Cloudera Developer Training for Apache Hadoop: Hands-On Exercises

General Notes

Cloudera's training courses use a Virtual Machine running the CentOS 5.6 Linux distribution. This VM has Cloudera's Distribution including Apache Hadoop version 3 (CDH3) installed in Pseudo-Distributed mode. Pseudo-Distributed mode is a method of running Hadoop whereby all five 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.

Points to note while working in the VM

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

2. 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.

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

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

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

2. Sample solutions are always available in the sample_solution subdirectory of the exercise directory.

3. 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! There are 'stub' files for each exercise to get you started, and you should feel free to ask your instructor for assistance at any time. We also provide some hints in many of the exercises. And, of course, you can always consult with your fellow students.

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. Hadoop is installed in the `/usr/lib/hadoop` directory. You can refer to this using the environment variable `$HADOOP_HOME`, which is automatically set in any terminal you open on your desktop.

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. For example, you can see the contents of the uploaded `shakespeare` directory by running:

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

You also *could* have uploaded the Shakespeare files into HDFS by running the following **although you should not do this now, as the directory has already been uploaded**:

```
$ hadoop fs -put shakespeare shakespeare
```

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: `rmr` (recursive `rm`), `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: Run 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 four Java classes:

```
$ javac -classpath $HADOOP_HOME/hadoop-core.jar *.java
```

Your command includes the classpath for the Hadoop core API classes. The compiled (`.class`) files are placed in your local directory. These Java files use the 'old' `mapred` API package, which is still valid and in common use: ignore any notes about deprecation of the API which you may see.

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 tries to prevent 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-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-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 -rmr 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:

```
$ hadoop 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:

```
$ hadoop 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.

Using the Local JobRunner and Local Filesystem

1. Run the commands:

```
$ cd ~/training_materials/developer/data
$ hadoop jar ../exercises/wordcount/wc.jar  \
WordCount -fs file:///  \
-jt local shakespeare wordcounts
```

The driver class uses the `Tool` and `ToolRunner` classes in the Hadoop API to support the use of 'generic options' on your command line when you submit of the MapReduce job. The options you use here are:

`-fs file:///` to use the local filesystem for your job instead of HDFS

`-jt local` to use the Local JobRunner instead of Hadoop

The Local JobRunner is a cut-down version of the MapReduce job execution engine that runs entirely in your client process. It runs quickly on small data

inputs because it avoids the overhead of submitting a true MapReduce job to the Hadoop cluster.

Also, because you ran this job using the local filesystem, you will find a new local directory named wordcounts with the output of your MapReduce job. Peruse and remove this directory as you wish.

Using the `Tool` class in your driver class lets you use command line options to easily test your MapReduce jobs on your local machine, with faster execution times on small data sets. Such test runs help you take relatively easy steps toward running real-world jobs on actual Hadoop clusters.

2. Enter:

```
$ cd ~/training_materials/developer/exercises/wordcount
$ hadoop jar wc.jar WordCount bogus
```

This invocation is deliberately incorrect, and will cause your program to display a usage report listing the generic command line options supported by the `Tool` class (and its embedded `GenericOptionsParser` object).

This is the end of the Exercise

Hands-On Exercise: Average Word Lengths

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 Java string processing is 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 shoud 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 20 for instructions.

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 datatypes 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
```

Notice 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 \
$HADOOP_HOME/contrib/streaming/ha*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 -rmr 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 subdirectory `additional_exercise`, which is in the `averagewordlength` directory, you will find an Apache log file and stubs for the Mapper and Reducer. (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 included in that directory.

1. Change directory to `~/training_materials/developer/exercises/averagewordlength/additional_exercise`

2. Extract the Apache log file

```
$ gunzip access_log.gz
```

3. Upload the file to HDFS

```
hadoop fs –mkdir weblog
hadoop fs –put access_log weblog
```

4. Using the stub files in that directory, write a Mapper and Driver code to count the number of hits made from each IP address. 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

Hands-On Exercise: Sqoop Imports

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

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 \
  --table movie --fields-terminated-by '\t' \
  --username training --password training
```

Note: separating the fields in the HDFS file with the tab character is one way to manage compatibility with Hive and Pig, which we will use in a future exercise.

