step.)

   c. Modify your `FOREACH...GENERATE` statement to multiply the value returned by the aggregate function by the total number of clicks we expect to have in the next campaign.

   d. Display the resulting value to the screen.

4. Once you have made these changes, try running your script against the data in HDFS:

```
$ pig project_next_campaign_cost.pig
```

Question: What is the maximum you expect this campaign might cost? You can compare your solution to the one in the `bonus_02/solution/` subdirectory.

Bonus Exercise 3: Calculating the Click-Through Rate (CTR)

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

The calculations you did at the start of this exercise gave us a rough idea about the success of our ad campaign, but they didn't account for the fact that some sites display our ads more than others. This makes it difficult to determine how effective our ads were by simply counting the number of clicks on one site and comparing it to the number of clicks on another site. One metric that would allow us to better make such comparisons is the click-through rate (`http://tiny.cloudera.com/ade03a`), commonly abbreviated as CTR. This value is simply the percentage of ads shown that users actually clicked, and it can be calculated by dividing the number of clicks by the total number of ads shown.

1. Change to the `bonus_03` subdirectory of the current exercise:

```
$ cd ../bonus_03
```

2. Edit the `lowest_ctr_by_site.pig` file and implement the following:

    a. Within the nested `FOREACH`, filter the records to include only records for which the ad was clicked.

    b. Create a new relation on the line that follows the `FILTER` statement which counts the number of records within the current group.

    c. Add another line below that to calculate the click-through rate in a new field named `ctr`.

    d. After the nested `FOREACH`, sort the records in ascending order of click-through rate and display the first three to the screen.

3. Once you have made these changes, try running your script against the data in HDFS:

```
$ pig lowest_ctr_by_site.pig
```

Question: Which three sites have the lowest click-through rate?

If you still have time remaining, modify your script to display the three keywords with the highest click-through rate. You can compare your solution to the `highest_ctr_by_keyword.pig` file in the `solution` directory.

This is the end of the exercise

Hands-On Exercise: Analyzing Disparate Datasets with Pig

Exercise directory: `$ADIR/exercises/disparate_datasets`

In this exercise, you will practice combining, joining, and analyzing the product sales data previously exported from Dualcore's MySQL database so you can observe the effects that our recent advertising campaign has had on sales.

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:

```
$ $ADIR/scripts/catchup.sh
```

Showing Per-Month Sales Before and After the Ad Campaign

Before proceeding with more sophisticated analysis, first calculate the number of orders Dualcore received each month for the three months before our ad campaign began (February–April, 2013), as well as for the month during which our campaign ran (May, 2013).

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

```
$ cd $ADIR/exercises/disparate_datasets
```

2. Open the `count_orders_by_period.pig` file in your editor. We have provided the `LOAD` statement as well as a `FILTER` statement that uses a regular expression to match the records in the data range you'll analyze. Make the following additional changes:

  a. Following the `FILTER` statement, create a new relation with just one field: the order's year and month. (Hint: Use the `SUBSTRING` built-in function to extract the first part of the `order_dtm` field, which contains the month and year.)

b. Count the number of orders in each of the months you extracted in the previous step.

c. Display the count by month to the screen.

3. Once you have made these changes, try running your script against the data in HDFS:

```
$ pig count_orders_by_period.pig
```

Question: Does the data suggest that the advertising campaign we started in May led to a substantial increase in orders?

Counting Advertised Product Sales by Month

Our analysis from the previous section of the exercise suggests that sales increased dramatically the same month we began advertising. Next, you'll compare the sales of the specific product we advertised (product ID 1274348) during the same period to see whether the increase in sales was actually related to our campaign.

You will be joining two datasets during this portion of the exercise. Since this is the first join you have done with Pig during class, now is a good time to mention a tip that can have a profound effect on the performance of your script. Filtering out unwanted data from each relation *before* you join them, as we've done in our example, means that your script will need to process less data and will finish more quickly. We will discuss several more Pig performance tips later in class, but this one is worth learning now.

4. Edit the `count_tablet_orders_by_period.pig` file and implement the following:

a. Join the two relations on the `order_id` field they have in common.

b. Create a new relation from the joined data that contains a single field: the order's year and month, similar to what you did previously in the `count_orders_by_period.pig` file.

c. Group the records by month and then count the records in each group.

d. Display the results to your screen.

5. Once you have made these changes, try running your script against the data in HDFS:

```
$ pig count_tablet_orders_by_period.pig
```

Question: Does the data show an increase in sales of the advertised product corresponding to the month in which Dualcore's campaign was active?

Bonus Exercise 1: Calculating Average Order Size

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

It appears that our advertising campaign was successful in generating new orders for Dualcore. Since we sell this tablet at a slight loss to attract new customers, let's see if customers who buy this tablet also buy other things. You will write code to calculate the average number of items for all orders that contain the advertised tablet during the campaign period.

1. Change to the `bonus_01` subdirectory of the current exercise:

```
$ cd bonus_01
```

2. Edit the `average_order_size.pig` file to calculate the average as described above. While there are multiple ways to achieve this, we recommend you implement the following:

   a. Filter the orders by date (using a regular expression) to include only those placed during the campaign period (May 1, 2013 through May 31, 2013).

   b. Filter the orders to include only those that contain the advertised product (product ID 1274348). (Hint: Apply DISTINCT to remove duplicate records representing orders that contain two or more of the advertised product.)

c. Create a new relation containing the `order_id` and `product_id` fields for these orders.

d. Count the total number of products in each order.

e. Calculate the average number of products for all orders.

3. Once you have made these changes, try running your script against the data in HDFS:

```
$ pig average_order_size.pig
```

Question: Does the data show that the average order contained at least two items in addition to the tablet we advertised?

