Hands-On Exercise: Using HDFS

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

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 start a terminal and run this program with no arguments, it prints a help message. To try this, run the following command:

```
$ hadoop
```

(Note: although your command prompt is more verbose, we use '`$`' to indicate the command prompt for brevity's sake.)

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 this 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 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. Try running:

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

There are no files, 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 directory containing the sample data we will be using in the course.

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

If you perform a 'regular' `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 Apache 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 copied into HDFS.

1. Enter:

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

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

2. The `glossary` file included in the tarball 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`.

Note that when you pipe the output of the `fs -cat` command to a local UNIX command, the full contents of the file are still extracted from HDFS and sent to your local machine. Once on your local machine, the file contents are then modified before being displayed.

4. If you want to download a file and manipulate it in the local filesystem, you can 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 commands associated with the `FsShell` subsystem, to perform most common filesystem manipulations: `mv`, `cp`, `mkdir`, etc.

1. Enter:

```
$ hadoop fs
```

This displays a brief usage report of the commands 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

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 working directory, and take a directory listing:

```
$ cd ~/training_materials/developer/exercises/wordcount
$ ls
```

   This directory 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. Compile the three Java classes:

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

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

Your command includes the classpath for the Hadoop core API classes. The compiled (`.class`) files are placed in your local directory.

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

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

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

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

This `hadoop jar` command names the JAR file to use (`wc.jar`), the class whose `main` method should be invoked (`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`.

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

```
$ hadoop jar wc.jar 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.

6. 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.)

7. 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. Note that you could have specified `wordcounts/*` just as well in this command.

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

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

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

9. 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. The MapReduce job, once submitted to the Hadoop daemons, runs independently of any initiating process.

Losing the connection to the initiating process does not kill a MapReduce 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 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, reporting its progress, informs you that the job has failed.

This is the end of the Exercise

Hands-On Exercise: Writing a MapReduce Program

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. You can write the job in Java or using Hadoop Streaming.

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:

```
Now is definitely the time
```

The output would be:

```
N    3
d    10
i    2
t    3.5
```

(For the initial solution, your program can be case-sensitive—which is the case for Java string processing by default.)

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:

```
Now is definitely the time
```

Your Mapper should emit:

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

The Reducer

Thanks to the sort/shuffle phase built in to MapReduce, the Reducer receives the keys in sorted order, and all the values for one key appear together. So, for the Mapper output above, the Reducer (if written in Java) receives this:

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

If you will be writing your code using Hadoop Streaming, your Reducer would receive the following:

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

For either type of input, the final output should be:

```
N	3
d	10
i	2
t	3.5
```

Choose Your Language

You can perform this exercise in Java or Hadoop Streaming (or both if you have the time). 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 if you like. Following are a discussion of the program in Java, and then a discussion of the program in Streaming.

If you complete the first part of the exercise, there is a further exercise for you to try. See page 21 for instructions.

Set up Eclipse

We have created Eclipse projects for each of the Hands-On Exercises. Using Eclipse will speed up your development time.

Even if you do not plan to use Eclipse to develop your code, you still need to set up Eclipse. In the "Writing Unit Tests With the MRUnit Framework" lab, you will use Eclipse to run unit tests.

Follow these instructions to set up Eclipse by importing projects into the environment:

1. Launch Eclipse.
2. Select Import from the File menu.
3. Select General -> Existing Projects into Workspace, and click Next.
4. Specify /home/training/workspace in the Select Root Directory field. All the exercise projects will appear in the Projects field.
5. Click OK, then click Finish. That will import all projects into your workspace.

The steps to export Java code to a JAR file and run the code are described in the slides for this chapter. The following is a quick review of the steps you will need to perform to run Hadoop source code developed in Eclipse:

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.

2. Right-click the default package entry for the Eclipse project (under the `src` entry).

3. Select Export

4. Select Java > .JAR File from the Export dialog box, then click Next.

5. Specify a location for the JAR file. You can place your JAR files wherever you like.

6. Run the `hadoop jar` command as you did previously in the Running a MapReduce Job exercise.

For more information about running a Hadoop job when working in Eclipse, including screen shots, refer to the slides for this chapter.

The Program in Java

Basic stub files for the exercise can be found in `~/training_materials/developer/exercises/averagewordlength`. If you like, you can use the `wordcount` example (in `~/training_materials/developer/exercises/wordcount` as a starting point for your Java code. Here are a few details to help you begin your Java programming:

1. 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.

2. 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
```

3. Define the Reducer

   In a single invocation the Reducer receives a string containing one letter along with an iterator of integers. For this call, the reducer should emit a single output of the letter and the average of the integers.

4. Test your program

   Compile, jar, and test your program. You can use the entire Shakespeare dataset for your input, or you can try it with just one of the files in the dataset, or with your own test data.

Solution in Java

The directory `~/training_materials/developer/exercises/averagewordlength/sample_solution` contains a set of Java class definitions that solve the problem.

The Program Using Hadoop Streaming

For your Hadoop Streaming program, 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
```

   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.

3. Run the streaming program

   You can run your program with Hadoop Streaming via:

```
$ 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`.)

Solution in Python

You can find a working solution to this exercise written in Python in the directory `~/training_materials/developer/exercises/averagewordlength/python`.

Additional Exercise

If you have more time, attempt the additional exercise. In the `log_file_analysis` directory, you will find stubs for the Mapper and Driver. (There is also a sample solution available.)

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

Note: If you want, you can 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.

1. Change directory to `~/training_materials/developer/exercises/log_file_analysis`.

2. Using the stub files in that 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: You can re-use the Reducer provided in the WordCount hands-on exercise, or you can write your own if you prefer.**

This is the end of the Exercise