5. Import the `movierating` table into Hadoop.

Repeat step 4, but for this table.

This is the end of the Exercise

Note:

This exercise 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

Optional Hands-On Exercise: Writing a Partitioner

In this Hands-On 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.

We will be modifying the program you wrote to solve the Additional Exercise on page 20.

Prepare The Exercise

1. Change directories to `~/training_materials/developer/exercises/partitioner`

2. If you completed the Additional Exercise on page 20, copy the code for your Mapper, Reducer and driver to this directory. If you did not, follow steps 1, 2, and 3 on page 20, then change directories back to the `partitioner` directory and copy the sample solution from `~/training_materials/developer/exercises/averagewordlength/additional_exercise/sample_solution` into the current directory.

The Problem

The code you now have in the partitioner directory counts all the hits made to a Web server from each different IP address. (If you did not complete the exercise, view the code and be sure you understand how it works.) Our aim in this exercise is to modify the code such that we get a different result: We want **one output file per month,** each file containing the number of hits from each IP address in that month. In other words, there will be 12 output files, one for each month.

Note: we are actually breaking the standard MapReduce paradigm here. The standard paradigm 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.

Writing The Solution

Before you start writing your code, ensure that you understand the required outcome; ask your instructor if you are unsure.

1. You will need to modify your driver code to specify that you want 12 Reducers. Hint: `conf.setNumReduceTasks()` specifies the number of Reducers for the job.

2. Change the Mapper so that instead of emitting a `1` for each value, instead it emits the entire line (or just the month portion if you prefer).

3. Write a Partitioner which 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.

4. Configure your job to use your custom Partitioner (hint: use `conf.setPartitionerClass()` in your driver code).

5. Compile and test your code. Hints:

    a. Write unit tests for your Partitioner!

    b. Create a directory in HDFS with just a small portion of the entire log file to run your initial tests

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

This is the end of the Exercise

Hands-On Exercise: Write and Implement a Combiner

In this Hands-On Exercise, you will write and implement a Combiner to reduce the amount of intermediate data sent from the Mapper to the Reducer. You will use the WordCount Mapper and Reducer from an earlier exercise.

There are further exercises if you have more time.

Implement a Combiner

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

2. Copy the WordCount Mapper and Reducer into this directory

```
$ cp ../wordcount/WordMapper.java .
$ cp ../wordcount/SumReducer.java .
```

3. Complete the `WordCountCombiner.java` code and, if necessary, the `SumCombiner.java` code to implement a Combiner for the Wordcount program.

4. Compile and test your solution.

If You Have More Time: Write a set of MRUnit tests for WordCount

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

1. Launch Eclipse. Expand the 'Hadoop' folder, right-click on the 'src' folder and from the menu choose New->Package.

2. Name the Package '`hadoop.dev.api.mrunit`'

3. Right-click on your new package, and choose 'Import'

4. Expand the 'General' folder, and choose 'File System'. Click on the 'Next' button.

5. In the 'From Directory' section, choose Browse and browse to the `wordcount` exercise directory, then click OK

6. Click the checkboxes beside '`WordMapper.java`' and '`SumReducer.java`' and click on Finish.

7. Notice that the two files have red crosses by their names in the Package Explorer. This is because they currently do not have package declarations. So add those, by double-clicking on each of the two files and adding the line:

```
package hadoop.dev.api.mrunit;
```

   Save each file after you have done this, and you will see that the warning red crosses disappear.

8. Now import the stub test file, by right-clicking on the package again, choosing 'Import', selecting 'General'->'File System', and navigating to the `~/training_materials/developer/exercises/mrunit` directory. Import the '`TestWordCount.java`' file.

9. Notice that, again, this file has a warning beside it. This is because we have not added the MRUnit libraries to our Hadoop project. Right-click on 'Hadoop' in the Package Explorer, and choose 'Build Path'->'Add External Archives'. Navigate to `/usr/lib/hadoop-0.20/contrib/mrunit`, select the `hadoop-mrunit` JAR, and click on OK.

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

11. Run the tests by right-clicking on `TestWordCount.java` and choosing 'Debug As' -> 'JUnit Test'.

12. Observe the failures.

13. Now implement the three tests. If you need hints, there is a sample solution in the sample solution directory within the `mrunit` directory.