Bonus Exercise 2: Customer Segmentation

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

Dualcore is considering starting a loyalty rewards program. This will provide exclusive benefits to our best customers, which will help us to retain them. Another advantage is that it will also allow us to capture even more data about how they shop with us; for example, we can easily track their in-store purchases when these customers give us their rewards program number during checkout.

To be considered for the program, a customer must have made at least five purchases from Dualcore during 2012. These customers will be segmented into groups based on the total retail price of all purchases each made during that year:

- **Platinum:** Purchases totaled at least $10,000
- **Gold:** Purchases totaled at least $5,000 but less than $10,000
- **Silver:** Purchases totaled at least $2,500 but less than $5,000

Since we are considering the total sales price of orders in addition to the number of orders a customer has placed, not every customer with at least five orders during 2012 will qualify. In fact, only about one percent of our customers will be eligible for membership in one of these three groups.

During this exercise, you will write the code needed to filter the list of orders based on date, group them by customer ID, count the number of orders per customer, and then filter this to exclude any customer who did not have at least five orders. You will then join this information with the order details and products datasets in order to calculate the total sales of those orders for each customer, split them into the groups based on the criteria described above, and then write the data for each group (customer ID and total sales) into a separate directory in HDFS.

1. Change to the `bonus_02` subdirectory of the current exercise:

```
$ cd ../bonus_02
```

2. Edit the `loyalty_program.pig` file and implement the steps described above. The code to load the three datasets you will need is already provided for you.

3. After you have written the code, run it against the data in HDFS:

```
$ pig loyalty_program.pig
```

4. If your script completed successfully, use the `hdfs dfs -getmerge` command to create a local text file for each group so you can check your work (note that the name of the directory shown here may not be the same as the one you chose):

```
$ hdfs dfs -getmerge /dualcore/loyalty/platinum platinum.txt
$ hdfs dfs -getmerge /dualcore/loyalty/gold gold.txt
$ hdfs dfs -getmerge /dualcore/loyalty/silver silver.txt
```

5. Use the UNIX `head` and `tail` commands to check a few records and ensure that the total sales prices fall into the correct ranges:

```
$ head platinum.txt
$ tail gold.txt
$ head silver.txt
```

6. Finally, count the number of customers in each group:

```
$ wc -l platinum.txt
$ wc -l gold.txt
$ wc -l silver.txt
```

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 Beeline shells to execute simple queries. These exercises use the tables that have been populated with data you imported to HDFS using Sqoop in an earlier exercise.

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:

```
$ $ADIR/scripts/catchup.sh
```

Exploring the `customers` Table Using Hue

One way to run Hive and Impala 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 web browser.

1. 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 press the `Enter` key).

2. 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.

3. Select the **Query Editors** menu in the Hue toolbar. Note that there are query editors for both Hive and Impala (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.

4. Make sure the `default` database is selected in the database list on the left side of the page.

5. Below the selected database is a list of the tables in that database. Click the `customers` table to view the columns in the table. It may take a few seconds to display the columns.

6. Click the **View Statistics** icon ( ) next to the table name to view sample data from the table.

Running 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.

7. 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.

8. After entering the query, click the **Execute** button ( ) to the left of the text area.

   While the query is executing, the **Query History** tab displays the status of 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?

Running a Query from the Impala Shell

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

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

10. 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, for example:

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

11. 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 semicolon (`;`), unlike in the Hue query editor.

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

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

13. Execute a query that displays the three most expensive products. (Hint: Use `ORDER BY`.)

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

```
quit;
```

Running 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 joins, you've been provided with a script in the exercise directory.

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

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

16. Review the code for the query:

```
$ cat verify_tablet_order.sql
```

17. 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?

Running a Query Using Beeline

18. 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, such as:

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

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

20. Exit the Beeline shell:

```
!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:

```
$ $ADIR/scripts/catchup.sh
```

Reviewing 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. Click the link for the `customers` table in the main panel, or move your cursor over the `customers` table in the left hand panel and click the **Open** icon (→), to display the table overview and review the list of columns.

4. Click the **Sample** tab to view the first hundred rows of data.

Creating and Loading a 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. In the main panel, click the **default** database to view the table browser.

8. Click on the **Create a new table manually** icon (➕) in the upper right 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 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.

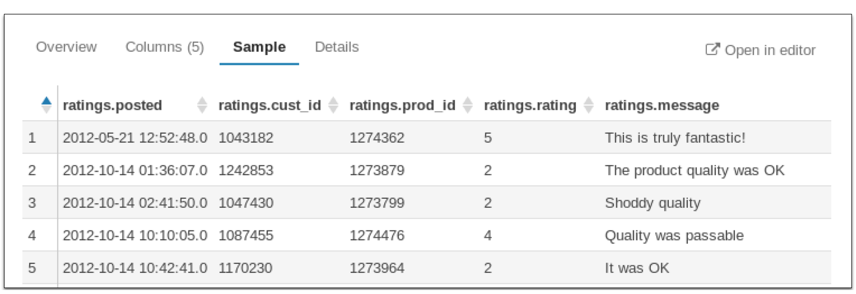
17. When the job is complete, the new table `ratings` will appear in the table browser.

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 the link for the new `ratings` table in the main panel, or move your cursor over the `ratings` table in the left hand panel and click the **Open** icon ( ). Then click the **Import Data** icon ( ) in the upper right.

20. In the Import data dialog, enter or browse to the HDFS location of the 2012 product ratings data file: `/dualcore/ratings_2012.txt`, and 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:



	ratings.posted	ratings.cust_id	ratings.prod_id	ratings.rating	ratings.message
1	2012-05-21 12:52:48.0	1043182	1274362	5	This is truly fantastic!
2	2012-10-14 01:36:07.0	1242853	1273879	2	The product quality was OK
3	2012-10-14 02:41:50.0	1047430	1273799	2	Shoddy quality
4	2012-10-14 10:10:05.0	1087455	1274476	4	Quality was passable
5	2012-10-14 10:42:41.0	1170230	1273964	2	It was OK

22. Try 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.

