cloudera®

Ask Bigger Questions

Cloudera Data Analyst Training: Using Pig, Hive, and Impala with Hadoop Hands-On Exercises

cloudera®

General Notes

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

Points to Note While Working in the VM

1. The VM is set to automatically log in as the user `training`. Should 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:

    ```
    $ hadoop fs -put accounting_reports_taxyear_2013 \
    /user/training/tax_analysis/
    ```

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

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

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

Class-Specific VM Customization

Your VM is used in several of Cloudera's training classes. This particular class does not require some of the services that start by default, while other services that do not start by default are helpful for this class. We have scripted the class-specific customizations and strongly recommend that you run the following command whenever you start the VM:

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

This will conserve memory and increase performance of the virtual machine. After running this command, you may safely ignore any messages about services that have already been started or shut down.

Points to Note During the Exercises

Sample Solutions

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

Catch-up Script

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

$ADIR Environment Variable

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

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

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

Bonus Exercises

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

Hands-On Exercise: Data Ingest With Hadoop Tools

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

Prepare your Virtual Machine

Launch the VM if you haven't already done so, and then run the following command to boost performance by disabling services that are not needed for this class:

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

Step 1: Exploring HDFS

1. Open a terminal window (if one is not already open) by double-clicking the Terminal icon on the desktop. Next, change to the directory for this exercise by running the following command:

```
$ cd $ADIR/exercises/data_ingest
```

2. You can use the `hadoop fs` command to interact with the Hadoop Distributed Filesystem (HDFS) from the command line. In the terminal window, enter:

```
$ hadoop fs
```

This displays a help message describing all subcommands associated with `hadoop fs`.

3. Run the following command:

```
$ hadoop fs -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. Since your user ID on the cluster is `training`, your home directory in HDFS is `/user/training`.

4. To see the contents of your home directory (which is currently empty), run the following command:

```
$ hadoop fs -ls /user/training
```

5. If you do not specify a path, `hadoop fs` assumes you are referring to your home directory. Therefore, the following command is equivalent to the one above:

```
$ hadoop fs -ls
```

6. Most of your work will be in the `/dualcore` directory, so create that now:

```
$ hadoop fs -mkdir /dualcore
```

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

```
$ hadoop fs -put $ADIR/data/access.log /dualcore
```

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:
```
$ hadoop fs -rm /dualcore/example.txt
```

To remove a directory (recursively):
```
$ hadoop fs -rm -r /dualcore/example/
```

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

```
$ hadoop fs -ls /dualcore
```

9. To practice removing a file, you may now delete the file you just added.

```
$ hadoop fs -rm /dualcore/access.log
```

Step 2: 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.

1. Log in to MySQL and select the dualcore database:

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

2. 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;
```

3. 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;
```

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

```
mysql> quit
```

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

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

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

> **Hiding Passwords**
>
> Typing the database password on the command line is a potential security risk since
> others may see it. An alternative to using the `--password` argument is to use `-P` and let
> Sqoop prompt you for the password, which is then not visible when you type it.
>
> **Sqoop Code Generation**
>
> After running the `sqoop import` command above, you may notice a new file named
> `employee.java` in your local directory. This is an artifact of Sqoop's code generation
> and is really only of interest to Java developers, so you can ignore it.

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

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

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

9. Next, you will import the `order_details` table into HDFS. The command is slightly different because this table only holds references to records in the `orders` and `products` table, and lacks a primary key of its own. Consequently, you will need to specify the `--split-by` option and instruct Sqoop to divide the import work among map 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
```

Bonus Exercise #1: Running a Python MapReduce Job

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

Dualcore's Human Resources manager requested that we provide him with the number of employees with salaries of at least $75,000, grouped by state, to help him plan meetings to inform those employees of a recent retirement plan change that affects them.

A software engineer on our team recently attended Cloudera's Developer training and was eager to try writing some MapReduce code in Python, so you will briefly examine the code she wrote and then run it to produce the data our HR department requested.

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

```
$ cd bonus_01
```

2. Examine the MapReduce code for both the mapper and reducer so you can see how it will produce the data we have been asked to provide:

```
$ cat mapper.py
$ cat reducer.py
```

3. Next, examine the shell script that the software engineer wrote:

```
$ cat runjob.sh
```

As you can see, this job defines the path of the Java library (JAR) file that contains support for Hadoop Streaming. It also defines the output directory, making sure that it does not already exist (for example, if a previous job had already created it) before submitting the job for execution on the cluster.

