

# Programming Assignment #2

## Map-Reduce

Due Date: Wednesday, December 3, 2008 at 5:00PM PST

In this assignment, you will become familiar with and explore the Map-Reduce programming model, as embodied by the Hadoop framework. Rather than grade this assignment in terms of absolute performance, we are looking for correct functionality and insightful analysis.

### SYNOPSIS

Your group's assignment is to take the application stubs that we provide and implement 3 simple Map-Reduce applications.

1. A two-letter digram counter.
2. A word-pair counter.
3. A sentence fragment counter, which is part of a simple random sentence generator.

We provide a fully-functional word counting application that you can use as a reference when developing these new applications. We have made available several datasets composed of subsets of Project Gutenberg's digital library. Project Gutenberg's mission is to archive out-of-print and public domain books in electronic format, and thus has an excellent corpus of English text.

After implementing the 3 applications, you will perform sensitivity experiments wherein you

1. Vary the size of the input dataset to your application.
2. Vary the number of Map tasks your application uses.
3. Toggle whether or not your application uses a Combiner class.

You will submit your applications and a short report, detailing:

1. Answers to questions from the Sensitivity Analysis section of this assignment. This is the most important part of this assignment. We expect 1 to 1.5 pages per topic.
2. Correctness results for each Map-Reduce application.

### DEVELOPMENT ENVIRONMENT

You will be coding your assignment with Hadoop 0.18.1 using Java 1.6.0b11 (OpenJDK) under Linux. We provide a complete development environment on the same cluster as used in Programming Assignment 1.

## GETTING STARTED

As soon as possible, ssh into *soda.stanford.edu* to ensure that your account is active. Since this is the first time we've given this programming assignment, EXPECT THINGS TO BREAK! Start MUCH sooner rather than later.

We have provided a simple Makefile that should encompass the tasks of a) compiling your applications, b) running them with Hadoop, and c) checking for correctness. Unpack the assignment into your home directory by typing:

```
cd
wget http://ee282.stanford.edu/pa2.tar.gz
tar zxvf pa2.tar.gz
cd pa2
```

Before working with Hadoop, you must import environment variables by typing:

```
source setup.sh
```

You will have to do this every time you log into soda. The included Makefile takes care of compiling your application, running it, and providing output for correctness checking. As an example of how to use it, we've included a fully functional WordCount application that you will now try.

The general usage for the Makefile is to type “make <dataset> <application>”, where dataset is one of “tiny”, “small”, “medium”, or “large” and application is one of “WordCount”, “DigramCount”, “WordCorrelate”, or “FragmentFinder”. For example, to run WordCount on the tiny dataset, type:

```
make tiny WordCount
```

You should see the Hadoop run-time system display Map and Reduce progress until the job completes. After the job has completed, you may check for correctness by typing “make check”. This will download the output from the Hadoop Filesystem into the “out” directory and probe it. For the WordCount application, correctness is checked by looking at how many instances of the word “and” appears in the text. For example, the output of “make check” after running WordCount on the tiny dataset should be:

```
and      2
```

which indicates that the word “and” appears twice. You can verify this manually by looking at “tiny.txt”, which is the tiny dataset and is included in pa2.tar.gz.

If you would like to study your output data further, look at the “part-XXXXX” files that appear in the “out” directory after you run “make check”.

## Application 1: DigramCounter

From Wikipedia: A digram is a pair of characters used to write one phoneme (distinct sound) or a sequence of phonemes that does not correspond to the normal values of the two characters combined. The “sc” at the beginning of “science” is an example of a digram.

In this application, you will use a relaxed definition of digram where any pair of letters that occurs in a text is a digram. For example, the list of digrams in the word “alphabet” is

al lp ph ha ab be et

## Programming

Using WordCount as a model, modify DigramCounter.java to count all digrams of the input text. Use the following statement at the top of your Map method to preprocess the input text:

```
String line = value.toString().  
    toLowerCase().replaceAll("[^a-zA-Z ]", " ").trim();
```

This statement makes every word lower-case, replaces non-letter characters with spaces, and removes leading and trailing spaces from the string. This should help ensure that everyone obtains the same results.

## Correctness

The following are counts of the “th” digram in each dataset:

tiny	9
small	320721
medium	3344914
large	101376555

Interestingly, you can verify the correctness of your application by comparing your results (found in the “out” subdirectory) with other digram measurements of English text.