```
INVALIDATE METADATA;
```

Alternately, you could click the **Manually refresh the table list** icon ( ) in the panel on the left, then select **Invalidate all metadata and rebuild index** and click the **Refresh** button. This will cause Impala's metadata cache to reload.

24. If the table does not appear in the table list on the left, click the **Manually refresh the table list** icon ( ) in the panel on the left, then select **Clear cache** and click the **Refresh** button. This refreshes the panel, 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 and the Hue Import Data tool both *move* the file to the table's directory. Using the Hue File Browser or `hdfs dfs` commands, verify that the files are no longer present in the original directory:

```
$ hdfs dfs -ls /dualcore/ratings_2012.txt
$ 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;
```

Creating 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 format is shown below:

Field Name	Field Type
emp_id	STRING
fname	STRING
lname	STRING
address	STRING
city	STRING
state	STRING
zipcode	STRING
job_title	STRING
email	STRING
active	STRING
salary	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: Using 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.

1. 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
```

2. 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: Altering 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/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.

1. Use `ALTER TABLE` to rename the `company` column to `name`.

2. Use the `DESCRIBE` command on the `suppliers` table to verify the change.

3. Use `ALTER TABLE` to rename the entire table to `vendors`.

4. 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, for example:

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

This is the end of the exercise

Hands-On Exercise: Data Storage and Performance

Exercise directory: `$ADIR/exercises/data_storage`

In this exercise, you will create a table for ad click data that is partitioned by the network on which the ad was displayed.

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:

```
$ $ADIR/scripts/catchup.sh
```

Creating and Loading a Static Partitioned Table

In an earlier exercise, you used Pig to process ad click data from two different ad networks. Recall that the ad networks provided data in different formats, and your Pig scripts processed the data into the same format. If you did not complete the Pig exercise, be sure you have run the catch-up script.

The ad networks are not identified by a field in the data; instead the networks are distinguished by keeping the data from the different networks in separate files. Now, however, we would like to be able to query the ad data in the same table, but be able to distinguish between the two networks. We can do this by creating partitions for each network.

The processed data is in the following HDFS directories:

Ad Network 1: `/dualcore/ad_data1`
Ad Network 2: `/dualcore/ad_data2`

1. *Optional:* View the first few lines in the data files for both networks.

2. Define a table called ads with the following characteristics:

Type: EXTERNAL
Columns:

Field Name	Field Type
campaign_id	STRING
display_date	STRING
display_time	STRING
keyword	STRING
display_site	STRING
placement	STRING
was_clicked	TINYINT
cpc	INT

Partition column: network (type TINYINT)
Field Terminator: \t (tab)
Location: /dualcore/ads

3. Alter the ads table to add two partitions, one for network 1 and one for network 2.

4. Load the data in /dualcore/ad_data1 into the Network 1 partition.

5. Load the data in /dualcore/ad_data2 into the Network 2 partition.

6. Verify that the data for both ad networks were correctly loaded by counting the records for each:

```
SELECT COUNT(*) FROM ads WHERE network=1;
SELECT COUNT(*) FROM ads WHERE network=2;
```

7. Network 1 should have **438,389** records and Network 2 should have **350,563** records.

Bonus Exercise 1: Accessing Data in Parquet File Format

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

For this exercise, you have been provided data in Parquet format that is output from another tool—in this example, the output from a Pig ETL process that maps US zip codes to latitude and longitude. You need to be able to query this data in Impala.

1. The ETL data output directory is provided as `$ADIR/data/latlon`. Copy the data directory to the `/dualcore` directory in HDFS. You can use the Hue File Browser, or the following `hdfs` command:

```
$ hdfs dfs -put $ADIR/data/latlon /dualcore/
```

2. In Impala, create an external Parquet-based table pointing to the `latlon` data directory.

    - Hint: The actual data in the data directory is in a MapReduce output file called `part-m-00000.parquet`. Use the `LIKE PARQUET` command to use the existing column definitions in the data file.

3. Review the table in Impala to confirm it was correctly created with columns `zip`, `latitude` and `longitude`.

4. Perform a query or preview the data in the Impala Query Editor to confirm that the data in the data file is being accessed correctly.

This is the end of the exercise

Hands-On Exercise: Relational Data 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 a previous exercise.

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:

```
$ $ADIR/scripts/catchup.sh
```

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

Calculating the Top N Products

- Which top three products have been sold more than any other products? (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.)

Calculating the Order Total

- Which ten orders had the highest total dollar amounts?

Calculating Revenue and Profit

- Run 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 `orders` table is of type `TIMESTAMP`. Use the function `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 `solution` directory.

Bonus Exercise 1: Ranking 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.

- Run 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/solution` directory.

This is the end of the exercise

Hands-On Exercise: Complex Data with Hive and Impala

Exercise directory: `$ADIR/exercises/complex_data`

In this exercise, you will use Hive and Impala to work with complex data from a customer loyalty program.

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:

```
$ $ADIR/scripts/catchup.sh
```

Creating, Loading, and Querying Complex Data with Hive

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 (in cents). You will use Hive to create the table, populate it with the provided data, and then run a few queries to practice referencing these types of fields. Finally, you will create a Parquet table containing the same data, to allow you to use Impala to query the data.