4. Execute this shell script to run the job:

```
$ ./runjob.sh
```

Hadoop should start displaying status messages to your screen within a few seconds, and the job should run for a few minutes before concluding with a message explaining that the output is in the `/user/training/empcounts` directory (in HDFS) and returning control to your terminal.

5. List the contents of this directory in HDFS:

```
$ hadoop fs -ls /user/training/empcounts
```

You should see three types of items in this directory: a `_SUCCESS` file indicating that the job completed successfully, a `_logs` directory containing log files from the job (if any were produced), and one or more files whose names start with `part`. These files contain the actual output records created by the job.

6. Since our job simply produces a small summary of the input data, it would be more convenient to retrieve the contents of the HDFS output directory and merge them into a single file on the local disk. The `hadoop fs -getmerge` command does exactly this:

```
$ hadoop fs -getmerge /user/training/empcounts results.txt
```

The `results.txt` file will not include any data from the `_SUCCESS` file or `_logs` subdirectory in HDFS because this command ignores files whose names begin with an underscore or a period (dot).

7. Now that this is a local file, you can examine its output with the UNIX `cat` command:

```
$ cat results.txt
```

You should now be able to answer the following questions:

    a. How many states match the criteria specified by the Human Resources department?

    b. Which state contains the most employees who match these criteria? (Hint: it is the same state where Dualcore was founded, where the corporate headquarters are located, and where we have the most employees overall.)

Bonus Exercise #2: Running a Java MapReduce Job

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

The software engineer who wrote the original Python MapReduce job was so excited about what she learned in Cloudera's Developer class that she wanted to create a MapReduce job in Java based on the Python one you just ran. This job is slightly different in that it identifies states with at least 25 employees having a salary of $50,000 or more.

1. Change to the `bonus_02` subdirectory for this hands-on exercise:

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

If you happen to know the Java programming language, have a look at the code in the `java/src/example/` subdirectory. It contains comments that explain the code for the Mapper and Reducer, as well as the Driver class that configures and submits the job to the cluster.

2. The software engineer wrote a script that uses the Apache Ant build system to compile, package, and run the code. This means you can perform all of these steps by typing just one command:

```
$ ant run
```

3. Once the job is complete, use `hadoop fs -cat` to display the output in HDFS directly to the terminal (note: the Java job is configured to write its output to a slightly different output path than the Python job):

```
$ hadoop fs -cat /user/training/empcounts_java/part*
```

This is the end of the Exercise

Hands-On Exercise: Using Pig for ETL Processing

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:

```
$ ~/scripts/analyst/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 they've 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:

- Filter invalid records

- Reorder fields

- Correct inconsistencies

- Write the corrected data to HDFS

Step #1: Working in the Grunt Shell

In this step, you will practice running Pig commands in the Grunt shell.

1. Change to the directory for this hands-on 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 in Pig to review the schema of first_2_cols:

```
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;
```

Step #2: Processing Input Data from the First Ad Network

In this step, you will process the input data from the first ad network. First, you will create a Pig script in a file, and then you will run the script. Many people find working this way easier than working directly in the Grunt shell.

1. Edit the `first_etl.pig` file to complete the `LOAD` statement and read the data from the sample you just created. 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`

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

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

Make sure the output looks correct (i.e., that you have the fields in the expected order and the values appear similar in format to that shown in the table above) before you continue with the next step.

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

    a. Update your script to filter out all records where the country field does not contain `USA`.

b. We need to store the fields in a different order than we received them. Use a `FOREACH … GENERATE` statement to create a new relation containing the fields in the same order as shown in the following table (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 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).

4. Add the complete data file to HDFS:

```
$ hadoop fs -put $ADIR/data/ad_data1.txt /dualcore
```

5. 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`).

6. 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.

7. Run this script in Pig's MapReduce 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.

8. Check the first 20 output records that your script wrote to HDFS and ensure they look correct (you can ignore the message "cat: Unable to write to output stream"; this simply happens because you are writing more data with the `fs -cat` command than you are reading with the `head` command):

```
$ hadoop fs -cat /dualcore/ad_data1/part* | head -20
```

    a. Are the fields in the correct order?

    b. Are all the keywords now in uppercase?

Step #3: 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.

1. 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
```