See <http://www-math.cudenver.edu/~wcherowi/courses/m5410/engstat.html>

**NOTE:** Your results *may* differ slightly from the above counts, since Hadoop may split the input datasets on arbitrary boundaries. Also, we may check for correctness by looking at a digram other than “th”. You should inspect both your output (found in the “out” subdirectory after you run “make check”) and the input files to convince yourself further that your application is functioning correctly. “tiny.txt” is included in the PA2 tarball. The other input files can be found under /athitos/export/fall08\_ee282/pa2/.

## Application 2: WordCorrelate

In this application, you will count how often a word appears on the same line as another word in a corpus of text. For example, in the lines:

```
this example illustrates the meaning of the description
and the next line clarifies the example further
```

the word “the” appears 4 times on the same line as “example”.

Do not double count a word as appearing on the same line as itself. For instance, in the above text, don’t count the word “example” as appearing on the same line as “example”, but DO count the word “the” as appearing on the same line as “the” twice.

As output from your Reduce method, use “word1:word2” as the format for your key and the number of times that word combination appears on a line as the value. In the above text, for example, you would emit

```
example:the    4
```

to signify that “example” appears 4 times on the same line as “the”.

## Programming

Again, using WordCount as a model, modify WordCorrelate.java to count every time a word appears on the same line as another word. Use the following statement at the top of your Map method to preprocess the input text:

```
String line = value.toString().
    toLowerCase().replaceAll("[^a-zA-Z ]", " ").trim();
```

In your Reduce method, do not output a word combination if it only appears once. This optimization will significantly reduce the size of the output file.

## Correctness

Do NOT test this application with the large dataset. The following are counts of how many times “it:the” appears in each dataset:

tiny	3
small	14659
medium	145693

## Application 3: FragmentFinder

This application is a component of a cute algorithm for generating random sentences that are occasionally quite coherent. The basic approach of this random sentence generator is to scrape sentence fragments (5 words in length) from a text corpus, and then to chain the fragments together so that the last word of a fragment matches the first word of the next fragment. For example, given the fragments “dig up on a *fellow*”, “*fellow* who has turned *opium*”, “*opium* to white men *insisted*”, “*insisted* upon an explanation *when*”, “*when* i was a child”, you could construct the semi-coherent sentence:

dig up on a *fellow* who has turned *opium* to white men *insisted* upon an explanation *when* i was a child

Your FragmentFinder application is going to solve the problem of finding the most common fragment that starts with a particular word. A simple sequential algorithm can then generate sentences given a database of “word → fragment” pairs.

This application is more complicated than the previous applications in that it requires two Map-Reduce passes. **In the first pass, you should count how often each sentence fragment occurs. In the second pass, you should determine the absolute most frequent sentence fragment that begins with each word that any sentence fragment starts with.** The output of the first pass will be used directly as the input for the second pass.

## Programming

Use the following statement at the top of your first Map method to preprocess the input text:

```
String line = value.toString().  
    replaceAll("[^a-zA-Z0-9,.' ]", " ").trim();
```

The first Map-Reduce pass counts fragments. It should take each line, break it into fragments of 5 words, and then count them. For example, a line that says “this is a simple example yo” should generate the fragments:

this is a simple example  
is a simple example yo

The Reduce will count how often these fragments occur. Your second Map-Reduce should associate words with a single fragment. For example, if you have the following fragments that start with “the”:

the rare fragment at hand      (only occurs once)  
the president of the united      (occurs thousands of times)

You should emit a single record with a key of “the” and a value of “the president of the united”.

## Correctness

Do NOT test this application with the large dataset. The following are the fragments associated with the word “and”:

tiny	and mathematician Leibnitz who said
small	and at the same time
medium	and at the same time

## Amusements

After checking for correctness, you can use the “chain.sh” script provided with PA2 to construct random sentences. For example, type “./chain.sh british”:

british poets of the nineteenth century but it is one of the most complete rapid and  
economical known in the trade

The output will, of course, depend upon the input dataset.

## Sensitivity Analysis

In this part of the assignment, you will vary parameters that affect the execution of your Map-Reduce programs and gain some intuition about the engineering involved in designing high-performance, distributed applications. **Note:** each time you run a job, the exact number of nodes used for map and reduce tasks may vary. Multiple runs for each dataset may be needed to achieve stable results.

### Map Tasks

The Hadoop Map-Reduce run-time system allows you to suggest the number of Map tasks an application should use. By default, this decision is based upon a heuristic for how to divide the input dataset, but it can be influenced by the application writer.

Using the WordCorrelate application with the “medium” dataset, experiment with different settings for the number of Map tasks the application requests (see the `main()` method).

