201403

Apache Hadoop: Hands-On Exercises

General Notes

Cloudera's training courses use a Virtual Machine running the CentOS 6.3 Linux distribution. This VM has CDH (Cloudera's Distribution, including Apache Hadoop) installed in Pseudo-Distributed mode. 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 only key difference (apart from speed, of course!) being that the block replication factor is set to 1, since there is only a single DataNode available.

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 exercises, 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.

Points to note during the exercises

1. For most exercises, three folders are provided. Which you use will depend on how you would like to work on the exercises:

- `stubs`: contains minimal skeleton code for the Java classes you'll need to write. These are best for those with Java experience.

- `hints`: contains Java class stubs that include additional hints about what's required to complete the exercise. These are best for developers with limited Java experience.

- `solution`: Fully implemented Java code which may be run "as-is", or you may wish to compare your own solution to the examples provided.

2. As the exercises progress, and you gain more familiarity with Hadoop and MapReduce, we provide fewer step-by-step instructions; as in the real world, we merely give you a requirement and it's up to you to solve the problem! You should feel free to refer to the hints or solutions provided, ask your instructor for assistance, or consult with your fellow students!

3. There are additional challenges for some of the Hands-On Exercises. If you finish the main exercise, please attempt the additional steps.

Hands-On Exercise: 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 exercise 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 exercises, 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 than if you ran `hadoop fs -ls /foo`, which refers to a directory that doesn't exist and which would display an error message.

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 course.

```
$ 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 also have a Web server log file, which we will put into HDFS for use in future exercises. 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 exercises.

```
$ 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 Exercise

Hands-On Exercise: 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 exercise 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 exercise source directory, and list the contents:

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

This directory contains three “package” subdirectories: `solution`, `stubs` and `hints`. In this example we will be using the solution code, so 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 Exercise

Hands-On Exercise: 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 exercise you 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

We have created Eclipse projects for each of the Hands-On Exercises that use Java. We encourage you to use Eclipse in this course. Using Eclipse will speed up your development time.

1. Be sure you have run the course setup script as instructed in the General Notes section at the beginning of this manual. This script sets up the exercise 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

We've provided stub files for each of the Java classes for this exercise: `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" exercise, or follow the steps below ("Using Eclipse to Compile Your Solution") to use Eclipse.

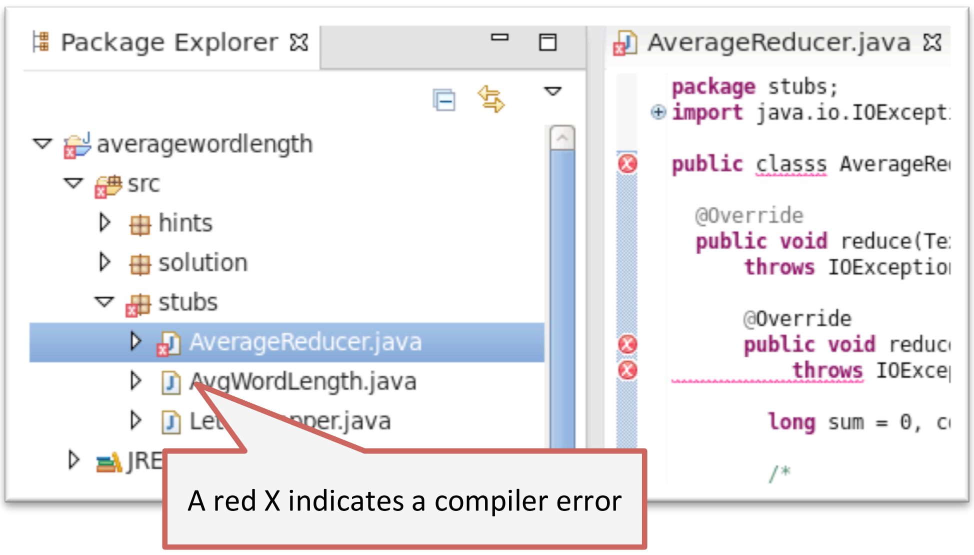
Step 3: Use Eclipse to Compile Your Solution

Follow these steps to use Eclipse to complete this exercise.

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

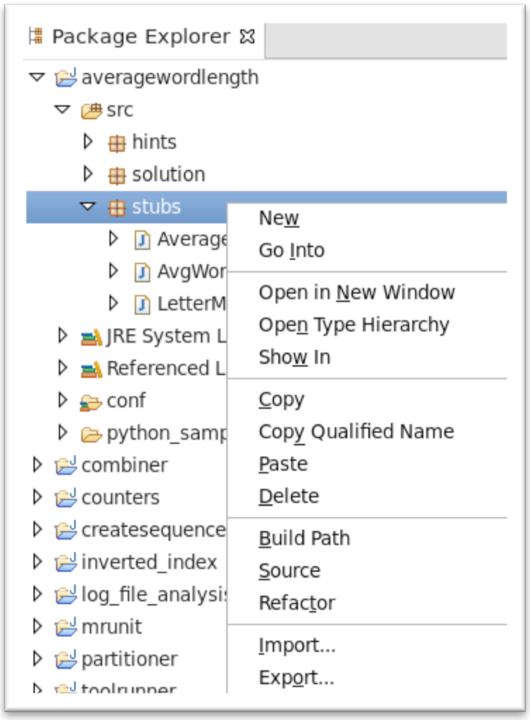
1. Verify that your Java code does not have any compiler errors or warnings.