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

1. Create a text file-based table with the following characteristics:

   Name: `loyalty_program`
   Type: `EXTERNAL`
   Columns:

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

Field Terminator: `|` (pipe)
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 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. Using either the Beeline shell or Hue's Hive Query Editor, run a query to select the `HOME` phone number for customer ID 1200866. (Hint: `MAP` keys are case-sensitive.) 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.

7. Create a Parquet-based table named `loyalty_program_parquet` with the same characteristics as the `loyalty_program` table, and use Hive to copy all the data from the `loyalty_program` table into this new `loyalty_program_parquet` table. (Hint: You can use `CREATE TABLE AS SELECT` (CTAS) to do this with a single HiveQL statement.)

Querying Complex Data in a Parquet Table with Impala

Now that you have used Hive to create, load, and query data with complex columns, continue by using Impala to query complex data in a Parquet-based table. Remember that the Impala SQL syntax for queries involving `ARRAY` and `MAP` types differs from the HiveQL syntax.

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

8. Invalidate Impala's metadata cache so that the `loyalty_program_parquet` table created and populated in the previous step can be accessed with Impala.

9. Run a query to return the distinct key values in the `phone` column. (Hint: You can query a `MAP` column in Impala as if it were a separate table with columns `key` and `value`.)

10. These distinct key values represent types of phone numbers. Now run a query to count how many of each type of phone number there are in the `phone` column.

11. Run a query to count the total number of order IDs listed in the `order_ids` column. (Hint: You can query an `ARRAY` column in Impala as if it were a separate table with columns `item` and `pos`.)

12. Select the first element in the `order_ids` `ARRAY` for each customer. (Hint: The `pos` column represents the position of each element in an `ARRAY`, indexed from zero.)

13. Run a query to count how many customers have 30 or more orders.

Bonus Exercise 1: Hive Table-Generating Functions

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

1. Run a Hive query to return the order IDs for customer ID 1200866, one per row. (Hint: Use the `explode` function.)

2. Run a Hive query to return the customer IDs and order IDs for all customers who have an average order total of greater than $900 (90,000 cents). The query should return one order ID per row. (Hint: Use `LATERAL VIEW`.)

Bonus Exercise 2: Impala Complex Column Join Notation

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

1. Run an Impala query to return the same result set as the previous Hive query: the customer IDs and order IDs for all customers who have an average order total of greater than $900. (Hint: Use join notation, and remember to query the Parquet-based table.)

2. Now run an Impala query that returns the customer IDs, first names, last names, phone numbers, and phone number types for these customers who have an average order total of greater than $900.

This is the end of the exercise

Hands-On Exercise: Analyzing Text with Hive

Exercise directory:	`$ADIR/exercises/text_analysis`

In this exercise, you will use a Regex SerDe to load web server log data into a table. Afterwards, you will 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:

```
$ $ADIR/scripts/catchup.sh
```

Creating and Populating the Web Logs Table

Many interesting analyses can be done on data from the usage of a website. 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 Regex SerDe and specify a pattern Hive can use to parse lines into individual fields you can then query.

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

1. Examine `create_web_logs.hql` to get an idea of how the `CREATE TABLE` command uses a Regex SerDe 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 Beeline in a terminal window:

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

2. 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/
```

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

```
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;
```

This query may take several minutes to run. 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: Analyzing 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 than structured data. 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.

1. Review the `ratings` table structure using the Hive Query Editor or using the `DESCRIBE` command in the Beeline shell.

2. We want to find the product that customers like most, but we 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;
```

3. 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: Analyzing Rating Comments

We observed in the previous bonus exercise that customers are very dissatisfied with one of the products we sell (the product with ID 1274673). Although numeric ratings helped us identify *which* product that is, they cannot tell us *why* customers don't like the product. We could simply read through all the comments associated with that product to learn this information, but that approach doesn't scale. Next, you will use Hive's text processing support to analyze the comments.

1. 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;
```

2. 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.

3. 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 lower(message) LIKE '%ten times more%'
     LIMIT 3;
```

   You should see comments that say, "Why does the red one cost ten times more than the others?" and "Red is ten times more expensive than the others!"

4. We can infer that customers are complaining about the price of the red-color version of this item. Write and execute a query that will find all distinct comments containing the word "red" that are associated with product ID 1274673.

5. The previous step should have displayed three comments:

   - "Red is ten times more expensive than the others!"
   - "What is so special about red?"
   - "Why does the red one cost ten times more than the others?"

   The first and third comment imply 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.

6. 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 products that are almost identical, differing only in color. However, the product with the negative reviews (the red one) is priced about ten times higher than the others, just as some of the comments said.

The costs for these differently colored products are the same, but the price of the red one is 42999 cents ($429.99) whereas the prices of the other two are both 4299 cents ($42.99). It appears that using text processing on the product reviews may have helped us to uncover a pricing error.

This is the end of the exercise

Hands-On Exercise: Hive Optimization

Exercise directory:	`$ADIR/exercises/hive_optimization`

In this exercise, you will practice techniques to improve Hive query performance.

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:

```
$ $ADIR/scripts/catchup.sh
```

IMPORTANT: Use a single session of Beeline or Hue's Hive Query Editor to complete all the steps of this exercise. Except for this one session, close all other sessions of Beeline and close all other web browser windows containing Hue's Hive Query Editor.

Using Hive with Hadoop Standalone Mode

Dualcore sells 1,114 different products representing 47 different brands. The Marketing Director needs your help to identify which brands have the largest numbers of products, so that advertising campaigns can be planned to increase customer awareness of these brands.