- What impact does varying the number of Map tasks have on application performance?
- Why does increasing the number of Map tasks change performance?
- Assuming the medium dataset (approximately 150 MB), what will be the most significant bottleneck as the number of Map tasks is increased asymptotically?

### Dataset Size

Run DigramCount for each dataset size. Following each run, inspect its job history by typing:

```
hadoop job -history /users/${USER}/out
```

- How does performance (in terms of map-input-records per second) scale as the dataset is increased? Why does the performance change?
- Consider a hypothetical dataset that is orders of magnitude larger than the “large” dataset we provide. Consider the performance it could obtain as a function of the number of map tasks used. What ultimately limits the performance in processing gigantic datasets in high-performance clusters?

### Combiner Classes

Combiner classes in Hadoop are used to perform local reductions immediately after a Map. The `main()` method of DigramCount specifies a Combiner class. With every dataset except for the “large” dataset, record the performance of your DigramCount application with and without a Combiner class specified.

- What impact does the presence of a Combiner class have on performance?
- Broadly, does having a Combiner class increase, decrease, or not affect the amount of computation that an application has to perform?
- What scarce resource does a Combiner class best conserve? Using the output of the Map-Reduce run-time system, quantify the resources saved by the presence of a Combiner class.

## Hadoop Tips and Tricks

### Filesystem

The input data for this assignment is stored in a Hadoop-custom filesystem called HDFS. Unfortunately, we don't have a terribly convenient way for you to browse this filesystem. You may, however, use the "hadoop" command to navigate the HDFS space. For example, to see the contents of your HDFS home directory, type:

```
hadoop fs -ls /users/${USER}
```

Type "hadoop fs" to see a list of filesystem actions you can take from the command line.

### Orphaned Jobs

It is possible for Hadoop jobs to become orphaned and run aimlessly on the cluster without you being aware. This can happen, for example, if you press Ctrl-C while a job is running. You can check for orphaned jobs by typing "hadoop job -list". Type "make kill" to kill your orphaned jobs. The Makefile we provide is designed to check for orphaned jobs before running a new job to try to mitigate this problem, but you should be diligent about cleaning up your own orphans. If you see a legion of zombie jobs, please e-mail us!

### Job Status and History

Hadoop records copious amounts of data about jobs after they have completed. Much of this data may be found in your "out" directory. This data can also be summarized by typing "hadoop job -history /users/\${USER}/out".

Finally, there is a nice web interface for seeing this data. If you are on campus, point your browser to <http://katri:50030/>. This data may help you to answer the questions in the Sensitivity Analysis portion of this assignment.

### Misc.

If the cluster crashes or becomes inoperable for any reason, please send e-mail to [ee282-aut0809-staff@lists.stanford.edu](mailto:ee282-aut0809-staff@lists.stanford.edu) ASAP. For other issues, please use the bulletin board.

## SUBMISSION

The final submission will be electronic. You must submit your code (“\*.java”), your Makefile, a GROUP file, and your written report (as a PDF named “report.pdf”).

The GROUP file (named “GROUP”) will list the SUNet IDs of your group’s members, one per line. This is the login name that you use for Axxess and other Stanford web sites, and not your numeric ID number.

The report should be short (3-4 pages), and include:

1. Answers to questions from the Sensitivity Analysis section of this assignment. This is the most important part of this assignment. We expect 1 to 1.5 pages per topic.
2. Correctness results for each Map-Reduce application.
3. Your favorite random sentence. (Not graded)

You are to submit your assignment on the *Leland* cluster (elaine, myth, bramble, hedge, etc.).

First, copy your assignment, report, and GROUP file to your Leland account. For example, if you have all of the necessary files in the subdirectory “pa2”, type:

```
make clean
cd
scp -r pa2 myth:
```

Then log into Leland and use the EE282 submit script:

```
ssh myth
cd pa2
/afs/ir/class/ee282/bin/submit
```

Follow the instructions in the submit script. If all goes well, you will get a “SUCCESS” message. You may submit your assignment multiple times before the deadline. We will grade the last submission. If you have problems, please post a message on the bulletin board or send e-mail to us.

## ADDITIONAL RESOURCES

1. Google!
2. Hadoop documentation, <http://hadoop.apache.org/core/docs/r0.18.1/>
3. Java documentation, <http://java.sun.com/javase/6/docs/api/>
4. MIT Map-Reduce Seminar, <http://mr.iap.2008.googlepages.com/home>
5. University of Maryland, CMSC433 Fall 2007, Homework 5, <http://www.cs.umd.edu/class/fall2007/cmssc433/homeworks/h5/hw5.pdf>