2. 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:

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	Where on page was ad displayed	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

3. 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
```

4. 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) and also use the `UPPER` and `TRIM` functions to correct the `keyword` field as you did earlier:

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 data set 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.

5. Once you are sure the script works locally, add the full data set to HDFS:

```
$ hadoop fs -put $ADIR/data/ad_data2.txt /dualcore
```

6. 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.

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

```
$ pig second_etl.pig
```

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

```
$ hadoop fs -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

During the previous exercise, you performed ETL processing on data sets 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:

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

Step #1: Find Low Cost Sites

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:

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

You can ignore the message "cat: Unable to write to output stream," which appears because you are writing more data with the `fs -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?

Step #2: Find 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.

1. 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
```

2. 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

3. 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: Count 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: Estimate 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. 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 total_click_count.pig project_next_campaign_cost.pig
```

2. 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 this field belongs to, in order to account for the removal of the `FILTER` statement in the previous 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.

3. 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/sample_solution/` subdirectory.

Bonus Exercise #3: Calculating 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 ad campaign, but 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 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 where 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 clickthrough 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 `sample_solution` directory.

This is the end of the Exercise

Hands-On Exercise: Analyzing Disparate Data Sets with Pig

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:

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

Step #1: Show Per-Month Sales Before and After Campaign

Before we proceed with more sophisticated analysis, you should 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?

Step #2: Count Advertised Product Sales by Month

Our analysis from the previous step 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 data sets 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.

1. 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

2. 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: Calculate 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. Exclude any orders which do not contain the advertised product (product ID #1274348)

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

   d. Count the total number of products per 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: Segment Customers for Loyalty Program

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 at 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 data sets 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 data sets 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 `hadoop fs -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):

```
$ hadoop fs -getmerge /dualcore/loyalty/platinum platinum.txt
$ hadoop fs -getmerge /dualcore/loyalty/gold gold.txt
$ hadoop fs -getmerge /dualcore/loyalty/silver silver.txt
```

5. Use the UNIX `head` and/or `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: Extending Pig with Streaming and UDFs

In this exercise you will use the `STREAM` keyword 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:

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

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` keyword 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.

Step #1: Extract Call Metadata

Note: Since the Python library we are using for extracting the 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, call category, agent ID, customer ID, and the timestamp of 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 -param MONTH=2013-04 extract_metadata.pig
```

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

```
$ pig -x local -param 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 the amount of calls as the other two.

Step #2: Choose Best Location for 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.
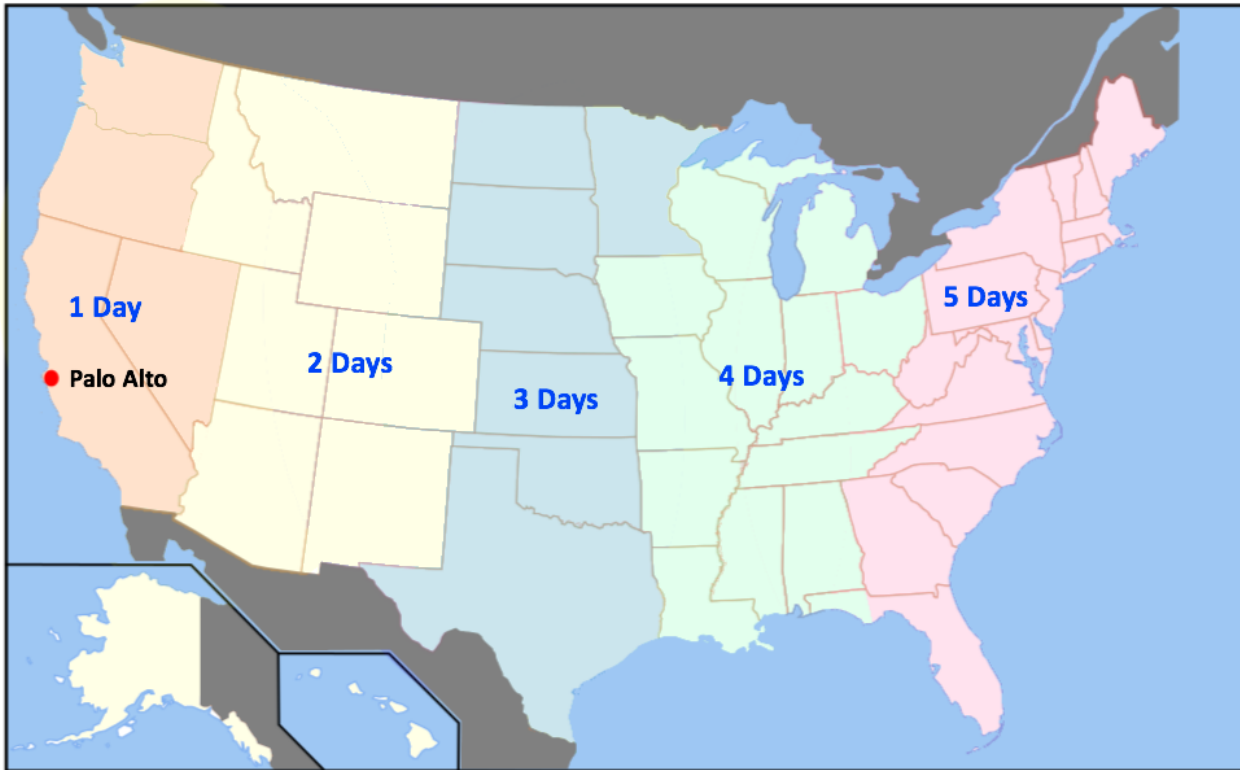
To solve this problem, Dualcore will open a new distribution center to improve shipping times.