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

1. The following Hive query returns the top 10 brands as measured by the number of products for sale. Paste this HiveQL into Beeline or Hue's Hive Query Editor and execute the query:

```
SELECT brand, COUNT(prod_id) AS num
  FROM products
  GROUP BY brand
  ORDER BY num DESC
  LIMIT 10;
```

2. Observe the amount of time the query takes to complete.

3. View Hive's execution plan by prefixing your query with EXPLAIN or by using the **Explain** button ( ) in Hue. Observe that the Hive query executes as a MapReduce job that includes both a map phase and reduce phase.

4. Because this query processes a very small amount of data (the `products` table contains 1,114 rows and its data is about 60kb in size), a large proportion of the query's running time is consumed by the overhead of the MapReduce job. Because the data is small, we can reduce this overhead by using Hadoop standalone mode to run the query in a single Java Virtual Machine (JVM) on the Hive Server. Enable standalone mode by issuing the Hive command:

```
SET mapreduce.framework.name=local;
```

5. Now execute the same Hive query that was executed in the first step of this exercise again, and observe the amount of time the query takes to complete.

6. In preparation for running other queries on larger datasets, switch back to using distributed mode by issuing the Hive command:

```
SET mapreduce.framework.name=yarn;
```

Using Hive on Spark

Dualcore's Marketing Director is planning an advertising campaign to improve customer awareness of the top-selling brands in each state. She needs your help to identify the top 10 brands as measured by total sales in each state, starting with New York.

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

7. The following Hive query returns the top 10 brands based on total sales for customers in New York. This requires joining four different tables, some of which contain millions of rows, as well as performing aggregation and ordering the result:

```
SELECT brand, SUM(price) AS total
  FROM products p
    JOIN order_details d ON p.prod_id = d.prod_id
    JOIN orders o ON d.order_id = o.order_id
    JOIN customers c ON o.cust_id = c.cust_id
  WHERE c.state='NY'
  GROUP BY brand
  ORDER BY total ASC
  LIMIT 10;
```

View Hive's execution plan by prefixing the query with `EXPLAIN` or by using the **Explain** button ( ) in Hue. Observe that the Hive query executes as a MapReduce job with numerous map and reduce phases.

8. You may execute this query if you wish. You should expect it to take about five minutes to complete.

9. Because this query runs slowly using Hive on MapReduce, we could instead run it using Hive on Spark and compare the query performance using these two engines. To use Spark as Hive's execution engine, issue the Hive command:

```
SET hive.execution.engine=spark;
```

10. Recall that Spark must initialize when you submit the first Hive on Spark query. Before issuing the complex query again to compare performance, first issue a simpler query to cause Spark to initialize. You should expect a long delay before the query completes:

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

11. Once the query completes and returns a result (1,114), issue the same simple query a second time and observe that it returns very quickly.

12. Now issue the complex query that returns the top 10 brands based on total sales for customers in New York, and observe the amount of time the query takes to complete.

13. View the Hive on Spark execution plan by prefixing the query with `EXPLAIN` or by using the **Explain** button in Hue. Observe that the number of stages is much smaller and the execution plan is much simpler than with Hive on MapReduce.

14. End the Spark session and return to using Hive on MapReduce by issuing the Hive command:

```
SET hive.execution.engine=mr;
```

This command will shut down Spark and free up memory on the hands-on environment, preventing out-of-memory errors or performance problems when completing later exercises.

This is the end of the exercise

Hands-On Exercise: Impala Optimization

Exercise directory: `$ADIR/exercises/impala_optimization`

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:

```
$ $ADIR/scripts/catchup.sh
```

Reviewing 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 the following command:

```
COMPUTE STATS products;
```

3. Now view the query plan again, this time without the warning.

4. The previous query was very simple, and included only a single table in the `FROM` clause. Try reviewing the query plan of a more complex query. The following query returns the top three products sold. Before using `EXPLAIN`, 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 each have only a single partition. Try reviewing the query plan for a partitioned table. Recall that in an earlier exercise, you created an `ads` table partitioned on the `network` column. First, compute stats for this table:

```
COMPUTE STATS ads;
```

6. Now 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=2
  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: Reviewing 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 available in Hue. Refer to the "Running Queries from the Shell, Scripts, and Hue" exercise for instructions on how to use the Impala shell, if needed.

1. Try executing one of the queries you examined above; for example:

```
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;
```

2. After the query completes, execute the `SUMMARY` command:

```
SUMMARY;
```

3. **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: 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 an earlier 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:

```
$ $ADIR/scripts/catchup.sh
```

Analyzing Customer Checkouts

As on many websites, 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`	Collect payment information
4	`/cart/checkout/step4-receipt`	Show receipt for completed order

Note: Because the `web_logs` table uses a Regex SerDe, which is a feature not supported by Impala, this section of the exercise must be completed using 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 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 completed only one checkout step (view cart), only two checkout steps (see shipping cost), or all four checkout steps:

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

You should see from the differences in these numbers that many customers abandon their order 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 SerDe, it can be queried in Impala. Try running the same query as in the previous step in Impala. What happens?

Using `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), you can gather enough data from the logs to estimate them.

You don't have the customer's address, but you 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 a `SELECT` statement.

   b. The `SELECT` statement transforms the `cookie`, `ip_address`, and `steps_completed` fields from the `checkout_sessions` table using a Python script (which you'll look at in the next step).

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 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 dfs` 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
```

Extracting a 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 and load a table called `cart_items` with two fields: `cookie` and `prod_id` based on data selected 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 `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, 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).

Creating 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 the products before you can estimate shipping costs. In order to do some more analysis later, you'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 this 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;
```

Creating a Table Using a UDF to Estimate Shipping Costs

You finally have all the information you 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 dfs` 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 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.

