Apache Hadoop –
A course for undergraduates

Homework Labs

with

Professor's Notes

Table of Contents

General Notes on Homework Labs

Students complete homework for this course using the student version of the training Virtual Machine (VM). Cloudera supplies a second VM, the professor's VM, in addition to the student VM. The professor's VM comes complete with solutions to the homework labs.

The professor's VM contains additional project subdirectories with hints and solutions. Subdirectories named src/hints and src/solution provide hints (partial solutions) and full solutions, respectively. The student VM will have src/stubs directories only, no hints or solutions directories. Full solutions can be distributed to students after homework has been submitted. In some cases, a lab may require that the previous lab(s) ran successfully, ensuring that the VM is in the required state. Providing students with the solution to the previous lab, and having them run the solution, will bring the VM to the required state. This should be completed prior to running code for the new lab.

Except for the presence of solutions in the professor VM, the student and professor versions of the training VM are the same. Both VMs run the CentOS 6.3 Linux distribution and come configured with CDH (Cloudera's Distribution, including Apache Hadoop) installed in pseudo-distributed mode. In addition to core Hadoop, the Hadoop ecosystem tools necessary to complete the homework labs are also installed (e.g. Pig, Hive, Flume, etc.). Perl, Python, PHP, and Ruby are installed as well.

Hadoop pseudo-distributed mode is a method of running Hadoop whereby all Hadoop daemons run on the same machine. It is, essentially, a cluster consisting of a single machine. It works just like a larger Hadoop cluster, the key difference (apart from speed, of course!) is that the block replication factor is set to one, since there is only a single DataNode available.

Note: Homework labs are grouped into individual files by lecture number for easy posting of assignments. The same labs appear in this document, but with references to hints and solutions where applicable. The students' homework labs will reference a 'stubs' subdirectory, not 'hints' or 'solutions'. Students will typically complete their coding in the 'stubs' subdirectories.

Getting Started

1. The VM is set to automatically log in as the user `training`. Should you log out at any time, you can log back in as the user `training` with the password `training`.

Working with the Virtual Machine

1. Should you need it, the root password is `training`. You may be prompted for this if, for example, you want to change the keyboard layout. In general, you should not need this password since the `training` user has unlimited sudo privileges.

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

```
$ hadoop fs -put shakespeare  \
/user/training/shakespeare
```

   The dollar sign ($) at the beginning of each line indicates the Linux shell prompt. The actual prompt will include additional information (e.g., `[training@localhost workspace]$`) but this is omitted from these instructions for brevity.

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

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 &
```

Lecture 1 Lab: Using HDFS

Files Used in This Exercise:

Data files (local)

```
~/training_materials/developer/data/shakespeare.tar.gz
~/training_materials/developer/data/access_log.gz
```

In this lab you will begin to get acquainted with the Hadoop tools. You will manipulate files in HDFS, the Hadoop Distributed File System.

Set Up Your Environment

1. Before starting the labs, run the course setup script in a terminal window:

```
$ ~/scripts/developer/training_setup_dev.sh
```

Hadoop

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

Most of your interaction with the system will be through a command-line wrapper called `hadoop`. If you run this program with no arguments, it prints a help message. To try this, run the following command in a terminal window:

```
$ hadoop
```

The `hadoop` command is subdivided into several subsystems. For example, there is a subsystem for working with files in HDFS and another for launching and managing MapReduce processing jobs.

Step 1: Exploring HDFS

The subsystem associated with HDFS in the Hadoop wrapper program is called `FsShell`. This subsystem can be invoked with the command `hadoop fs`.

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

2. In the terminal window, enter:

```
$ hadoop fs
```

You see a help message describing all the commands associated with the `FsShell` subsystem.

3. Enter:

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

This shows you the contents of the root directory in HDFS. There will be multiple entries, one of which is `/user`. Individual users have a "home" directory under this directory, named after their username; your username in this course is `training`, therefore your home directory is `/user/training`.

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

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

You will see your home directory in the directory listing.

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

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

There are no files yet, so the command silently exits. This is different from running `hadoop fs -ls /foo`, which refers to a directory that doesn't exist. In this case, an error message would be displayed.

Note that the directory structure in HDFS has nothing to do with the directory structure of the local filesystem; they are completely separate namespaces.

Step 2: Uploading Files

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

1. Change directories to the local filesystem directory containing the sample data we will be using in the homework labs.

```
$ cd ~/training_materials/developer/data
```

If you perform a regular Linux `ls` command in this directory, you will see a few files, including two named `shakespeare.tar.gz` and `shakespeare-stream.tar.gz`. Both of these contain the complete works of Shakespeare in text format, but with different formats and organizations. For now we will work with `shakespeare.tar.gz`.

2. Unzip `shakespeare.tar.gz` by running:

```
$ tar zxvf shakespeare.tar.gz
```

This creates a directory named `shakespeare/` containing several files on your local filesystem.

3. Insert this directory into HDFS:

```
$ hadoop fs -put shakespeare /user/training/shakespeare
```

This copies the local `shakespeare` directory and its contents into a remote, HDFS directory named `/user/training/shakespeare`.

4. List the contents of your HDFS home directory now:

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

You should see an entry for the `shakespeare` directory.

5. Now try the same `fs -ls` command but without a path argument:

```
$ hadoop fs -ls
```

You should see the same results. If you don't pass a directory name to the `-ls` command, it assumes you mean your home directory, i.e. `/user/training`.

Relative paths

If you pass any relative (non-absolute) paths to `FsShell` commands (or use relative paths in MapReduce programs), they are considered relative to your home directory.

6. We will also need a sample web server log file, which we will put into HDFS for use in future labs. This file is currently compressed using GZip. Rather than extract the file to the local disk and then upload it, we will extract and upload in one step. First, create a directory in HDFS in which to store it:

```
$ hadoop fs -mkdir weblog
```

7. Now, extract and upload the file in one step. The `-c` option to `gunzip` uncompresses to standard output, and the dash (`-`) in the `hadoop fs -put` command takes whatever is being sent to its standard input and places that data in HDFS.

```
$ gunzip -c access_log.gz \
| hadoop fs -put - weblog/access_log
```

8. Run the `hadoop fs -ls` command to verify that the log file is in your HDFS home directory.

9. The access log file is quite large – around 500 MB. Create a smaller version of this file, consisting only of its first 5000 lines, and store the smaller version in HDFS. You can use the smaller version for testing in subsequent labs.

```
$ hadoop fs -mkdir testlog
$ gunzip -c access_log.gz | head -n 5000 \
| hadoop fs -put - testlog/test_access_log
```

Step 3: Viewing and Manipulating Files

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

1. Enter:

```
$ hadoop fs -ls shakespeare
```

This lists the contents of the `/user/training/shakespeare` HDFS directory, which consists of the files `comedies`, `glossary`, `histories`, `poems`, and `tragedies`.

2. The `glossary` file included in the compressed file you began with is not strictly a work of Shakespeare, so let's remove it:

```
$ hadoop fs -rm shakespeare/glossary
```

Note that you *could* leave this file in place if you so wished. If you did, then it would be included in subsequent computations across the works of Shakespeare, and would skew your results slightly. As with many real-world big data problems, you make trade-offs between the labor to purify your input data and the precision of your results.

3. Enter:

```
$ hadoop fs -cat shakespeare/histories | tail -n 50
```

This prints the last 50 lines of *Henry IV, Part 1* to your terminal. This command is handy for viewing the output of MapReduce programs. Very often, an individual output file of a MapReduce program is very large, making it inconvenient to view the entire file in the terminal. For this reason, it's often a good idea to pipe the output of the `fs -cat` command into `head`, `tail`, `more`, or `less`.

4. To download a file to work with on the local filesystem use the `fs -get` command. This command takes two arguments: an HDFS path and a local path. It copies the HDFS contents into the local filesystem:

```
$ hadoop fs -get shakespeare/poems ~/shakepoems.txt
$ less ~/shakepoems.txt
```

Other Commands

There are several other operations available with the `hadoop fs` command to perform most common filesystem manipulations: `mv`, `cp`, `mkdir`, etc.

1. Enter:

```
$ hadoop fs
```

This displays a brief usage report of the commands available within `FsShell`. Try playing around with a few of these commands if you like.

This is the end of the lab.

Lecture 2 Lab: Running a MapReduce Job

Files and Directories Used in this Exercise

Source directory: `~/workspace/wordcount/src/solution`

Files:

`WordCount.java`: A simple MapReduce driver class.

`WordMapper.java`: A mapper class for the job.

`SumReducer.java`: A reducer class for the job.

`wc.jar`: The compiled, assembled WordCount program

In this lab you will compile Java files, create a JAR, and run MapReduce jobs.

In addition to manipulating files in HDFS, the wrapper program `hadoop` is used to launch MapReduce jobs. The code for a job is contained in a compiled JAR file. Hadoop loads the JAR into HDFS and distributes it to the worker nodes, where the individual tasks of the MapReduce job are executed.

One simple example of a MapReduce job is to count the number of occurrences of each word in a file or set of files. In this lab you will compile and submit a MapReduce job to count the number of occurrences of every word in the works of Shakespeare.

Compiling and Submitting a MapReduce Job

1. In a terminal window, change to the lab source directory, and list the contents:

```
$ cd ~/workspace/wordcount/src
$ ls
```

List the files in the `solution` package directory:

```
$ ls solution
```

The package contains the following Java files:

`WordCount.java`: A simple MapReduce driver class.
`WordMapper.java`: A mapper class for the job.
`SumReducer.java`: A reducer class for the job.

Examine these files if you wish, but do not change them. Remain in this directory while you execute the following commands.

2. Before compiling, examine the classpath Hadoop is configured to use:

```
$ hadoop classpath
```

This shows lists the locations where the Hadoop core API classes are installed.

3. Compile the three Java classes:

```
$ javac -classpath `hadoop classpath` solution/*.java
```

Note: in the command above, the quotes around `hadoop classpath` **are backquotes. This runs the** `hadoop classpath` **command and uses its output as part of the** `javac` **command.**

The compiled (`.class`) files are placed in the `solution` directory.

4. Collect your compiled Java files into a JAR file:

```
$ jar cvf wc.jar solution/*.class
```

5. Submit a MapReduce job to Hadoop using your JAR file to count the occurrences of each word in Shakespeare:

```
$ hadoop jar wc.jar solution.WordCount \
shakespeare wordcounts
```

This `hadoop jar` command names the JAR file to use (`wc.jar`), the class whose `main` method should be invoked (`solution.WordCount`), and the HDFS input and output directories to use for the MapReduce job.

Your job reads all the files in your HDFS `shakespeare` directory, and places its output in a new HDFS directory called `wordcounts`.

6. Try running this same command again without any change:

```
$ hadoop jar wc.jar solution.WordCount \
shakespeare wordcounts
```

Your job halts right away with an exception, because Hadoop automatically fails if your job tries to write its output into an existing directory. This is by design; since the result of a MapReduce job may be expensive to reproduce, Hadoop prevents you from accidentally overwriting previously existing files.

7. Review the result of your MapReduce job:

```
$ hadoop fs -ls wordcounts
```

This lists the output files for your job. (Your job ran with only one Reducer, so there should be one file, named `part-`**`r`**`-00000`, along with a `_SUCCESS` file and a `_logs` directory.)

8. View the contents of the output for your job:

```
$ hadoop fs -cat wordcounts/part-r-00000 | less
```

You can page through a few screens to see words and their frequencies in the works of Shakespeare. (The spacebar will scroll the output by one screen; the letter '`q`' will quit the `less` utility.) Note that you could have specified `wordcounts/*` just as well in this command.

Wildcards in HDFS file paths

Take care when using wildcards (e.g. `*`) when specifying HFDS filenames; because of how Linux works, the shell will attempt to expand the wildcard before invoking `hadoop`, and then pass incorrect references to local files instead of HDFS files. You can prevent this by enclosing the wildcarded HDFS filenames in single quotes, e.g. `hadoop fs -cat 'wordcounts/*'`

9. Try running the WordCount job against a single file:

```
$ hadoop jar wc.jar solution.WordCount \
shakespeare/poems pwords
```

When the job completes, inspect the contents of the `pwords` HDFS directory.

10. Clean up the output files produced by your job runs:

```
$ hadoop fs -rm -r wordcounts pwords
```

Stopping MapReduce Jobs

It is important to be able to stop jobs that are already running. This is useful if, for example, you accidentally introduced an infinite loop into your Mapper. An important point to remember is that pressing `^C` to kill the current process (which is displaying the MapReduce job's progress) does **not** actually stop the job itself.

A MapReduce job, once submitted to Hadoop, runs independently of the initiating process, so losing the connection to the initiating process does not kill the job. Instead, you need to tell the Hadoop JobTracker to stop the job.

1. Start another word count job like you did in the previous section:

```
$ hadoop jar wc.jar solution.WordCount shakespeare \
count2
```

2. While this job is running, open another terminal window and enter:

```
$ mapred job -list
```

This lists the job ids of all running jobs. A job id looks something like: `job_200902131742_0002`

3. Copy the job id, and then kill the running job by entering:

```
$ mapred job -kill jobid
```

The JobTracker kills the job, and the program running in the original terminal completes.

This is the end of the lab.

Lecture 3 Lab: Writing a MapReduce Java Program

Projects and Directories Used in this Exercise

Eclipse project: `averagewordlength`

Java files:

`AverageReducer.java` (Reducer)

`LetterMapper.java` (Mapper)

`AvgWordLength.java` (driver)

Test data (HDFS):

`shakespeare`

Exercise directory: `~/workspace/averagewordlength`

In this lab, you will write a MapReduce job that reads any text input and computes the average length of all words that start with each character.

For any text input, the job should report the average length of words that begin with 'a', 'b', and so forth. For example, for input:

```
No now is definitely not the time
```

The output would be:

```
N    2.0
n    3.0
d    10.0
i    2.0
t    3.5
```

(For the initial solution, your program should be case-sensitive as shown in this example.)

The Algorithm

The algorithm for this program is a simple one-pass MapReduce program:

The Mapper

The Mapper receives a line of text for each input value. (Ignore the input key.) For each word in the line, emit the first letter of the word as a key, and the length of the word as a value. For example, for input value:

```
No now is definitely not the time
```

Your Mapper should emit:

```
N    2
n    3
i    2
d    10
n    3
t    3
t    4
```

The Reducer

Thanks to the shuffle and sort phase built in to MapReduce, the Reducer receives the keys in sorted order, and all the values for one key are grouped together. So, for the Mapper output above, the Reducer receives this:

```
N     (2)
d     (10)
i     (2)
n     (3,3)
t     (3,4)
```

The Reducer output should be:

```
N     2.0
d     10.0
i     2.0
n     3.0
t     3.5
```

Step 1: Start Eclipse

There is one Eclipse project for each of the labs that use Java. Using Eclipse will speed up your development time.

1. Be sure you have run the course setup script as instructed earlier in the General Notes section. That script sets up the lab workspace and copies in the Eclipse projects you will use for the remainder of the course.

2. Start Eclipse using the icon on your VM desktop. The projects for this course will appear in the Project Explorer on the left.

Step 2: Write the Program in Java

There are stub files for each of the Java classes for this lab: `LetterMapper.java` (the Mapper), `AverageReducer.java` (the Reducer), and `AvgWordLength.java` (the driver).

If you are using Eclipse, open the stub files (located in the `src/stubs` package) in the `averagewordlength` project. If you prefer to work in the shell, the files are in `~/workspace/averagewordlength/src/stubs`.

You may wish to refer back to the `wordcount` example (in the `wordcount` project in Eclipse or in `~/workspace/wordcount`) as a starting point for your Java code. Here are a few details to help you begin your Java programming:

3. Define the driver

This class should configure and submit your basic job. Among the basic steps here, configure the job with the Mapper class and the Reducer class you will write, and the data types of the intermediate and final keys.

4. Define the Mapper

Note these simple string operations in Java:

```
str.substring(0, 1)  // String : first letter of str
str.length()         // int : length of str
```

5. Define the Reducer

In a single invocation the `reduce()` method receives a string containing one letter (the key) along with an iterable collection of integers (the values), and should emit a single key-value pair: the letter and the average of the integers.

6. Compile your classes and assemble the jar file

To compile and jar, you may either use the command line `javac` command as you did earlier in the "Running a MapReduce Job" lab, or follow the steps below ("Using Eclipse to Compile Your Solution") to use Eclipse.

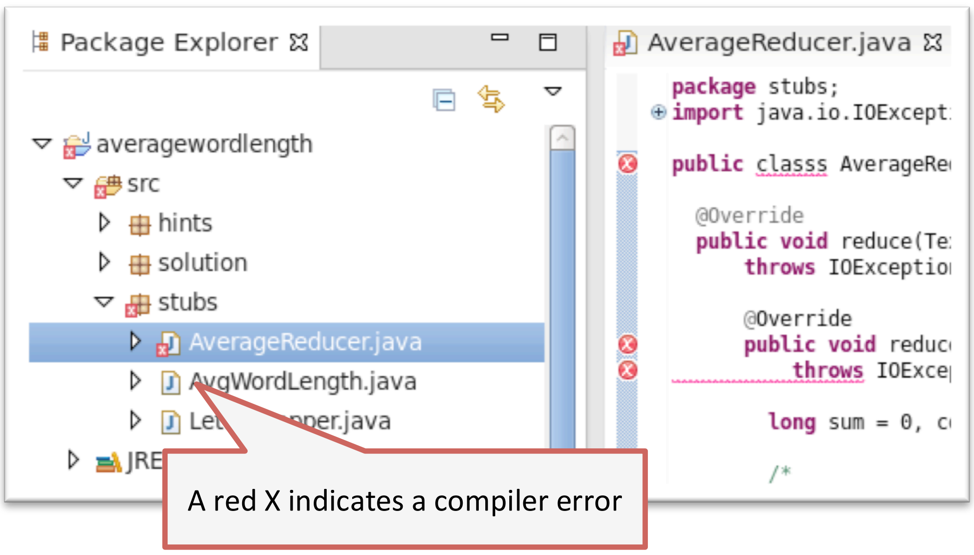
Step 3: Use Eclipse to Compile Your Solution

Follow these steps to use Eclipse to complete this lab.

Note: These same steps will be used for all subsequent labs. The instructions will not be repeated each time, so take note of the steps.

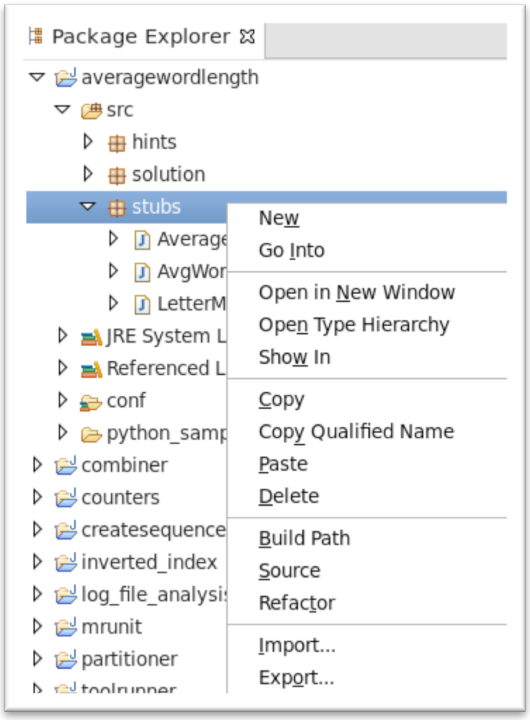
1. Verify that your Java code does not have any compiler errors or warnings.

The Eclipse software in your VM is pre-configured to compile code automatically without performing any explicit steps. Compile errors and warnings appear as red and yellow icons to the left of the code.

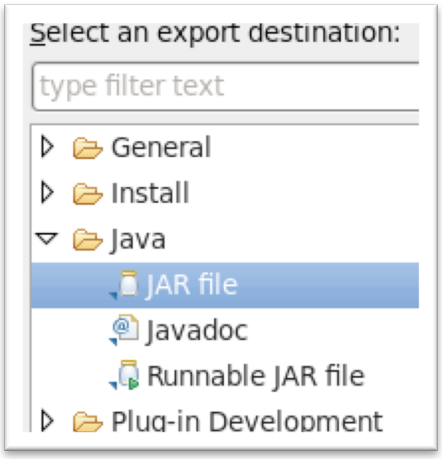


```
package stubs;
import java.io.IOExcept
public classs AverageRe

   @Override
   public void reduce(Te
       throws IOExceptio

      @Override
      public void reduc
          throws IOExce

         long sum = 0, c

         /*
```

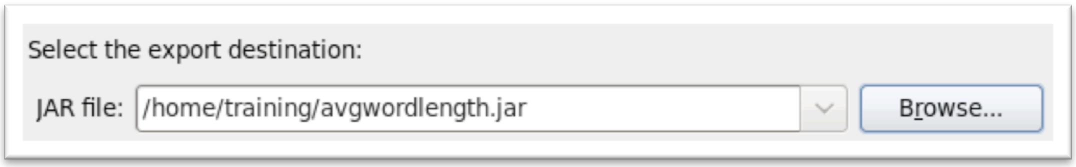
2. In the Package Explorer, open the Eclipse project for the current lab (i.e. `averagewordlength`). Right-click the default package under the `src` entry and select Export.



3. Select **Java > JAR file** from the Export dialog box, then click Next.



4. Specify a location for the JAR file. You can place your JAR files wherever you like, e.g.:



JAR file: /home/training/avgwordlength.jar

Note: For more information about using Eclipse, see the *Eclipse Reference* in Homework_EclipseRef.docx.

Step 3: Test your program

1. In a terminal window, change to the directory where you placed your JAR file. Run the `hadoop jar` command as you did previously in the "Running a MapReduce Job" lab.