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 data set. Once you have the coordinates, you will invoke the use the `HaversineDistInMiles` UDF distributed with DataFu to determine how far each customer is from the three data centers. You will then calculate the average distance for all customers to each of these data centers in order to propose the one that will benefit the most customers.

1. Add the tab-delimited file mapping ZIP codes to latitude/longitude points to HDFS:

```
$ hadoop fs -mkdir /dualcore/distribution
$ hadoop fs -put $ADIR/data/latlon.tsv \
/dualcore/distribution
```

2. 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 the ones 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 data set 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
```

3. You will use the `HaversineDistInMiles` function to calculate 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 data set 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
```

4. Next, add this file to HDFS:

```
$ hadoop fs -put warehouses.tsv /dualcore/distribution
```

5. 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 data sets 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 function 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

6. 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 lowest average mileage to our customers?

This is the end of the Exercise

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

In this exercise you will write HiveQL queries to analyze data in Hive tables that have been populated with data you placed in HDFS during earlier exercises.

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

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

Step #1: Running a Query from the Hive Shell

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 Hive query that identifies the winner's record in our customer database so that we can send her the $5,000 prize.

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

```
$ cd $ADIR/exercises/analyzing_sales
```

2. Start Hive:

```
$ hive
```

> ### Hive Prompt
>
> To make it easier to copy queries and paste them into your terminal window, we do not show the **hive>** prompt in subsequent steps. Steps prefixed with $ should be executed on the UNIX command line; the rest should be run in Hive unless otherwise noted.

3. Make the query results easier to read by setting the property that will make Hive show column headers:

```
set hive.cli.print.header=true;
```

4. All you know about the winner is that her name is Bridget and she lives in Kansas City. Use Hive's LIKE operator to do a wildcard search for names such as "Bridget", "Bridgette" or "Bridgitte". Remember to filter on the customer's city.

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

Step #2: Running a Query Directly from the Command Line

You will now run a top-N query to identify the three most expensive products that Dualcore currently offers.

1. Exit the Hive shell and return to the command line:

```
quit;
```

2. Although HiveQL statements are terminated by semicolons in the Hive shell, it is not necessary to do this when running a single query from the command line using the `-e` option. Run the following command to execute the quoted HiveQL statement:

```
$ hive -e 'SELECT price, brand, name FROM PRODUCTS ORDER BY
price DESC LIMIT 3'
```

Question: Which three products are the most expensive?

Step #3: Running a HiveQL Script

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, it's a perfect case for running it as a Hive script.

1. Study the HiveQL code for the query to learn how it works:

```
$ cat verify_tablet_order.hql
```

2. Execute the HiveQL script using the hive command's `-f` option:

```
$ hive -f verify_tablet_order.hql
```

Question: Did Bridget order the advertised tablet in May?

Step #4: Running a Query Through Hue and Beeswax

Another way to run Hive queries is through your Web browser using Hue's Beeswax application. 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 Hive itself – because it does not require any software other than a browser.