This is the end of the Exercise

Hands-On Exercise: Create an 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.

We have provided stub files in the directory `~/training_materials/developer/exercises/inverted_index`

Prepare the Input Data

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

```
$ cd \
~/training_materials/developer/exercises/inverted_index
$ 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:

```
conf.setInputFormat(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 Reporter object can be used to retrieve the name of the file like this:

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

Build and Test Your Solution

Test against the `invertedIndexInput` data you loaded in step 1 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 should 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.

Solution

You can find a working solution to this exercise in the directory `~/training_materials/developer/exercises/invertedindex/sample_solution`.

Additional Exercise: Word Co-Occurrence

If you have more time, continue on with this additional exercise, which will count the number of times words appear next to each other, using the same input data as in the previous exercise. (Note: this 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 directory 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.

3. Extra credit: Write a further MapReduce job to sort the output from the first job so that the list of pairs of words appears in ascending frequency.

This is the end of the Exercise

Hands-On Exercise: Using 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 a previous exercise.

Review the Data

1. Review the data already loaded into HDFS:

```
$ 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:

1. Invoke the Hive shell:

```
$ hive
```

2. 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';
```

3. 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';
```

4. Quit the Hive shell:

```
hive> QUIT;
```

The Questions

Now that the data is imported and suitably prepared, use Hive to answer the following questions.

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.)

This is the end of the Exercise

Hands-On Exercise: Using Counters and a Map-Only Job

In this Hands-on Exercise you will create a Map-only MapReduce job which will use 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.

If you have not already done so, upload the Web access log to HDFS.

1. Change directory to `~/training_materials/developer/exercises/averagewordlength/additional_exercise`

2. Extract the Apache log file

```
$ gunzip access_log.gz
```

3. Upload the file to HDFS

```
$ hadoop fs –mkdir weblog
$ hadoop fs –put access_log weblog
```

4. Change directory to the counters directory within the exercises directory

```
$ cd ../../counters
```

5. Complete the stub files to provide the solution. As before, you can reuse the SumReducer from the WordCount exercise (or write your own).

Hints

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

Use a counter group called something like 'ImageCounter', with names 'gif', 'jpeg' and 'other'.

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

As always, a sample solution is available if you need more hints.

This is the end of the Exercise

Hands-On Exercise: Implementing Word Co-Occurrence with a Custom WritableComparable

In this Hands-On Exercise, you will create a custom WritableComparable to improve the Word Co-Occurrence exercise from earlier in the course.

1. Change directories to the `writables` directory in the `exercises` directory.

2. Edit the stub files to solve the problem. If you completed the Word Co-Occurrence additional exercise earlier, use those files as your starting point for the Mapper and Reducer. If you did not complete the exercise earlier in the course, you will find a sample solution in the `word_co-occurrence` directory in the exercises directory. Copy that and modify it appropriately.

Hints

You need to create a WritableComparable object which will hold the two strings. The object will need an empty constructor for serialization, a standard constructor which will be given two strings, a `toString` method, and the methods required by WritableComparables.

Note that Eclipse can automatically generate the `hashCode` and `equals` methods for you: right-click on the source code and choose 'Source'->'Generate hashCode() and equals()'.

As always, a sample solution is available.

This is the end of the Exercise

Hands-On Exercise: Running an Oozie Workflow

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

1. Change directories to the `oozie-labs` directory within the `exercises` directory

2. Start the Oozie server

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

3. Change directories to `lab1-java-mapreduce/job`

```
$ cd lab1-java-mapreduce/job
```

4. Inspect the contents of the `job.properties` and `workflow.xml` files. You will see that this is our standard WordCount job.

5. Change directories back to the main `oozie-labs` directory

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

6. We have provided a simple shell script to submit the Oozie workflow. Inspect `run.sh`

```
$ cat run.sh
```

7. Submit the workflow to the Oozie server

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

   Notice that Oozie returns a job identification number.

8. Inspect the progress of the job

```
$ oozie job -oozie http://localhost:11000/oozie –info <job_id>
```

9. When the job has completed, inspect HDFS to confirm that the output has been produced as expected.

10. 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. 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