```
$ hadoop jar avgwordlength.jar stubs.AvgWordLength \
  shakespeare wordlengths
```

2. List the results:

```
$ hadoop fs -ls wordlengths
```

A single reducer output file should be listed.

3. Review the results:

```
$ hadoop fs -cat wordlengths/*
```

The file should list all the numbers and letters in the data set, and the average length of the words starting with them, e.g.:

```
1      1.02
2      1.0588235294117647
3      1.0
4      1.5
5      1.5
6      1.5
7      1.0
8      1.5
9      1.0
A      3.891394576646375
B      5.139302507836991
C      6.629694233531706
…
```

This example uses the entire Shakespeare dataset for your input; you can also try it with just one of the files in the dataset, or with your own test data.

This is the end of the lab.

Lecture 3 Lab: More Practice with MapReduce Java Programs

Files and Directories Used in this Exercise

Eclipse project: `log_file_analysis`

Java files:

`SumReducer.java` – the Reducer

`LogFileMapper.java` – the Mapper

`ProcessLogs.java` – the driver class

Test data (HDFS):

`weblog` (full version)

`testlog` (test sample set)

Exercise directory: `~/workspace/log_file_analysis`

In this lab, you will analyze a log file from a web server to count the number of hits made from each unique IP address.

Your task is to count the number of hits made from each IP address in the sample (anonymized) web server log file that you uploaded to the `/user/training/weblog` directory in HDFS when you completed the "Using HDFS" lab.

In the `log_file_analysis` directory, you will find stubs for the Mapper and Driver.

1. Using the stub files in the `log_file_analysis` project directory, write Mapper and Driver code to count the number of hits made from each IP address in the access log file. Your final result should be a file in HDFS containing each IP address, and the count of log hits from that address. **Note: The Reducer for this lab performs the exact same function as the one in the WordCount program you ran earlier. You can reuse that code or you can write your own if you prefer.**

2. Build your application jar file following the steps in the previous lab.

3. Test your code using the sample log data in the `/user/training/weblog` directory. **Note**: You may wish to test your code against the smaller version of the access log you created in a prior lab (located in the `/user/training/testlog` HDFS directory) before you run your code against the full log which can be quite time consuming.

This is the end of the lab.

Lecture 3 Lab: Writing a MapReduce Streaming Program

Files and Directories Used in this Exercise

Project directory: `~/workspace/averagewordlength`

Test data (HDFS):
`shakespeare`

In this lab you will repeat the same task as in the previous lab: writing a program to calculate average word lengths for letters. However, you will write this as a streaming program using a scripting language of your choice rather than using Java.

Your virtual machine has Perl, Python, PHP, and Ruby installed, so you can choose any of these—or even shell scripting—to develop a Streaming solution.

For your Hadoop Streaming program you will not use Eclipse. Launch a text editor to write your Mapper script and your Reducer script. Here are some notes about solving the problem in Hadoop Streaming:

1. The Mapper Script

The Mapper will receive lines of text on `stdin`. Find the words in the lines to produce the intermediate output, and emit intermediate (key, value) pairs by writing strings of the form:

```
key <tab> value <newline>
```

These strings should be written to `stdout`.

2. The Reducer Script

For the reducer, multiple values with the same key are sent to your script on `stdin` as successive lines of input. Each line contains a key, a tab, a value, and a newline. All lines with the same key are sent one after another, possibly followed by lines with a different key, until the reducing input is complete. For example, the reduce script may receive the following:

```
t      3
t      4
w      4
w      6
```

For this input, emit the following to `stdout`:

```
t      3.5
w      5.0
```

Observe that the reducer receives a key with each input line, and must "notice" when the key changes on a subsequent line (or when the input is finished) to know when the values for a given key have been exhausted. This is different than the Java version you worked on in the previous lab.

3. Run the streaming program:

```
$ hadoop jar /usr/lib/hadoop-0.20-mapreduce/\
contrib/streaming/hadoop-streaming*.jar \
-input inputDir -output outputDir \
-file pathToMapScript -file pathToReduceScript \
-mapper mapBasename -reducer reduceBasename
```

(Remember, you may need to delete any previous output before running your program by issuing: `hadoop fs -rm -r dataToDelete`.)

4. Review the output in the HDFS directory you specified (`outputDir`).

Professor's Note~

The Perl example is in: `~/workspace/wordcount/perl_solution`

Professor's Note~

Solution in Python

You can find a working solution to this lab written in Python in the directory `~/workspace/averagewordlength/python_sample_solution.`

To run the solution, change directory to `~/workspace/averagewordlength` and run this command:

```
$ hadoop jar /usr/lib/hadoop-0.20-mapreduce\
/contrib/streaming/hadoop-streaming*.jar \
-input shakespeare -output avgwordstreaming \
-file python_sample_solution/mapper.py \
-file python_sample_solution/reducer.py \
-mapper mapper.py -reducer reducer.py
```

This is the end of the lab.

Lecture 3 Lab: Writing Unit Tests with the MRUnit Framework

Projects Used in this Exercise

Eclipse project: `mrunit`

Java files:

`SumReducer.java` (Reducer from WordCount)

`WordMapper.java` (Mapper from WordCount)

`TestWordCount.java` (Test Driver)

In this Exercise, you will write Unit Tests for the WordCount code.

1. Launch Eclipse (if necessary) and expand the `mrunit` folder.

2. Examine the `TestWordCount.java` file in the `mrunit` project `stubs` package. Notice that three tests have been created, one each for the Mapper, Reducer, and the entire MapReduce flow. Currently, all three tests simply fail.

3. Run the tests by right-clicking on `TestWordCount.java` in the Package Explorer panel and choosing **Run As > JUnit Test**.

4. Observe the failure. Results in the JUnit tab (next to the Package Explorer tab) should indicate that three tests ran with three failures.

5. Now implement the three tests.

6. Run the tests again. Results in the JUnit tab should indicate that three tests ran with no failures.

7. When you are done, close the JUnit tab.

This is the end of the lab.

Lecture 4 Lab: Using ToolRunner and Passing Parameters

Files and Directories Used in this Exercise

Eclipse project: `toolrunner`

Java files:

`AverageReducer.java` (Reducer from AverageWordLength)

`LetterMapper.java` (Mapper from AverageWordLength)

`AvgWordLength.java` (driver from AverageWordLength)

Exercise directory: `~/workspace/toolrunner`

In this Exercise, you will implement a driver using ToolRunner.

Follow the steps below to start with the Average Word Length program you wrote in an earlier lab, and modify the driver to use ToolRunner. Then modify the Mapper to reference a Boolean parameter called `caseSensitive`; if true, the mapper should treat upper and lower case letters as different; if false or unset, all letters should be converted to lower case.

Modify the Average Word Length Driver to use Toolrunner

1. Copy the Reducer, Mapper and driver code you completed in the "Writing Java MapReduce Programs" lab earlier, in the `averagewordlength` project.

> **Copying Source Files**
>
> You can use Eclipse to copy a Java source file from one project or package to another by right-clicking on the file and selecting Copy, then right-clicking the new package and selecting Paste. If the packages have different names (e.g. if you copy from `averagewordlength.solution` to `toolrunner.stubs`), Eclipse will automatically change the `package` directive at the top of the file. If you copy the file using a file browser or the shell, you will have to do that manually.

2. Modify the `AvgWordLength` driver to use ToolRunner. Refer to the slides for details.

    a. Implement the `run` method

    b. Modify `main` to call `run`

3. Jar your solution and test it before continuing; it should continue to function exactly as it did before. Refer to the *Writing a Java MapReduce Program* lab for how to assemble and test if you need a reminder.

Modify the Mapper to use a configuration parameter

4. Modify the `LetterMapper` class to

    a. Override the `setup` method to get the value of a configuration parameter called `caseSensitive`, and use it to set a member variable indicating whether to do case sensitive or case insensitive processing.

    b. In the `map` method, choose whether to do case sensitive processing (leave the letters as-is), or insensitive processing (convert all letters to lower-case) based on that variable.

Pass a parameter programmatically

5. Modify the driver's `run` method to set a Boolean configuration parameter called `caseSensitive`. (Hint: Use the `Configuration.setBoolean` method.)

6. Test your code twice, once passing `false` and once passing `true`. When set to true, your final output should have both upper and lower case letters; when false, it should have only lower case letters.

   Hint: Remember to rebuild your Jar file to test changes to your code.

Pass a parameter as a runtime parameter

7. Comment out the code that sets the parameter programmatically. (Eclipse hint: Select the code to comment and then select Source > Toggle Comment). Test again, this time passing the parameter value using `-D` on the Hadoop command line, e.g.:

```
$ hadoop jar toolrunner.jar stubs.AvgWordLength \
-DcaseSensitive=true shakespeare toolrunnerout
```

8. Test passing both `true` and `false` to confirm the parameter works correctly.

This is the end of the lab.

Lecture 4 Lab: Using a Combiner

Files and Directories Used in this Exercise

Eclipse project: `combiner`

Java files:

`WordCountDriver.java` (Driver from WordCount)

`WordMapper.java` (Mapper from WordCount)

`SumReducer.java` (Reducer from WordCount)

Exercise directory: `~/workspace/combiner`

In this lab, you will add a Combiner to the WordCount program to reduce the amount of intermediate data sent from the Mapper to the Reducer.

Because summing is associative and commutative, the same class can be used for both the Reducer and the Combiner.

Implement a Combiner

1. Copy `WordMapper.java` and `SumReducer.java` from the `wordcount` project to the `combiner` project.

2. Modify the `WordCountDriver.java` code to add a Combiner for the WordCount program.

3. Assemble and test your solution. (The output should remain identical to the WordCount application without a combiner.)

This is the end of the lab.

Lecture 5 Lab: Testing with LocalJobRunner

Files and Directories Used in this Exercise

Eclipse project: `toolrunner`

Test data (local):

`~/training_materials/developer/data/shakespeare`

Exercise directory: `~/workspace/toolrunner`

In this lab, you will practice running a job locally for debugging and testing purposes.

In the "Using ToolRunner and Passing Parameters" lab, you modified the Average Word Length program to use ToolRunner. This makes it simple to set job configuration properties on the command line.

Run the Average Word Length program using LocalJobRunner on the command line

1. Run the Average Word Length program again. Specify `–jt=local` to run the job locally instead of submitting to the cluster, and `–fs=file:///` to use the local file system instead of HDFS. Your input and output files should refer to local files rather than HDFS files.

   Note: Use the program you completed in the ToolRunner lab.