1. Start the Firefox Web browser by clicking the orange and blue icon near the top of the VM window, just to the right of the System menu. Once Firefox starts, type `http://localhost:8888/` into the address bar, and then hit the enter key.

2. After a few seconds, you should see Hue's login screen. Enter `training` in both the username and password fields, and then click the "Sign In" button. If prompted to remember the password, decline by hitting the ESC key so you can practice this step again later if you choose.

   Although several Hue applications are available through the icons at the top of the page, the Beeswax query editor is shown by default.

3. Select `default` from the database list on the left side of the page.

4. Write a query in the text area that will count the number of records in the `customers` table, and then click the "Execute" button.

   Question: How many customers does Dualcore serve?

5. Click the "Query Editor" link in the upper left corner, and then write and run a query to find the ten states with the most customers.

   Question: Which state has the most customers?

Bonus Exercise #1: Calculating Revenue and Profit

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

Several more questions are described below and you will need to write the HiveQL code to answer them. You can use whichever method you like best, including Hive shell, Hive Script, or Hue, to run your queries.

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

- What was Dualcore's total revenue in May, 2013?

- What was Dualcore's gross profit (sales price minus cost) in May, 2013?

- The results of the above queries are shown in cents. Rewrite the gross profit query to format the value in dollars and cents (e.g., $2000000.00). To do this, you can divide the profit by 100 and format the result using the `PRINTF` function and the format string `"$%.2f"`.

There are several ways you could write each query, and you can find one solution for each problem in the `bonus_01/sample_solution/` directory.

This is the end of the Exercise

Hands-On Exercise: Data Management with Hive

In this exercise you will practice using several common techniques for creating and populating Hive tables. You will also create and query a table containing each of the complex field types we studied: array, map, and struct.

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

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

Additionally, many of the commands you will run use environmental variables and relative file paths. It is important that you use the Hive shell, rather than Hue or another interface, as you work through the steps that follow.

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

You used Sqoop in an earlier exercise to import data from MySQL into HDFS. Sqoop can also create a Hive 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. Change to the directory for this Hands-On Exercise:

```
$ cd $ADIR/exercises/data_mgmt
```

2. Execute the following command to import the `suppliers` table from MySQL as a new Hive-managed table:

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

3. Start Hive:

```
$ hive
```

4. It is always a good idea to validate data after adding it. Execute the Hive query shown below to count the number of suppliers in Texas:

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

The query should show that nine records match.

Step #2: Create an External Table in Hive

You imported data from the `employees` table in MySQL in an earlier exercise, but it would be convenient to be able to query this from Hive. Since the data already exists in HDFS, this is a good opportunity to use an external table.

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

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

2. Run the following Hive 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.

Step #3: Create and Load a Hive-Managed Table

Next, you will create and then load a Hive-managed table with product ratings data.

1. Create a table named `ratings` for storing tab-delimited records using this structure:

Field Name	Field Type
posted	TIMESTAMP
cust_id	INT
prod_id	INT
rating	TINYINT
message	STRING

2. Show the table description and verify that its fields have the correct order, names, and types:

```
DESCRIBE ratings;
```

3. Next, open a separate terminal window (File -> Open Terminal) so you can run the following shell command. This will populate the table directly by using the `hadoop fs` command to copy product ratings data from 2012 to that directory in HDFS:

```
$ hadoop fs -put $ADIR/data/ratings_2012.txt \
/user/hive/warehouse/ratings
```

Leave the window open afterwards so that you can easily switch between Hive and the command prompt.

4. Next, verify that Hive can read the data we just added. Run the following query in Hive to count the number of records in this table (the result should be 464):

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

5. Another way to load data into a Hive table is through the `LOAD DATA` command. The next few commands will lead you through the process of copying a local file to HDFS and loading it into Hive. First, copy the 2013 ratings data to HDFS:

```
$ hadoop fs -put $ADIR/data/ratings_2013.txt /dualcore
```

6. Verify that the file is there:

```
$ hadoop fs -ls /dualcore/ratings_2013.txt
```

7. Use the `LOAD DATA` statement in Hive to load that file into the `ratings` table:

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

8. The `LOAD DATA INPATH` command *moves* the file to the table's directory. Verify that the file is no longer present in the original directory:

```
$ hadoop fs -ls /dualcore/ratings_2013.txt
```