This is the end of the exercise

Optional Hands-On Exercise: Analyzing Abandoned Carts

Exercise directory:	`$ADIR/exercises/abandoned_carts`

In this exercise, you will analyze the abandoned cart data you extracted in an earlier exercise.

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:

```
$ $ADIR/scripts/catchup.sh
```

For this exercise, you can use whichever tool you prefer—Hive or Impala—using whichever method you like best, including shell, script, or the Hue Query Editor.

If you plan to use Impala for these exercises, you will first need to invalidate Impala's metadata cache in order to access those tables.

Analyzing Lost Revenue

1. First, calculate how much revenue the abandoned carts represent. Remember, there are four steps in the checkout process, so only records in the `cart_shipping` table with a `steps_completed` value of four represent a completed purchase:

```
SELECT SUM(total_price) AS lost_revenue
    FROM cart_shipping
    WHERE steps_completed < 4;
```

Lost Revenue From Abandoned Shipping Carts

cart_shipping				
cookie	steps_completed	total_price	total_cost	shipping_cost
100054318085	4	6899	6292	425
100060397203	4	19218	17520	552
100062224714	2	7609	7155	556
100064732105	2	53137	50685	839
100107017704	1	44928	44200	720
...	...	...	...	...

Sum of `total_price` where `steps_completed` < 4

You should see that abandoned carts mean that Dualcore is potentially losing out on more than $2 million in revenue. Clearly it's worth the effort to do further analysis.

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.

2. The number returned by the previous query is revenue, but what counts is profit. Gross profit for an item is the price minus the cost. Write and execute a query similar to the one above, but which reports the total lost profit from abandoned carts. If you need a hint on how to write this query, you can check the `solution/abandoned_checkout_profit.sql` file.

   - After running your query, you should see that Dualcore is potentially losing $111,058.90 in profit due to customers not completing the checkout process.

3. How does this compare to the amount of profit Dualcore receives from customers who do complete the checkout process? Modify your previous query to consider only those records where `steps_completed = 4`, and then re-execute it. Check `solution/completed_checkout_profit.sql` if you need a hint.

   - The result should show that we earn a total of $177,932.93 on completed orders, so abandoned carts represent a substantial proportion of additional profits.

4. The previous two queries returned the *total* profit for abandoned and completed orders, but these aren't directly comparable because there were different numbers of each. It might be the case that one is much more profitable than the other on a per-order basis. Write and execute a query that will calculate the *average* profit based on the number of steps completed during the checkout process. If you need help writing this query, check the `solution/checkout_profit_by_step.sql` file.

    - You should observe that carts abandoned after step 2 represent an even higher average profit per order than completed orders.

Calculating Cost and Profit for a Free Shipping Offer

You have observed that most carts—and the most *profitable* carts—are abandoned at the point where we display the shipping cost to the customer. You will now run some queries to determine whether offering free shipping, on at least some orders, would actually bring in more revenue, assuming this offer prompted more customers to finish the checkout process.

5. Run the following query to compare the average shipping cost for orders abandoned after the second step versus completed orders:

```
SELECT steps_completed, AVG(shipping_cost) AS avg_ship_cost
   FROM cart_shipping
   WHERE steps_completed IN (2,4)
   GROUP BY steps_completed;
```

Average Shipping Cost for Carts Abandoned After Steps 2 and 4

cart_shipping				
cookie	steps_completed	total_price	total_cost	shipping_cost
100054318085	4	6899	6292	425
100060397203	4	19218	17520	552
100062224714	2	7609	7155	556
100064732105	2	53137	50685	839
100107017704	1	44928	44200	720
...	...	...	...	...

Average of `shipping_cost` where `steps_completed` = 2 or 4

- You will see that the shipping cost for abandoned orders was almost 10% higher than for completed purchases. Offering free shipping, at least for some orders, might actually bring in more money than passing on the cost to customers and risking abandoned orders.

6. Run the following query to determine the average profit per order over the entire month for the data you are analyzing in the log file. This will help you to determine whether we could absorb the cost of offering free shipping:

```
SELECT AVG(price - cost) AS avg_profit
  FROM products p
  JOIN order_details d
    ON (d.prod_id = p.prod_id)
  JOIN orders o
    ON (d.order_id = o.order_id)
 WHERE YEAR(order_date) = 2013
       AND MONTH(order_date) = 05;
```

Average Profit per Order, May 2013

products

prod_id	price	cost
1273641	1839	1275
1273642	1949	721
1273643	2149	845
1273644	2029	763
1273645	1909	1234
...	...	...

order_details

order_id	product_id
6547914	**1273641**
6547914	**1273644**
6547914	**1273645**
6547915	**1273645**
6547916	**1273641**
...	...

orders

order_id	order_date
6547914	2013-05-01 00:02:08
6547915	2013-05-01 00:02:55
6547916	2013-05-01 00:06:15
6547917	2013-06-12 00:10:41
6547918	2013-06-12 00:11:30
...	...

Average the profit...

... on orders made in May, 2013

- You should see that the average profit for all orders during May was $7.80. An earlier query you ran showed that the average shipping cost was $8.83 for completed orders and $9.66 for abandoned orders, so clearly we would lose money by offering free shipping on all orders. However, it might still be worthwhile to offer free shipping on orders over a certain amount.

7. Run the following query, which is a slightly revised version of the previous one, to determine whether offering free shipping only on orders of $10 or more would be a good idea:

```
SELECT AVG(price - cost) AS avg_profit
  FROM products p
  JOIN order_details d
    ON (d.prod_id = p.prod_id)
  JOIN orders o
    ON (d.order_id = o.order_id)
  WHERE YEAR(order_date) = 2013
        AND MONTH(order_date) = 05
        AND price >= 1000;
```

- You should see that our average profit on orders of $10 or more was $9.09, so absorbing the cost of shipping would leave very little profit.