```
$ hadoop jar toolrunner.jar stubs.AvgWordLength \
  -fs=file:/// -jt=local \
  ~/training_materials/developer/data/shakespeare \
  localout
```

2. Review the job output in the local output folder you specified.

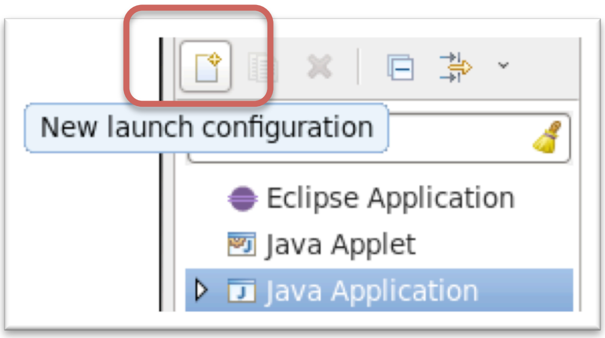
Optional: Run the Average Word Length program using LocalJobRunner in Eclipse

1. In Eclipse, locate the toolrunner project in the Package Explorer. Open the `stubs` package.

2. Right click on the driver class (`AvgWordLength`) and select **Run As > Run Configurations…**

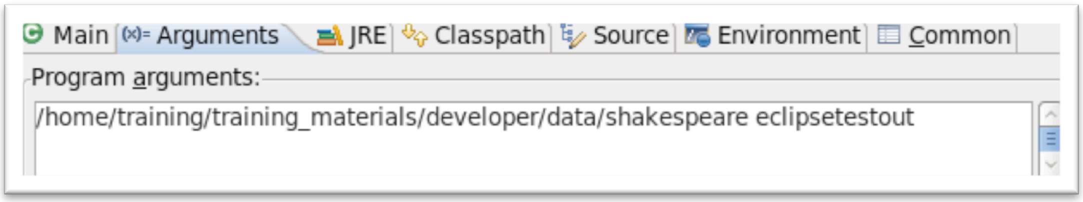
3. Ensure that **Java Application** is selected in the run types listed in the left pane.

4. In the Run Configuration dialog, click the **New launch configuration** button:



5. On the Main tab, confirm that the Project and Main class are set correctly for your project, e.g. Project: toolrunner   and Main class: stubs.AvgWordLength

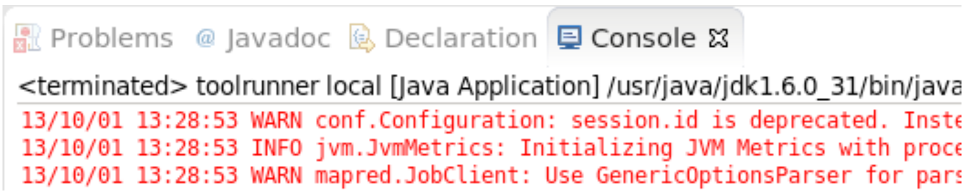
6. Select the Arguments tab and enter the input and output folders. (These are local, not HDFS folders, and are relative to the run configuration's working folder, which by default is the project folder in the Eclipse workspace: e.g. `~/workspace/toolrunner`.)



Program arguments:

```
/home/training/training_materials/developer/data/shakespeare eclipsetestout
```

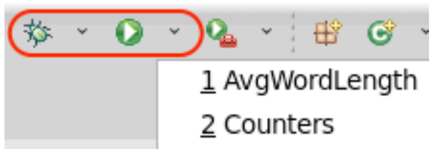
7. Click the **Run** button. The program will run locally with the output displayed in the Eclipse console window.



```
<terminated> toolrunner local [Java Application] /usr/java/jdk1.6.0_31/bin/java
13/10/01 13:28:53 WARN conf.Configuration: session.id is deprecated. Inste
13/10/01 13:28:53 INFO jvm.JvmMetrics: Initializing JVM Metrics with proce
13/10/01 13:28:53 WARN mapred.JobClient: Use GenericOptionsParser for pars
```

8. Review the job output in the local output folder you specified.

Note: You can re-run any previous configurations using the Run or Debug history buttons on the Eclipse tool bar.



This is the end of the lab.

Lecture 5 Lab: Logging

Files and Directories Used in this Exercise

Eclipse project: `logging`

Java files:

`AverageReducer.java` (Reducer from ToolRunner)

`LetterMapper.java` (Mapper from ToolRunner)

`AvgWordLength.java` (driver from ToolRunner)

Test data (HDFS):

`shakespeare`

Exercise directory: `~/workspace/logging`

In this lab, you will practice using log4j with MapReduce.

Modify the Average Word Length program you built in the *Using ToolRunner and Passing Parameters* lab so that the Mapper logs a debug message indicating whether it is comparing with or without case sensitivity.

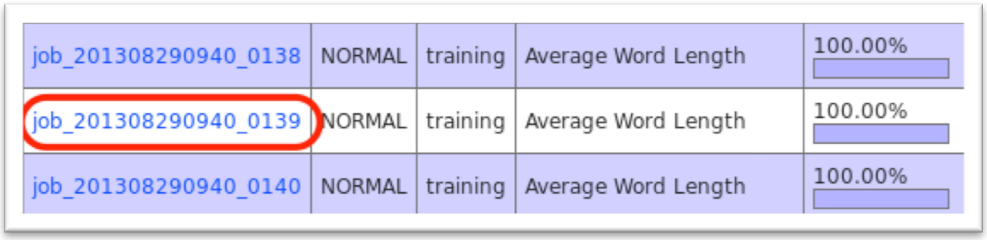
Enable Mapper Logging for the Job

1. Before adding additional logging messages, try re-running the toolrunner lab solution with Mapper debug logging enabled by adding
   `-Dmapred.map.child.log.level=DEBUG`
   to the command line. E.g.

```
$ hadoop jar toolrunner.jar stubs.AvgWordLength \
-Dmapred.map.child.log.level=DEBUG shakespeare outdir
```

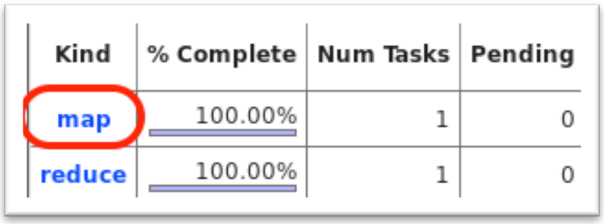
2. Take note of the Job ID in the terminal window or by using the `maprep job` command.

3. When the job is complete, view the logs. In a browser on your VM, visit the Job Tracker UI: `http://localhost:50030/jobtracker.jsp`. Find the job you just ran in the Completed Jobs list and click its Job ID. E.g.:



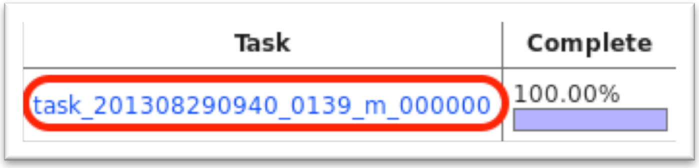
job_201308290940_0138	NORMAL	training	Average Word Length	100.00%
job_201308290940_0139	NORMAL	training	Average Word Length	100.00%
job_201308290940_0140	NORMAL	training	Average Word Length	100.00%

4. In the task summary, click map to view the map tasks.



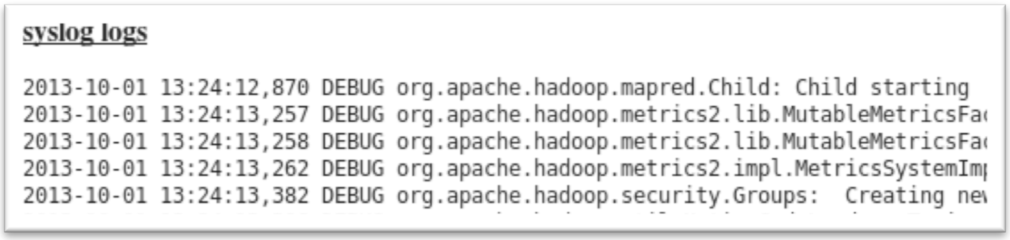
Kind	% Complete	Num Tasks	Pending
map	100.00%	1	0
reduce	100.00%	1	0

5. In the list of tasks, click on the map task to view the details of that task.



Task	Complete
task_201308290940_0139_m_000000	100.00%

6. Under Task Logs, click "All". The logs should include both INFO and DEBUG messages. E.g.:



syslog logs