9. Verify that the file is shown alongside the 2012 ratings data in the table's directory:

```
$ hadoop fs -ls /user/hive/warehouse/ratings
```

10. Finally, count the records in the ratings table to ensure that all 21,997 are available:

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

Step #4: Create, Load, and Query a Table with Complex Fields

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

1. Run the following statement in Hive to create the table:

```
CREATE TABLE loyalty_program
    (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>)
   ROW FORMAT DELIMITED
      FIELDS TERMINATED BY '|'
      COLLECTION ITEMS TERMINATED BY ','
      MAP KEYS TERMINATED BY ':';
```

2. Examine the data in `loyalty_data.txt` to see how it corresponds to the fields in the table and then load it into Hive:

```
LOAD DATA LOCAL INPATH 'loyalty_data.txt' INTO TABLE
loyalty_program;
```

3. Run a query to select the `HOME` phone number (hint: map keys are case-sensitive) for customer ID 1200866. You should see 408-555-4914 as the result.

4. Select the third element from the `order_ids` array for customer ID 1200866 (hint: elements are indexed from zero). The query should return 5278505.

5. Select the `total` attribute from the `order_value` struct for customer ID 1200866. The query should return 401874.

Bonus Exercise #1: Alter and Drop 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_01/sample_solution/` subdirectory.

1. Use `ALTER TABLE` to rename the `level` column to `status`.

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

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

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 (the result should be "SILVER"):

    ```
    SELECT status FROM reward_program WHERE cust_id = 1200866;
    ```

5. As sometimes happens in the corporate world, priorities have shifted and the program is now canceled. Drop the `reward_program` table.

This is the end of the Exercise

Optional Hands-On Exercise: Gaining Insight with Sentiment Analysis

In this optional exercise, you will use Hive's text processing features to analyze customers' comments and product ratings. You will uncover problems and propose potential solutions.

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

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

Background Information

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

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

Step #1: Analyze Numeric Product Ratings

Before delving into text processing, you'll begin by analyzing the numeric ratings customers have assigned to various products.

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

```
$ cd $ADIR/exercises/sentiment
```

2. Start Hive and use the `DESCRIBE` command to remind yourself of the table's structure.

3. We want to find the product that customers like most, but must guard against being misled by products that have few ratings assigned. Run the following query to find the product with the highest average among all those with at least 50 ratings:

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

4. 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.

Step #2: Analyze Rating Comments

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

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 in Hive:

```
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 message LIKE '%ten times more%'
    LIMIT 3;
```

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

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

5. The previous step should have displayed two comments:

   - "What is so special about red?"

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

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

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 almost identical products, but the product with the negative reviews (the red one) costs about ten times as much as the others, just as some of the comments said.

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

This is the end of the Exercise

Hands-On Exercise: Data Transformation with Hive

In this exercise you will create and populate a table with log data from Dualcore's Web server. Queries on that data will reveal that many customers abandon their shopping carts before completing the checkout process. You will create several additional tables, using data from a `TRANSFORM` script and a supplied UDF, which you will use later to analyze how Dualcore could turn this problem into an opportunity.

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

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

Step #1: Create and Populate the Web Logs Table

Typical log file formats are not delimited, so you will need to use the RegexSerDe and specify a pattern Hive can use to parse lines into individual fields you can then query.

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

```
$ cd $ADIR/exercises/transform
```

2. Examine the `create_web_logs.hql` script to get an idea of how it uses a RegexSerDe to parse lines in the log file (an example log line is shown in the comment at the top of the file). When you have examined the script, run it to create the table in Hive:

```
$ hive -f create_web_logs.hql
```

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

```
$ hadoop fs -put $ADIR/data/access.log /dualcore/web_logs
```

4. Start the Hive shell in another terminal window

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

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

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

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

Step #2: Analyze Customer Checkouts

You've just queried the logs to see what users search for on Dualcore's Web site, but now you'll run some queries to learn whether they buy. As on many Web sites, customers add products to their shopping carts and then follow a "checkout" process to complete their purchase. Since each part of this four-step process can be identified by its URL in the logs, we can use a regular expression to easily identify them:

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

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

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

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

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

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

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

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

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

Step #3: Use TRANSFORM for IP Geolocation

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

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

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

1. Examine the `create_cart_zipcodes.hql` script and observe the following:

    a. It creates a new table called `cart_zipcodes` based on select statement.

    b. That select statement transforms the `ip_address`, `cookie`, and `steps_completed` fields from the `checkout_sessions` table using a Python script.

    c. The new table contains the ZIP code instead of an IP address, plus the other two fields from the original table.