   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.

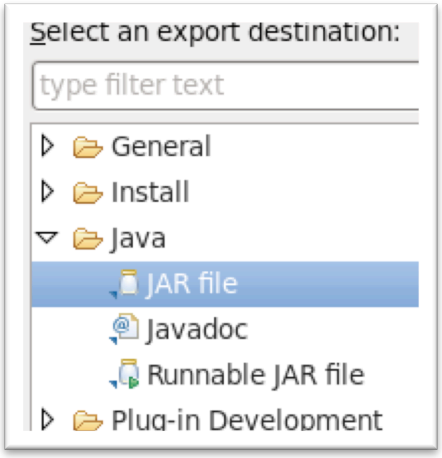


```
package stubs;
import java.io.IOExcept

public classs AverageRe

  @Override
  public void reduce(Te
      throws IOExceptio

      @Override
      public void reduc
          throws IOExce

          long sum = 0, c

          /*
```

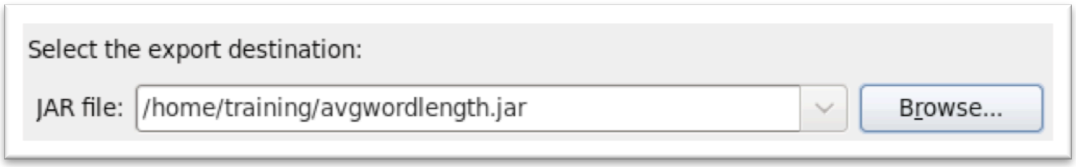
2. In the Package Explorer, open the Eclipse project for the current exercise (i.e. `averagewordlength`). Right-click the default package under the `src` entry and select Export.



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.:


Select the export destination:

JAR file: /home/training/avgwordlength.jar

Note: for more information about using Eclipse in this course, see the *Eclipse Exercise Guide*.

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" exercise. Make sure you use the correct package name depending on whether you are working with the provided `stubs`, stubs with additional `hints`, or just running the `solution` as is.

   (Throughout the remainder of the exercises, the instructions will assume you are working in the `stubs` package. Remember to replace this with the correct package name if you are using `hints` or `solution`.)

   ```
   $ 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.

Solution

You can view the code for the solution in Eclipse in the `averagewordlength/src/solution` folder.

This is the end of the Exercise

Hands-On Exercise: 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 exercise, 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" exercise.

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 exercise 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 exercise.

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 exercise (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 Exercise

Optional Hands-On Exercise: Writing a MapReduce Streaming Program

Files and Directories Used in this Exercise

Project directory: `~/workspace/averagewordlength`

Test data (HDFS):

`shakespeare`

In this exercise you will repeat the same task as in the previous exercise: 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 exercise.

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 with `hadoop fs -rm -r` *`dataToDelete`*.)

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

Note: The Perl example that was covered in class is in `~/workspace/wordcount/perl_solution`.

Solution in Python

You can find a working solution to this exercise 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 Exercise

Hands-On Exercise: 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. (If you need hints, refer to the code in the `hints` or `solution` packages.)

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 Exercise

Hands-On Exercise: 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 exercise, 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 exercise 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 exercise 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 Exercise

Hands-On Exercise: 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 exercise, 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" exercise.

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 `exercises` 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" exercise. If you completed the exercise (in the `writables` project) copy that code to the current project. Otherwise copy the class from the `writables solution` package.

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'll need to extend `org.apache.hadoop.io.LongWritable.Comparator`.

This is the end of the Exercise

Hands-On Exercise: Importing Data With Sqoop

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

Consider the MySQL database `movielens`, derived from the MovieLens project from University of Minnesota. (See note at the end of this exercise.) 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 Exercise

Note:

Hands-On Exercise: Manipulating Data With Hive

Files and Directories Used in this Exercise

Test data (HDFS):

`movie`

`movierating`

Exercise directory: `~/workspace/hive`

In this exercise, you will practice data processing in Hadoop using Hive.

The data sets for this exercise are the `movie` and `movierating` data imported from MySQL into Hadoop in the "Importing Data with Sqoop" exercise.

Review the Data

1. Make sure you've completed the "Importing Data with Sqoop" exercise. Review the data you already loaded into HDFS in that exercise:

```
$ hadoop fs -cat movie/part-m-00000 | head
…
$ hadoop fs -cat movierating/part-m-00000 | head
```

Prepare The Data For Hive

For Hive data sets, you create *tables*, which attach field names and data types to your Hadoop data for subsequent queries. You can create *external* tables on the `movie` and `movierating` data sets, without having to move the data at all.

Prepare the Hive tables for this exercise by performing the following steps:

2. Invoke the Hive shell:

```
$ hive
```

3. Create the `movie` table:

```
hive> CREATE EXTERNAL TABLE movie
      (id INT, name STRING, year INT)
      ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
      LOCATION '/user/training/movie';
```

4. Create the `movierating` table:

```
hive> CREATE EXTERNAL TABLE movierating
      (userid INT, movieid INT, rating INT)
      ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
      LOCATION '/user/training/movierating';
```

5. Quit the Hive shell:

```
hive> QUIT;
```

Practicing HiveQL

If you are familiar with SQL, most of what you already know is applicably to HiveQL. Skip ahead to section called "The Questions" later in this exercise, and see if you can solve the problems based on your knowledge of SQL.

If you are unfamiliar with SQL, follow the steps below to learn how to use HiveSQL to solve problems.

1. Start the Hive shell.

2. Show the list of tables in Hive:

```
hive> SHOW TABLES;
```

The list should include the tables you created in the previous steps.

> **Note:** By convention, SQL (and similarly HiveQL) keywords are shown in upper case. However, HiveQL is not case sensitive, and you may type the commands in any case you wish.

3. View the metadata for the two tables you created previously:

```
hive> DESCRIBE movie;
hive> DESCRIBE movieratings;
```

Hint: You can use the up and down arrow keys to see and edit your command history in the hive shell, just as you can in the Linux command shell.

4. The `SELECT * FROM` *`TABLENAME`* command allows you to query data from a table. Although it is very easy to select *all* the rows in a table, Hadoop generally deals with very large tables, so it is best to limit how many you select. Use LIMIT to view only the first *N* rows:

```
hive> SELECT * FROM movie LIMIT 10;
```

5. Use the WHERE clause to select only rows that match certain criteria. For example, select movies released before 1930:

```
hive> SELECT * FROM movie WHERE year < 1930;
```

6. The results include movies whose year field is 0, meaning that the year is unknown or unavailable. Exclude those movies from the results:

```
hive> SELECT * FROM movie WHERE year < 1930
      AND year != 0;
```

7. The results now correctly include movies before 1930, but the list is unordered. Order them alphabetically by title:

```
hive> SELECT * FROM movie WHERE year < 1930
      AND year != 0 ORDER BY name;
```

8. Now let's move on to the movierating table. List all the ratings by a particular user, e.g.

```
hive> SELECT * FROM movierating WHERE userid=149;
```

9. `SELECT *` shows all the columns, but as we've already selected by `userid`, display the other columns but not that one:

```
hive> SELECT movieid,rating FROM movierating WHERE
userid=149;
```

10. Use the JOIN function to display data from both tables. For example, include the name of the movie (from the movie table) in the list of a user's ratings:

```
hive> select movieid,rating,name from movierating join
movie on movierating.movieid=movie.id where userid=149;
```

11. How tough a rater is user 149? Find out by calculating the average rating she gave to all movies using the AVG function:

```
hive> SELECT AVG(rating) FROM movierating WHERE
userid=149;
```

12. List each user who rated movies, the number of movies they've rated, and their average rating.

```
hive> SELECT userid, COUNT(userid),AVG(rating) FROM
movierating GROUP BY userid;
```

13. Take that same data, and copy it into a new table called `userrating`.

```
hive> CREATE TABLE USERRATING (userid INT,
      numratings INT, avgrating FLOAT);
hive> insert overwrite table userrating
      SELECT userid,COUNT(userid),AVG(rating)
      FROM movierating GROUP BY userid;
```

Now that you've explored HiveQL, you should be able to answer the questions below.

The Questions

Now that the data is imported and suitably prepared, write a HiveQL command to implement each of the following queries.

Working Interactively or In Batch

Hive:

You can enter Hive commands interactively in the Hive shell:

```
$ hive
. . .
hive>  Enter interactive commands here
```

Or you can execute text files containing Hive commands with:

```
$ hive -f file_to_execute
```

1. What is the oldest known movie in the database? Note that movies with unknown years have a value of 0 in the `year` field; these do not belong in your answer.

2. List the name and year of all unrated movies (movies where the `movie` data has no related `movierating` data).

3. Produce an updated copy of the `movie` data with two new fields:

`numratings`	- the number of ratings for the movie
`avgrating`	- the average rating for the movie

   Unrated movies are not needed in this copy.

4. What are the 10 highest-rated movies? (Notice that your work in step 3 makes this question easy to answer.)

Note: The solutions for this exercise are in `~/workspace/hive`.

This is the end of the Exercise

Hands-On Exercise: 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 exercise, you will inspect and run Oozie workflows.

1. Start the Oozie server

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

2. Change directories to the exercise 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 Exercise