```
2013-10-01 13:24:12,870 DEBUG org.apache.hadoop.mapred.Child: Child starting
2013-10-01 13:24:13,257 DEBUG org.apache.hadoop.metrics2.lib.MutableMetricsFa
2013-10-01 13:24:13,258 DEBUG org.apache.hadoop.metrics2.lib.MutableMetricsFa
2013-10-01 13:24:13,262 DEBUG org.apache.hadoop.metrics2.impl.MetricsSystemIm
2013-10-01 13:24:13,382 DEBUG org.apache.hadoop.security.Groups:  Creating ne
```

Add Debug Logging Output to the Mapper

7. Copy the code from the `toolrunner` project to the `logging` project `stubs` package. (Use your solution from the ToolRunner lab.)

8. Use log4j to output a debug log message indicating whether the Mapper is doing case sensitive or insensitive mapping.

Build and Test Your Code

9. Following the earlier steps, test your code with Mapper debug logging enabled. View the map task logs in the Job Tracker UI to confirm that your message is included in the log. (Hint: Search for LetterMapper in the page to find your message.)

10. Optional: Try running map logging set to INFO (the default) or WARN instead of DEBUG and compare the log output.

This is the end of the lab.

Lecture 5 Lab: Using Counters and a Map-Only Job

Files and Directories Used in this Exercise

Eclipse project: `counters`

Java files:

`ImageCounter.java` (driver)

`ImageCounterMapper.java` (Mapper)

Test data (HDFS):

`weblog` (full web server access log)

`testlog` (partial data set for testing)

Exercise directory: `~/workspace/counters`

In this lab you will create a Map-only MapReduce job.

Your application will process a web server's access log to count the number of times gifs, jpegs, and other resources have been retrieved. Your job will report three figures: number of gif requests, number of jpeg requests, and number of other requests.

Hints

1. You should use a Map-only MapReduce job, by setting the number of Reducers to 0 in the driver code.

2. For input data, use the Web access log file that you uploaded to the HDFS `/user/training/weblog` directory in the "Using HDFS" lab.

Note: Test your code against the smaller version of the access log in the `/user/training/testlog` directory before you run your code against the full log in the `/user/training/weblog` directory.

3. Use a counter group such as `ImageCounter`, with names `gif`, `jpeg` and `other`.

4. In your driver code, retrieve the values of the counters after the job has completed and report them using `System.out.println`.

5. The output folder on HDFS will contain Mapper output files which are empty, because the Mappers did not write any data.

This is the end of the lab.

Lecture 6 Lab: Writing a Partitioner

Files and Directories Used in this Exercise

Eclipse project: `partitioner`

Java files:

`MonthPartitioner.java` (Partitioner)

`ProcessLogs.java` (driver)

`CountReducer.java` (Reducer)

`LogMonthMapper.java` (Mapper)

Test data (HDFS):

`weblog` (full web server access log)

`testlog` (partial data set for testing)

Exercise directory: `~/workspace/partitioner`

In this Exercise, you will write a MapReduce job with multiple Reducers, and create a Partitioner to determine which Reducer each piece of Mapper output is sent to.

The Problem

In the “More Practice with Writing MapReduce Java Programs” lab you did previously, you built the code in `log_file_analysis` project. That program counted the number of hits for each different IP address in a web log file. The final output was a file containing a list of IP addresses, and the number of hits from that address.

This time, you will perform a similar task, but the final output should consist of 12 files, one each for each month of the year: January, February, and so on. Each file will contain a list of IP addresses, and the number of hits from that address *in that month*.

We will accomplish this by having 12 Reducers, each of which is responsible for processing the data for a particular month. Reducer 0 processes January hits, Reducer 1 processes February hits, and so on.

Note: We are actually breaking the standard MapReduce paradigm here, which says that all the values from a particular key will go to the same Reducer. In this example, which is a very common pattern when analyzing log files, values from the same key (the IP address) will go to multiple Reducers, based on the month portion of the line.

Write the Mapper

1. Starting with the `LogMonthMapper.java` stub file, write a Mapper that maps a log file output line to an IP/month pair. The map method will be similar to that in the `LogFileMapper` class in the `log_file_analysis` project, so you may wish to start by copying that code.

2. The Mapper should emit a Text key (the IP address) and Text value (the month). E.g.:

Input: `96.7.4.14 - - [24/Apr/2011:04:20:11 -0400] "GET /cat.jpg HTTP/1.1" 200 12433`

Output key: `96.7.4.14`

Output value: `Apr`

Hint: In the Mapper, you may use a regular expression to parse to log file data if you are familiar with regex processing (see file Homework_RegexRef.docx for reference).

Remember that the log file may contain unexpected data – that is, lines that do not conform to the expected format. Be sure that your code copes with such lines.

Write the Partitioner

3. Modify the `MonthPartitioner.java` stub file to create a Partitioner that sends the (key, value) pair to the correct Reducer based on the month. Remember that the Partitioner receives both the key and value, so you can inspect the value to determine which Reducer to choose.

Modify the Driver

4. Modify your driver code to specify that you want 12 Reducers.

5. Configure your job to use your custom Partitioner.

Test your Solution

6. Build and test your code. Your output directory should contain 12 files named `part-r-000`*`xx`*. Each file should contain IP address and number of hits for month *xx*.

Hints:

- Write unit tests for your Partitioner!
- You may wish to test your code against the smaller version of the access log in the `/user/training/testlog` directory before you run your code against the full log in the `/user/training/weblog` directory. However, note that the test data may not include all months, so some result files will be empty.

This is the end of the lab.

Lecture 6 Lab: Implementing a Custom WritableComparable

Files and Directories Used in this Exercise

Eclipse project: `writables`

Java files:

`StringPairWritable` – implements a `WritableComparable type`

`StringPairMapper` – Mapper for test job

`StringPairTestDriver` – Driver for test job

Data file:

`~/training_materials/developer/data/nameyeartestdata` (small set of data for the test job)

Exercise directory: `~/workspace/writables`

In this lab, you will create a custom WritableComparable type that holds two strings.

Test the new type by creating a simple program that reads a list of names (first and last) and counts the number of occurrences of each name.

The mapper should accepts lines in the form:

```
lastname firstname other data
```

The goal is to count the number of times a lastname/firstname pair occur within the dataset. For example, for input:

```
Smith Joe 1963-08-12 Poughkeepsie, NY
Smith Joe 1832-01-20 Sacramento, CA
Murphy Alice 2004-06-02 Berlin, MA
```

We want to output:

```
(Smith,Joe)     2
(Murphy,Alice)  1
```

Note: You will use your custom WritableComparable type in a future lab, so make sure it is working with the test job now.

StringPairWritable

You need to implement a WritableComparable object that holds the two strings. The stub provides an empty constructor for serialization, a standard constructor that will be given two strings, a `toString` method, and the generated `hashCode` and `equals` methods. You will need to implement the `readFields`, `write`, and `compareTo` methods required by WritableComparables.

Note that Eclipse automatically generated the `hashCode` and `equals` methods in the stub file. You can generate these two methods in Eclipse by right-clicking in the source code and choosing 'Source' > 'Generate hashCode() and equals()'.

Name Count Test Job

The test job requires a Reducer that sums the number of occurrences of each key. This is the same function that the SumReducer used previously in wordcount, except that SumReducer expects Text keys, whereas the reducer for this job will get StringPairWritable keys. You may either re-write SumReducer to accommodate other types of keys, or you can use the LongSumReducer Hadoop library class, which does exactly the same thing.

You can use the simple test data in `~/training_materials/developer/data/nameyeartestdata` to make sure your new type works as expected.

You may test your code using local job runner or by submitting a Hadoop job to the (pseudo-)cluster as usual. If you submit the job to the cluster, note that you will need to copy your test data to HDFS first.

This is the end of the lab.

Lecture 6 Lab: Using SequenceFiles and File Compression

Files and Directories Used in this Exercise

Eclipse project: `createsequencefile`

Java files:

`CreateSequenceFile.java` (a driver that converts a text file to a sequence file)

`ReadCompressedSequenceFile.java` (a driver that converts a compressed sequence file to text)

Test data (HDFS):

`weblog` (full web server access log)

Exercise directory: `~/workspace/createsequencefile`

In this lab you will practice reading and writing uncompressed and compressed SequenceFiles.

First, you will develop a MapReduce application to convert text data to a SequenceFile. Then you will modify the application to compress the SequenceFile using Snappy file compression.

When creating the SequenceFile, use the full access log file for input data. (You uploaded the access log file to the HDFS `/user/training/weblog` directory when you performed the "Using HDFS" lab.)

After you have created the compressed SequenceFile, you will write a second MapReduce application to read the compressed SequenceFile and write a text file that contains the original log file text.

Write a MapReduce program to create sequence files from text files

1. Determine the number of HDFS blocks occupied by the access log file:

   a. In a browser window, start the Name Node Web UI. The URL is `http://localhost:50070`

   b. Click "Browse the filesystem."

   c. Navigate to the `/user/training/weblog/access_log` file.

   d. Scroll down to the bottom of the page. The total number of blocks occupied by the access log file appears in the browser window.

2. Complete the stub file in the `createsequencefile` project to read the access log file and create a SequenceFile. Records emitted to the SequenceFile can have any key you like, but the values should match the text in the access log file. (Hint: You can use Map-only job using the default Mapper, which simply emits the data passed to it.)

   Note: If you specify an output key type other than `LongWritable`, you must call `job.setOutputKeyClass` – *not* `job.setMapOutputKeyClass`. If you specify an output value type other than `Text`, you must call `job.setOutputValueClass` – *not* `job.setMapOutputValueClass`.

3. Build and test your solution so far. Use the access log as input data, and specify the `uncompressedsf` directory for output.

4. Examine the initial portion of the output SequenceFile using the following command:

```
$ hadoop fs -cat uncompressedsf/part-m-00000 | less
```

   Some of the data in the SequenceFile is unreadable, but parts of the SequenceFile should be recognizable:

   - The string `SEQ`, which appears at the beginning of a SequenceFile
   - The Java classes for the keys and values
   - Text from the access log file

5. Verify that the number of files created by the job is equivalent to the number of blocks required to store the uncompressed SequenceFile.

Compress the Output

6. Modify your MapReduce job to compress the output SequenceFile. Add statements to your driver to configure the output as follows:

- Compress the output file.
- Use block compression.
- Use the Snappy compression codec.

7. Compile the code and run your modified MapReduce job. For the MapReduce output, specify the `compressedsf` directory.

8. Examine the first portion of the output SequenceFile. Notice the differences between the uncompressed and compressed SequenceFiles:

- The compressed SequenceFile specifies the `org.apache.hadoop.io.compress.SnappyCodec` compression codec in its header.
- You cannot read the log file text in the compressed file.

9. Compare the file sizes of the uncompressed and compressed SequenceFiles in the `uncompressedsf` and `compressedsf` directories. The compressed SequenceFiles should be smaller.

Write another MapReduce program to uncompress the files

10. Starting with the provided stub file, write a second MapReduce program to read the compressed log file and write a text file. This text file should have the same text data as the log file, plus keys. The keys can contain any values you like.

11. Compile the code and run your MapReduce job.

For the MapReduce input, specify the `compressedsf` directory in which you created the compressed SequenceFile in the previous section.

For the MapReduce output, specify the `compressedsftotext` directory.

12. Examine the first portion of the output in the `compressedsftotext` directory.

You should be able to read the textual log file entries.

Optional: Use command line options to control compression

13. If you used ToolRunner for your driver, you can control compression using command line arguments. Try commenting out the code in your driver where you call. Then test setting the `mapred.output.compressed` option on the command line, e.g.:

```
$ hadoop jar sequence.jar \
 stubs.CreateUncompressedSequenceFile \
 -Dmapred.output.compressed=true \
 weblog outdir
```

14. Review the output to confirm the files are compressed.

This is the end of the lab.

Lecture 7 Lab: Creating an Inverted Index

Files and Directories Used in this Exercise

Eclipse project: `inverted_index`

Java files:

`IndexMapper.java` (Mapper)

`IndexReducer.java` (Reducer)

`InvertedIndex.java` (Driver)

Data files:

`~/training_materials/developer/data/invertedIndexInput.tgz`

Exercise directory: `~/workspace/inverted_index`

In this lab, you will write a MapReduce job that produces an inverted index.

For this lab you will use an alternate input, provided in the file `invertedIndexInput.tgz`. When decompressed, this archive contains a directory of files; each is a Shakespeare play formatted as follows:

```
0       HAMLET
1
2
3       DRAMATIS PERSONAE
4
5
6       CLAUDIUS        king of Denmark. (KING CLAUDIUS:)
7
8       HAMLET  son to the late, and nephew to the present king.
9
10      POLONIUS        lord chamberlain. (LORD POLONIUS:)
```

```
...
```

Each line contains:

Line number
separator: a tab character
value: the line of text

This format can be read directly using the `KeyValueTextInputFormat` class provided in the Hadoop API. This input format presents each line as one record to your Mapper, with the part before the tab character as the key, and the part after the tab as the value.

Given a body of text in this form, your indexer should produce an index of all the words in the text. For each word, the index should have a list of all the locations where the word appears. For example, for the word 'honeysuckle' your output should look like this:

```
honeysuckle     2kinghenryiv@1038,midsummernightsdream@2175,...
```

The index should contain such an entry for every word in the text.

Prepare the Input Data

1. Extract the `invertedIndexInput` directory and upload to HDFS:

```
$ cd ~/training_materials/developer/data
$ tar zxvf invertedIndexInput.tgz
$ hadoop fs -put invertedIndexInput invertedIndexInput
```

Define the MapReduce Solution

Remember that for this program you use a special input format to suit the form of your data, so your driver class will include a line like:

```
job.setInputFormatClass(KeyValueTextInputFormat.class);
```

Don't forget to import this class for your use.

Retrieving the File Name

Note that the lab requires you to retrieve the file name - since that is the name of the play. The Context object can be used to retrieve the name of the file like this:

```
FileSplit fileSplit = (FileSplit) context.getInputSplit();
Path path = fileSplit.getPath();
String fileName = path.getName();
```

Build and Test Your Solution

Test against the `invertedIndexInput` data you loaded above.

Hints

You may like to complete this lab without reading any further, or you may find the following hints about the algorithm helpful.

The Mapper

Your Mapper should take as input a key and a line of words, and emit as intermediate values each word as key, and the key as value.

For example, the line of input from the file 'hamlet':

```
282 Have heaven and earth together
```

produces intermediate output:

```
Have      hamlet@282
heaven    hamlet@282
and       hamlet@282
earth     hamlet@282
together  hamlet@282
```

The Reducer

Your Reducer simply aggregates the values presented to it for the same key, into one value. Use a separator like ',' between the values listed.

This is the end of the lab.

Lecture 7 Lab: Calculating Word Co-Occurrence

Files and Directories Used in this Exercise

Eclipse project: `word_co-occurrence`

Java files:

`WordCoMapper.java` (Mapper)

`SumReducer.java` (Reducer from WordCount)

`WordCo.java` (Driver)

Test directory (HDFS):

`shakespeare`

Exercise directory: `~/workspace/word_co-occurence`

In this lab, you will write an application that counts the number of times words appear next to each other.

Test your application using the files in the `shakespeare` folder you previously copied into HDFS in the "Using HDFS" lab.

Note that this implementation is a specialization of Word Co-Occurrence as we describe it in the notes; in this case **we are only interested in pairs of words which appear directly next to each other.**

1. Change directories to the `word_co-occurrence` directory within the `labs` directory.

2. Complete the Driver and Mapper stub files; you can use the standard SumReducer from the WordCount project as your Reducer. Your Mapper's intermediate output should be in the form of a Text object as the key, and an IntWritable as the value; the key will be `word1,word2`, and the value will be `1`.

Extra Credit

If you have extra time, please complete these additional challenges:

Challenge 1: Use the `StringPairWritable` key type from the "Implementing a Custom WritableComparable" lab. Copy your completed solution (from the `writables` project) into the current project.

Challenge 2: Write a second MapReduce job to sort the output from the first job so that the list of pairs of words appears in ascending frequency.

Challenge 3: Sort by descending frequency instead (sort that the most frequently occurring word pairs are first in the output.) Hint: You will need to extend `org.apache.hadoop.io.LongWritable.Comparator`.

This is the end of the lab.

Lecture 8 Lab: Importing Data with Sqoop

In this lab you will import data from a relational database using Sqoop. The data you load here will be used subsequent labs.

Consider the MySQL database `movielens`, derived from the MovieLens project from University of Minnesota. (See note at the end of this lab.) The database consists of several related tables, but we will import only two of these: `movie`, which contains about 3,900 movies; and `movierating`, which has about 1,000,000 ratings of those movies.

Review the Database Tables

First, review the database tables to be loaded into Hadoop.

1. Log on to MySQL:

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

2. Review the structure and contents of the `movie` table:

```
mysql> DESCRIBE movie;
. . .
mysql> SELECT * FROM movie LIMIT 5;
```

3. Note the column names for the table:

____

4. Review the structure and contents of the `movierating` table:

```
mysql> DESCRIBE movierating;
…
mysql> SELECT * FROM movierating LIMIT 5;
```

5. Note these column names:

____

6. Exit mysql:

```
mysql> quit
```

Import with Sqoop

You invoke Sqoop on the command line to perform several commands. With it you can connect to your database server to list the databases (schemas) to which you have access, and list the tables available for loading. For database access, you provide a connect string to identify the server, and - if required - your username and password.

1. Show the commands available in Sqoop:

```
$ sqoop help
```

2. List the databases (schemas) in your database server:

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

(Note: Instead of entering `--password training` on your command line, you may prefer to enter `-P`, and let Sqoop prompt you for the password, which is then not visible when you type it.)

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

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

4. Import the `movie` table into Hadoop:

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

5. Verify that the command has worked.

```
$ hadoop fs -ls movie
$ hadoop fs -tail movie/part-m-00000
```

6. Import the `movierating` table into Hadoop.

Repeat the last two steps, but for the `movierating` table.

This is the end of the lab.

Note:

This lab uses the MovieLens data set, or subsets thereof. This data is freely available for academic purposes, and is used and distributed by Cloudera with the express permission of the UMN GroupLens Research Group. If you would like to use this data for your own research purposes, you are free to do so, as long as you cite the GroupLens Research Group in any resulting publications. If you would like to use this data for commercial purposes, you must obtain explicit permission. You may find the full dataset, as well as detailed license terms, at http://www.grouplens.org/node/73

Lecture 8 Lab: Running an Oozie Workflow

Files and Directories Used in this Exercise

Exercise directory: `~/workspace/oozie_labs`

Oozie job folders:

`lab1-java-mapreduce`

`lab2-sort-wordcount`

In this lab, you will inspect and run Oozie workflows.

1. Start the Oozie server

```
$ sudo /etc/init.d/oozie start
```

2. Change directories to the lab directory:

```
$ cd ~/workspace/oozie-labs
```

3. Inspect the contents of the `job.properties` and `workflow.xml` files in the `lab1-java-mapreduce/job` folder. You will see that this is the standard WordCount job.

   In the `job.properties` file, take note of the job's base directory (`lab1-java-mapreduce`), and the input and output directories relative to that. (These are HDFS directories.)

4. We have provided a simple shell script to submit the Oozie workflow. Inspect the `run.sh` script and then run:

```
$ ./run.sh lab1-java-mapreduce
```

   Notice that Oozie returns a job identification number.

5. Inspect the progress of the job:

```
$ oozie job -oozie http://localhost:11000/oozie \
-info job_id
```

6. When the job has completed, review the job output directory in HDFS to confirm that the output has been produced as expected.

7. Repeat the above procedure for `lab2-sort-wordcount`. Notice when you inspect `workflow.xml` that this workflow includes two MapReduce jobs which run one after the other, in which the output of the first is the input for the second. When you inspect the output in HDFS you will see that the second job sorts the output of the first job into descending numerical order.

This is the end of the lab.

Lecture 8 Bonus Lab: Exploring a Secondary Sort Example

Files and Directories Used in this Exercise

Eclipse project: `secondarysort`

Data files:

`~/training_materials/developer/data/nameyeartestdata`

Exercise directory: `~/workspace/secondarysort`

In this lab, you will run a MapReduce job in different ways to see the effects of various components in a secondary sort program.

The program accepts lines in the form

```
lastname firstname birthdate
```

The goal is to identify the youngest person with each last name. For example, for input:

```
Murphy Joanne 1963-08-12
Murphy Douglas 1832-01-20
Murphy Alice 2004-06-02
```

We want to write out:

```
Murphy Alice 2004-06-02
```

All the code is provided to do this. Following the steps below you are going to progressively add each component to the job to accomplish the final goal.

Build the Program

1. In Eclipse, review but do not modify the code in the `secondarysort` project `example` package.

2. In particular, note the `NameYearDriver` class, in which the code to set the partitioner, sort comparator, and group comparator for the job is commented out. This allows us to set those values on the command line instead.

3. Export the jar file for the program as `secsort.jar`.

4. A small test datafile called `nameyeartestdata` has been provided for you, located in the secondary sort project folder. Copy the datafile to HDFS, if you did not already do so in the Writables lab.

Run as a Map-only Job

5. The Mapper for this job constructs a composite key using the `StringPairWritable` type. See the output of just the mapper by running this program as a Map-only job:

```
$ hadoop jar secsort.jar example.NameYearDriver \
-Dmapred.reduce.tasks=0 nameyeartestdata secsortout
```

6. Review the output. Note the key is a string pair of last name and birth year.

Run using the default Partitioner and Comparators

7. Re-run the job, setting the number of reduce tasks to `2` instead of `0`.

8. Note that the output now consists of two files; one each for the two reduce tasks. Within each file, the output is sorted by last name (ascending) and year (ascending). But it isn't sorted between files, and records with the same last name may be in different files (meaning they went to different reducers).

Run using the custom partitioner

9. Review the code of the custom partitioner class: `NameYearPartitioner`.

10. Re-run the job, adding a second parameter to set the partitioner class to use:
    `-Dmapreduce.partitioner.class=example.NameYearPartitioner`

11. Review the output again, this time noting that all records with the same last name have been partitioned to the same reducer.

    However, they are still being sorted into the default sort order (name, year ascending). We want it sorted by name ascending/year descending.

Run using the custom sort comparator

12. The `NameYearComparator` class compares Name/Year pairs, first comparing the names and, if equal, compares the year (in descending order; i.e. later years are considered "less than" earlier years, and thus earlier in the sort order.) Re-run the job using NameYearComparator as the sort comparator by adding a third parameter:
    ```
    -D mapred.output.key.comparator.class=
    example.NameYearComparator
    ```

13. Review the output and note that each reducer's output is now correctly partitioned and sorted.

Run with the NameYearReducer

14. So far we've been running with the default reducer, which is the Identity Reducer, which simply writes each key/value pair it receives. The actual goal of this job is to emit the record for the *youngest person* with each last name. We can do this easily if all records for a given last name are passed to a single reduce call, sorted in descending order, which can then simply emit the first value passed in each call.

15. Review the NameYearReducer code and note that it emits

16. Re-run the job, using the reducer by adding a fourth parameter:
    `-Dmapreduce.reduce.class=example.NameYearReducer`

    Alas, the job still isn't correct, because the data being passed to the reduce method is being grouped according to the full key (name and year), so multiple records with the same last name (but different years) are being output. We want it to be grouped by name only.

Run with the custom group comparator

17. The `NameComparator` class compares two string pairs by comparing only the name field and disregarding the year field. Pairs with the same name will be grouped into the same reduce call, regardless of the year. Add the group comparator to the job by adding a final parameter:
    ```
    -Dmapred.output.value.groupfn.class=
    example.NameComparator
    ```

18. Note the final output now correctly includes only a single record for each different last name, and that that record is the youngest person with that last name.

This is the end of the lab.

Notes for Upcoming Labs

VM Services Customization

For the remainder of the labs, there are services that must be running in your VM, and others that are optional. It is strongly recommended 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.

Data Model Reference

For your convenience, you will find a reference document depicting the structure for the tables you will use in the following labs. See file: Homework_DataModelRef.docx

Regular Expression (Regex) Reference

For your convenience, you will find a reference document describing regular expressions syntax. See file: Homework_RegexRef.docx

Lecture 9 Lab: Data Ingest With Hadoop Tools

In this lab 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: Setup 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 lab by running the following command:

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

2. To see the contents of your home directory, run the following command:

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

3. 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
```

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

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

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
```

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
```

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
```

This is the end of the lab.

Lecture 9 Lab: Using Pig for ETL Processing

In this lab 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: Since this lab builds on the previous one, it is important that you successfully complete the previous lab before starting this lab.

Background Information

Dualcore has recently started using online advertisements to attract new customers to its e-commerce site. Each of the two ad networks they 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 lab:

```
$ 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 labs 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 the 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 the 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 the 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:

d. 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`.

e. 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 the 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

f. 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 lab.

Lecture 9 Lab: Analyzing Ad Campaign Data with Pig

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

IMPORTANT: Since this lab builds on the previous one, it is important that you successfully complete the previous lab before starting this lab.

Step #1: Find Low Cost Sites

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

1. Change to the directory for this lab:

```
$ 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 lab, it is faster to test Pig scripts by using a local subset of the input data. Although explicit steps are not provided for creating local data subsets in upcoming labs, doing so will help you perform the labs 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 lab).

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 Dualcore 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 Lab #1: Count Ad Clicks

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

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

```
$ 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 Lab #2: Estimate the Maximum Cost of the Next Ad Campaign

When you reported the total number of clicks, the Marketing Director said that the goal is to get about three times that amount during the next campaign. Unfortunately, because the cost is based on the site and keyword, it isn't clear how much to budget for that campaign. You can 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 desired 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?

Professor's Note~

You can compare your solution to the one in the `bonus_02/sample_solution/` subdirectory.

Bonus Lab #3: Calculating Click-Through Rate (CTR)

The calculations you did at the start of this lab provided a rough idea about the success of the ad campaign, but didn't account for the fact that some sites display Dualcore's ads more than others. This makes it difficult to determine how effective their 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 Dualcore 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 lab:

```
$ 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.

This is the end of the lab.

Lecture 10 Lab: Analyzing Disparate Data Sets with Pig

In this lab, 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 the recent advertising campaign has had on sales.

IMPORTANT: Since this lab builds on the previous one, it is important that you successfully complete the previous lab before starting this lab.

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 their ad campaign began (February – April, 2013), as well as for the month during which their campaign ran (May, 2013).

1. Change to the directory for this lab:

```
$ 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 Dualcore began advertising. Next, you'll compare the sales of the specific product Dualcore advertised (product ID #1274348) during the same period to see whether the increase in sales was actually related to their campaign.

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

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

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

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

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

d. Display the results to your screen

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

```
$ pig count_tablet_orders_by_period.pig
```

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

Bonus Lab #1: Calculate Average Order Size

It appears that Dualcore's advertising campaign was successful in generating new orders. Since they 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 lab:

```
$ 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, it is recommended that 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 Dualcore advertised?

Bonus Lab #2: Segment Customers for Loyalty Program

Dualcore is considering starting a loyalty rewards program. This will provide exclusive benefits to their best customers, which will help to retain them. Another advantage is that it will also allow Dualcore to capture even more data about the shopping habits of their customers; for example, Dualcore can easily track their customers' in-store purchases when these customers provide 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 the customers will be eligible for membership in one of these three groups.

During this lab, 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 lab:

```
$ 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 lab.

Lecture 10 Lab: Extending Pig with Streaming and UDFs

In this lab 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: Since this lab builds on the previous one, it is important that you successfully complete the previous lab before starting this lab.

Background Information

Dualcore outsources its call center operations and costs have recently risen due to an increase in the volume of calls handled by these agents. Unfortunately, Dualcore does not have access to the call center's database, but they are provided 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 lab:

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

2. A Python script (`readtags.py`) is provided 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
```

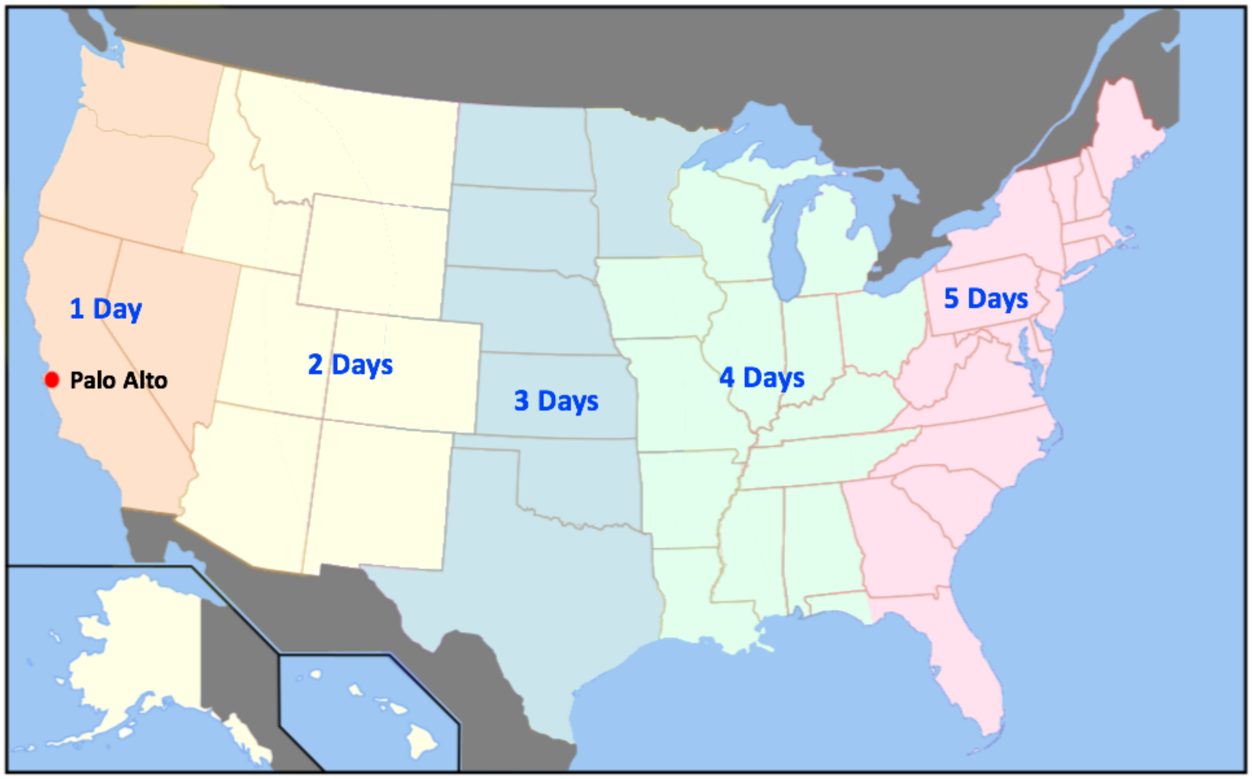
5. Now run the script again, this time specifying the parameter for May:

```
$ 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: their 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 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 script (`create_cust_location_data.pig`) has been provided to find the ZIP codes for customers who placed orders during the period of the ad campaign. It also excludes the ones who are already close to the 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 Dualcore's customers?

This is the end of the lab.

Lecture 11 Lab: Running Hive Queries from the Shell, Scripts, and Hue

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

IMPORTANT: Since this lab builds on the previous one, it is important that you successfully complete the previous lab before starting this lab.

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 they 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 the customer database so that Dualcore can send her the $5,000 prize.

1. Change to the directory for this lab:

```
$ 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.

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

```
quit;
```

6. 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

Dualcore can authorize the 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 Lab #1: Calculating Revenue and Profit

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"`.

Professor's Note~

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 lab.

Lecture 11 Lab: Data Management with Hive

In this lab 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: Since this lab builds on the previous one, it is important that you successfully complete the previous lab before starting this lab.

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 lab 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 lab:

```
$ 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 lab, 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 their best customers. Dualcore has 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 Lab #1: Alter and Drop a Table

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 lab.