8. Repeat the previous query, modifying it slightly each time to find the average profit on orders of at least $50, $100, and $500.

- You should see that there is a huge spike in the amount of profit for orders of $500 or more ($111.05 on average for these orders).

9. How much does shipping cost on average for orders totaling $500 or more? Write and run a query to find out (`solution/avg_shipping_cost_50000.sql` contains the solution, in case you need a hint).

- You should see that the average shipping cost is $12.28, which happens to be about 11% of the profit on those orders.

10. Since there's no way to know in advance who will abandon their cart, Dualcore would have to absorb the $12.28 average cost on *all* orders of at least $500. Would the extra money from abandoned carts offset the added cost of free shipping for customers who would have completed their purchases anyway? Run the following query to see the total profit on completed purchases:

```
SELECT SUM(total_price - total_cost) AS total_profit
   FROM cart_shipping
   WHERE total_price >= 50000
     AND steps_completed = 4;
```

- After running this query, you should see that the total profit for completed orders is $107,582.97.

11. Now, run the following query to find the potential profit, after subtracting shipping costs, if all customers completed the checkout process:

```
SELECT SUM(total_price - total_cost - shipping_cost)
      AS potential_profit
   FROM cart_shipping
   WHERE total_price >= 50000;
```

Since the result of $120,355.26 is greater than the current profit of $107,582.97 from completed orders, it appears that Dualcore could earn nearly $13,000 more by offering free shipping for all orders of at least $500.

Congratulations! Your hard work analyzing a variety of data with Hadoop's tools has helped make Dualcore more profitable than ever.

This is the end of the exercise

Bonus Hands-On Exercise: Extending Pig with Streaming and UDFs

Exercise Directory:	`$ADIR/exercises/extending_pig`

In this exercise, you will use `STREAM` in Pig to analyze metadata from Dualcore's customer service call recordings to identify the cause of a sudden increase in complaints. You will then use this data in conjunction with a user-defined function to propose a solution for resolving the problem.

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:

```
$ $ADIR/scripts/catchup.sh
```

When prompted, enter the number for the previous exercise, "Analyzing Abandoned Carts."

Background Information

Dualcore outsources its call center operations, and our costs have recently risen due to an increase in the volume of calls handled by these agents. Unfortunately, we do not have access to the call center's database, but they provide us with recordings of these calls stored in MP3 format. By using Pig's `STREAM` to invoke a provided Python script, you can extract the category and timestamp from the files, and then analyze that data to learn what is causing the recent increase in calls.

Extracting Call Metadata

Note: Since the Python library that the script uses to extract tags doesn't support HDFS, we run this script in local mode on a small sample of the call recordings. Because you will use Pig's local mode, there will be no need to "ship" the script to the nodes in the cluster.

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

```
$ cd $ADIR/exercises/extending_pig
```

2. A programmer on our team provided us with a Python script (`readtags.py`) for extracting the metadata from the MP3 files. This script takes the path of a file on the command line and returns a record containing five tab-delimited fields: the file path, the call category, the agent ID, the customer ID, and the timestamp for when the agent answered the call.

   Your first step is to create a text file containing the paths of the files to analyze, with one line for each file. You can easily create the data in the required format by capturing the output of the UNIX `find` command:

```
$ find $ADIR/data/cscalls/ -name '*.mp3' > call_list.txt
```

3. Edit the `extract_metadata.pig` file and make the following changes:

   a. Replace the hardcoded parameter in the `SUBSTRING` function used to filter by month with a parameter named `MONTH` whose value you can assign on the command line. This will make it easy to check the leading call categories for different months without having to edit the script.

   b. Add the code necessary to count calls by category

   c. Display the top three categories (based on number of calls) to the screen.

4. Once you have made these changes, run your script to check the top three categories in the month before Dualcore started the online advertising campaign:

```
$ pig -x local -p MONTH=2013-04 extract_metadata.pig
```

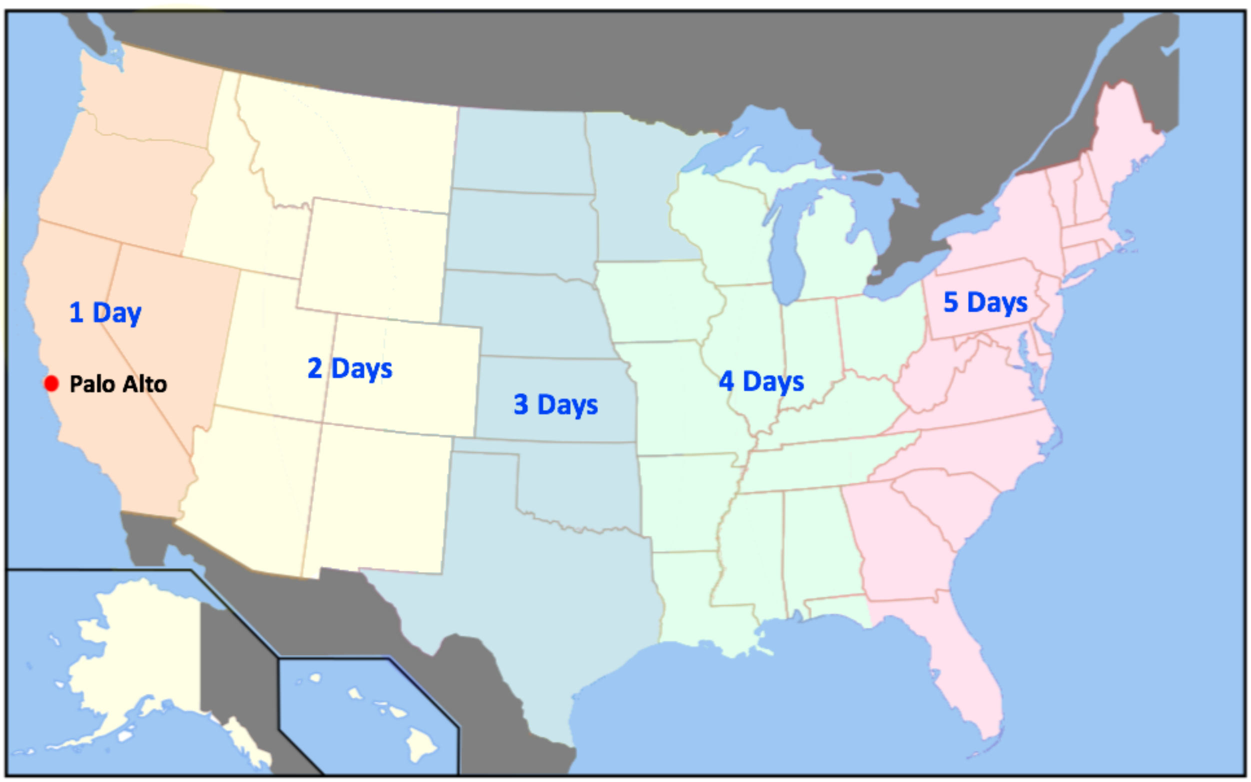
5. Now run the script again, this time specifying the parameter for May:

```
$ pig -x local -p MONTH=2013-05 extract_metadata.pig
```