2. 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 are printed to standard output.

3. Run the script to create the `cart_zipcodes` table:

```
$ hive -f create_cart_zipcodes.hql
```

Step #4: Extract List of Products Added to Each Cart

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

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

1. Write a HiveQL statement to create 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. Add 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 `sample_solution/create_cart_items.hql` file.

2. Execute the HiveQL statement from you just wrote.

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

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

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

Step #5: Create Tables to Join Web Logs with Product Data

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

cloudera®

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

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

Step #6: Create a Table Using a UDF to Estimate Shipping Cost

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

1. Before you can use a UDF, you must add it to Hive's classpath. Run the following command in Hive to do that:

```
ADD JAR geolocation_udf.jar;
```

2. Next, you must 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 our UDF with the alias `CALC_SHIPPING_COST`:

```
CREATE TEMPORARY FUNCTION CALC_SHIPPING_COST AS
'com.cloudera.hive.udf.UDFCalcShippingCost';
```

3. 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;
```

4. 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

Hands-On Exercise: Interactive Analysis with Impala

In this exercise you will examine abandoned cart data using the tables created in the previous exercise. You will use Impala to quickly determine how much lost revenue these abandoned carts represent and use several "what if" scenarios to determine whether we should offer free shipping to encourage customers to complete their purchases.

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

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

Step #1: Start the Impala Shell and Refresh the Cache

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

```
$ cd $ADIR/exercises/interactive
```

2. First, start the Impala shell:

```
$ impala-shell
```

3. Since you created tables and modified data in Hive, Impala's cache of the metastore is outdated. You must refresh it before continuing by entering the following command in the Impala shell:

```
REFRESH;
```

Step #2: Calculate Lost Revenue

1. First, you'll 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. We calculate gross profit by subtracting the cost from the price. 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 `sample_solution/abandoned_checkout_profit.sql` file.

    - After running your query, you should see that we are 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 we receive from customers who do complete the checkout process? Modify your previous query to consider only those records where `steps_completed = 4`, and then execute it in the Impala shell. Check `sample_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 told us 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 `sample_solution/checkout_profit_by_step.sql` file.

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

Step #3: Calculate Cost/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.

1. 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 ship_cost
   FROM cart_shipping
   WHERE steps_completed = 2 OR steps_completed = 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 of 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 and risking abandoned orders.

2. 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 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
...	...	...

Average the profit...

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
...	...

... 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.

3. 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:

```sql
SELECT AVG(price - cost) AS 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.

4. 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 (we make $111.05 on average for these orders).

5. How much does shipping cost on average for orders totaling $500 or more? Write and run a query to find out (`sample_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 we bring in on those orders.

6. Since we won't know in advance who will abandon their cart, we would have to absorb the $12.28 average cost on *all* orders of at least $500. Would the extra money we might bring in 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.

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

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

Since the result of $120,355.26 is greater than the current profit of $107,582.97 we currently earn from completed orders, it appears that we 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

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	e-mail address	`br5331404@example.com`
9	active	Is actively employed?	`Y`
10	salary	Annual pay (in dollars)	`136900`

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

Index	Field	Description	Example
0	**order_id**	Order ID	`3213254`
1	cust_id	Customer ID	`1846532`
2	order_date	Date/time of order	`2013-05-31 16:59:34`

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

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

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

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

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

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

Hive Tables

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

Table Name	Record Count
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
order_details	3,333,244
orders	1,662,951
products	1,114
ratings	21,997
web_logs	412,860

Other Data Added to HDFS

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

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

- `/dualcore/ad_data1` (438,389 records)

- `/dualcore/ad_data2` (350,563 records).

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

access.log: 412,860 records (uploaded to `/dualcore/access.log`)
This file is used to populate the `web_logs` table in Hive. Note that the RFC 931 and Username fields are seldom populated in log files for modern public Web sites and are ignored in our RegexSerDe.

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

Regular Expression Reference

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

```
http://tiny.cloudera.com/dae9a
```

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 a "greedy" by default. Adding a question mark makes the quantifier match as few characters as possible instead, so the pattern `ore.+?a` on this string would match `ore ha`.

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

Boundary Matching Metacharacters

^	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 interepreted 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.