Lecture 12 Lab: Gaining Insight with Sentiment Analysis

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

IMPORTANT: Since this lab builds on the previous one, it is important that you successfully complete the previous lab before starting this lab.

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 will begin by analyzing the numeric ratings customers have assigned to various products.

1. Change to the directory for this lab:

```
$ 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 that Dualcore sells. Although numeric ratings can help identify *which* product that is, they don't tell Dualcore *why* customers don't like the product. We could simply read through all the comments associated with that product to learn this information, but that approach doesn't scale. Next, you will use Hive's text processing support to analyze the comments.

1. The following query normalizes all comments on that product to lowercase, breaks them into individual words using the `SENTENCES` function, and passes those to the `NGRAMS` function to find the five most common bigrams (two-word combinations). Run the query 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 there are 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 Dualcore uncover a pricing error.

This is the end of the lab.

Lecture 12 Lab: Data Transformation with Hive

In this lab 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: Since this lab builds on the previous one, it is important that you successfully complete the previous lab before starting this lab.

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 lab:

```
$ 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 Dualcore's 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 abandons their cart after the second step. This might mean millions of dollars in lost revenue, so let's see if we can determine the cause.

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

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

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

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

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

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 is not 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 lab, 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 labs 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.

Professor's Note~

If you need a hint on how to write the statement, look at the file: `sample_solution/create_cart_items.hql`

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;
```

Professor's Note~

If this doesn't return 47, then compare your statement to the file: `sample_solution/create_cart_items.hql`. 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.

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. You will use a Hive UDF to 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 lab.

Lecture 13 Lab: Interactive Analysis with Impala

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

IMPORTANT: Since this lab builds on the previous one, it is important that you successfully complete the previous lab before starting this lab.

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

1. Issue the following commands to start Impala, then change to the directory for this lab:

```
$ sudo service impala-server start
$ sudo service impala-state-store start
$ cd $ADIR/exercises/interactive
```

2. First, start the Impala shell:

```
$ impala-shell
```

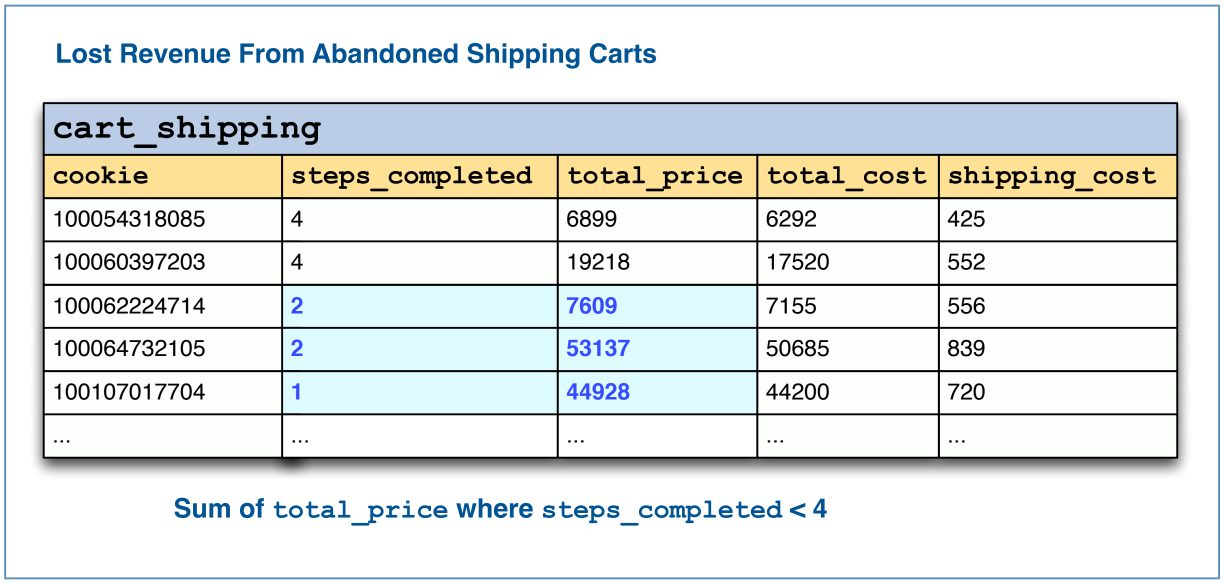
3. Since you created tables and modified data in Hive, Impala's cache of the metastore is outdated. You must refresh it before continuing by entering the following command in the Impala shell:

```
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.

   Professor's Note~

   If you need a hint on how to write this query, you can check the file:
   `sample_solution/abandoned_checkout_profit.sql`

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

3. How does this compare to the amount of profit Dualcore receives from customers who do complete the checkout process? Modify your previous query to consider only those records where `steps_completed = 4`, and then execute it in the Impala shell.

   Professor's Note~

   Check `sample_solution/completed_checkout_profit.sql` for a hint.

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

4. The previous two queries show 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.

   Professor's Note~

   If you need help writing this query, check the file:
   `sample_solution/checkout_profit_by_step.sql`

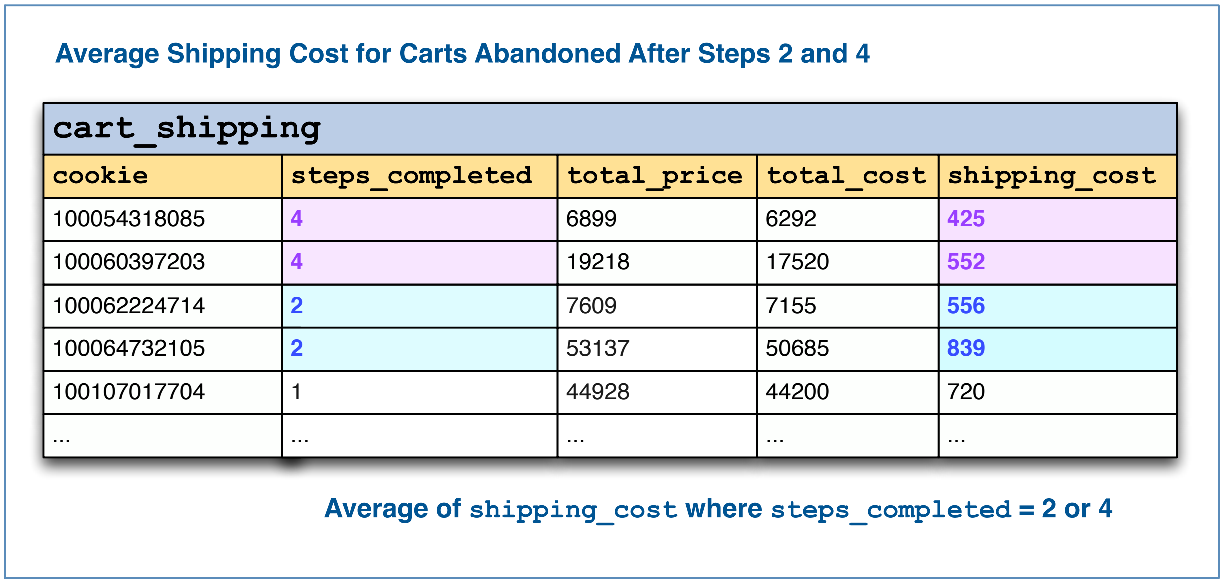
   You should observe that carts abandoned after step two represent an even higher average profit per order than completed orders.

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 the shipping cost is displayed 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

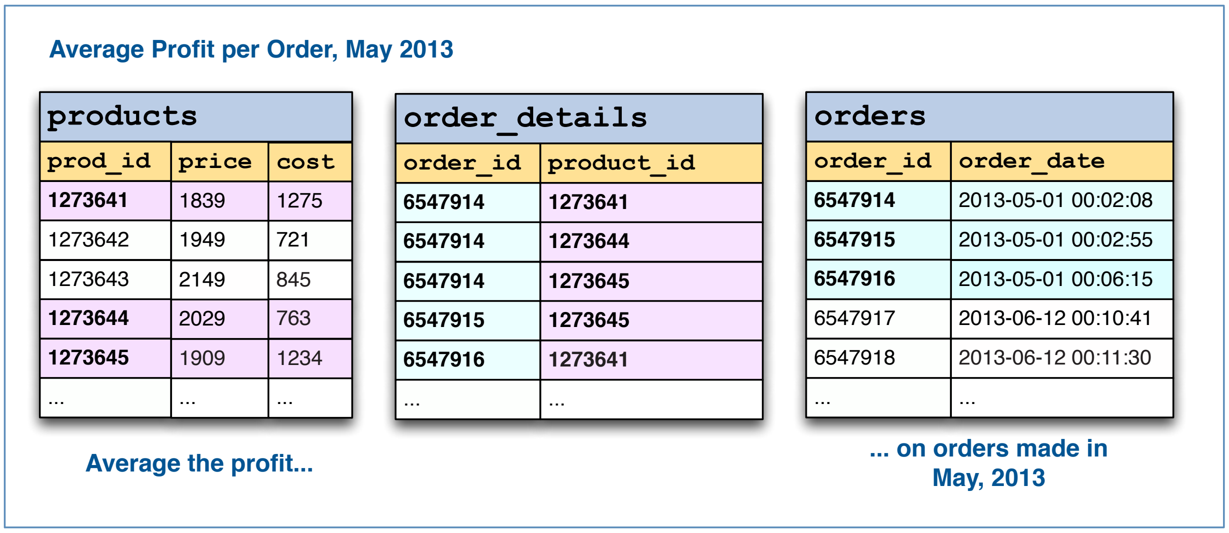
cookie	steps_completed	total_price	total_cost	shipping_cost
100054318085	4	6899	6292	425
100060397203	4	19218	17520	552
100062224714	2	7609	7155	556
100064732105	2	53137	50685	839
100107017704	1	44928	44200	720
...	...	...	...	...

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 Dualcore 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
Average the profit...
... on orders made in May, 2013

products

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

order_details

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

orders

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

- 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 Dualcore 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:

```
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 the 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 (Dualcore makes $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.

   Professor's Note~

   The file `sample_solution/avg_shipping_cost_50000.sql` contains the solution.

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

6. Since Dualcore won't know in advance who will abandon their cart, they would have to absorb the $12.28 average cost on *all* orders of at least $500. Would the extra money they 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 Dualcore currently earns from completed orders, it appears that they 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 lab.