The output should confirm that not only is call volume substantially higher in May, the `SHIPPING_DELAY` category has more than twice as many calls as the other two combined.

Choosing the Best Location for a Distribution Center

The analysis you just completed uncovered a problem. Dualcore's Vice President of Operations launched an investigation based on your findings and has now confirmed the cause: our online advertising campaign is indeed attracting many new customers, but many of them live far from Dualcore's only distribution center in Palo Alto, California. All our shipments are transported by truck, so an order can take up to five days to deliver depending on the customer's location.



To solve this problem, Dualcore will open a new distribution center to improve shipping times.

The ZIP codes for the three proposed sites are 02118, 63139, and 78237. You will look up the latitude and longitude of these ZIP codes, as well as the ZIP codes of customers who have recently ordered, using a supplied dataset. Once you have the coordinates, you will use the `HaversineDistInMiles` UDF distributed with DataFu to determine how far each customer is from the three distribution centers. You will then calculate the average distance for all customers to each of these distribution centers in order to propose the one that will benefit the most customers.

6. The `latlon.tsv` file on your local drive is a tab-delimited file that maps ZIP codes to latitude/longitude points. Add it to HDFS:

```
$ hdfs dfs -mkdir /dualcore/distribution
$ hdfs dfs -put $ADIR/data/latlon.tsv \
/dualcore/distribution/
```

7. A co-worker provided a script (`create_cust_location_data.pig`) that finds the ZIP codes for customers who placed orders during the period we ran the ad campaign. It also excludes customers who are already close to our current facility, as well as customers in the remote states of Alaska and Hawaii (where orders are shipped by airplane). The Pig Latin code joins these customers' ZIP codes with the latitude/longitude dataset uploaded in the previous step, then writes those three columns (ZIP code, latitude, and longitude) as the result. Examine the script to see how it works, and then run it to create the customer location data in HDFS:

```
$ pig create_cust_location_data.pig
```

8. The `HaversineDistInMiles` function calculates the distance from each customer to each of the three proposed warehouse locations. This function requires us to supply the latitude and longitude of both the customer and the warehouse. While the script you just executed created the latitude and longitude for each customer, you must create a dataset containing the ZIP code, latitude, and longitude for these warehouses. Do this by running the following UNIX command:

```
$ egrep '^02118|^63139|^78237' \
   $ADIR/data/latlon.tsv > warehouses.tsv
```

9. Next, add this file to HDFS:

```
$ hdfs dfs -put warehouses.tsv /dualcore/distribution/
```

10. Edit the `calc_average_distances.pig` file. The UDF is already registered and an alias for this function named `DIST` is defined at the top of the script, just before the two datasets you will use are loaded. You need to complete the rest of this script:

    a. Create a record for every combination of customer and proposed distribution center location

    b. Use the UDF to calculate the distance from the customer to the warehouse

    c. Calculate the average distance for all customers to each warehouse

    d. Display the result to the screen

11. After you have finished implementing the Pig Latin code described above, run the script:

```
$ pig calc_average_distances.pig
```

Question: Which of these three proposed ZIP codes has the smallest average number of miles to our customers?

This is the end of the exercise

Data Model Reference

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	email 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	company	Company name	ACME Inc.
2	contact	Full name of contact	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
`latlon`	42968
`loyalty_program`	311
`loyalty_program_parquet`	311
`order_details`	3,333,244
`orders`	1,662,951
`products`	1,114
`ratings`	21,997
`suppliers` (renamed `vendors`)	66
`web_logs`	412,860

Other Data Added to HDFS

The following describes the structure of other important datasets 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 websites and are ignored in our Regex SerDe.

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

`\\d`	Matches any single digit
`\\w`	Matches any word character (letters of any case, plus digits or underscore)
`\\s`	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

`\\D`	Matches any single non-digit character
`\\W`	Matches any non-word character
`\\S`	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

`{5}`	Preceding character may occur exactly five times
`{0,6}`	Preceding character may occur between zero and six times
`?`	Preceding character is optional (may occur zero or one times)
`+`	Preceding character may occur one or more times
`*`	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 are "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

`^`	Matches only at the beginning of a string
`$`	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.