Assignment 1
Introduction to the Hadoop Environment

Elements:
1. Get the tools you need to complete Hadoop activities
2. Run and understand a word counting program
3. Design, implement, and understand an inverted indexing program
4. Submit your code and a short write-up to demonstrate that you completed this successfully.

DUE DATE: October 14, 2008 at 4:30pm (just before class starts)

All code and the writeup must be submitted electronically via "turnin." Late assignments will be penalized at the rate of 20% per day. Code that does not compile will receive an automatic 0%.

Download Hadoop:

If you have not already done so, download Hadoop version 18 from http://hadoop.apache.org/core/

Direct link for 18.1:
http://mirrors.isc.org/pub/apache/hadoop/core/hadoop-0.18.1/hadoop-0.18.1.tar.gz

This will download as a .tar.gz file named hadoop-0.18.1.tar.gz. Unzip this someplace on your computer (e.g., C:\hadoop). Remember where you put it. We'll call this directory $HADOOP.

Even if you do not run Hadoop locally, you will need to have a copy of the .jar so that you can compile against it.

Setting up Eclipse:

If you'd like to use a GUI to edit your Java source, we recommend Eclipse. We will not be executing our programs from within Eclipse, but you can use it for editing. Download at www.eclipse.org.

Getting Cluster Access:

Step 1: Acquire credentials from Cloud Administration to access the cluster

Using a web browser, navigate to http://www.cs.washington.edu/lab/facilities/hadoop.html and follow the instructions on that page to create your user account.

Then, using an ssh client such as putty, log in to the cloud gateway server, **hadoop.cs.washington.edu**. Enter your username as you chose it on this site.

Your password is: pwd4cloud

You will be prompted to choose a new password; select one and enter it. It will automatically log you out immediately after you set it.

There is a second server behind the gateway which you must also reset your password on. This machine will still have the default password "pwd4cloud" attached to your name.
To set this password:
1) ssh in to hadoop.cs.washington.edu again. Use your username and your new password you set yourself
2) ssh into 10.1.133.1. We will refer to this as the "submission node."
3) Your password here is pwd4cloud. Change this password to one of your liking (for simplicity, the same one as hadoop.cs.washington.edu); it will log you out as soon as it is set.
4) log out of hadoop.cs.washington.edu by typing "exit" and hitting enter.

There are two tracks through the rest of this setup, depending on the machine you're working on:

- If you have your **own machine** that you can administer (Linux/OSX: you have "root" access; Windows: you are an "Administrator" of your machine), then the following instructions will allow your machine to connect directly to the cluster. Perform the "self-administered machine" steps as well as all "common" steps.
- If you are using a CSE Undergrad Lab machine or other machine which you do not administer, then skip to step 4. Perform all "common" steps, as well as all "lab machine-only" steps.

(It is a good idea to read all the steps regardless of where you're computing from.)

Self-Administered Machine Step 2: Configure hadoop-site

Hadoop uses a configuration file named hadoop-site.xml to describe which cluster to connect to. We have prepared a file for you to use to connect to the UW cluster. Download it from the link in:

http://www.cs.washington.edu/education/courses/cse490h/CurrentQtr/hadoop.htm

(see the "Config File" section)

Open this file in a text editor (notepad, vim, etc). In the section:

```
<property>
   <name>hadoop.job.ugi</name>
   <value>YOUR_USER_NAME,YOUR_USER_NAME,students,users</value>
 </property>
```

Replace YOUR_USER_NAME with theusername you selected for the Hadoop cluster. This does not need to be the same as your local machine's username.

If you are running your own services or doing something else fancy on port 2600, change uw.gateway.port to another value between 1025 and 32765. If you don't think this applies to you, it probably doesn't. Everywhere the number '2600' appears below, replace it with whatever you chose here.

Go to the $HADOOP/conf directory and replace the existing hadoop-site.xml file with the version you just edited.

Self-Administered Machine Step 3: Configure hosts File

To connect directly to the Hadoop cluster, your machine will need to know the DNS name of one of the machines behind the firewall. You must manually add this DNS name to your computer's settings.

Linux users: As root, edit the file /etc/hosts ('sudo vi /etc/hosts')
OSX users: As root, edit the file /private/etc/hosts ('sudo vi /private/etc/hosts')

Windows users:
Edit the file C:\WINDOWS\system32\drivers\etc\hosts with an Administrator account

In all cases, create the file if it does not already exist. Just use a regular text editor (vim, notepad, emacs, pico, etc..) for this purpose. Add the following lines to the hosts file:

```
10.1.133.0 XenHost-000D601B547A-2
10.1.133.0 XenHost-000D601B547A-2.internal
```

Save and close the file.

Common Setup Step 4: Start SOCKS Proxy

Users of lab machines resume the process here

Our cluster, in the 10.1.133.X network space, is not directly accessible; we must access it through a SOCKS proxy connected to the gateway node, hadoop.cs.washington.edu. You must now configure a proxy connection to allow you to make this connection.

If you have **cygwin** installed (or are using OSX/Linux), open a cygwin terminal, and type:
ssh -D 2600 *username*@hadoop.cs.washington.edu
Replacing *username* with your login name. When prompted for your password, enter it. You will see an ssh session to this node. You will not use the ssh session directly -- just minimize the window. (It is forwarding traffic over port 2600 in the background.)

If you are a Windows user and have **putty** installed, start putty, and in the "PuTTY Configuration" window, go to Connection -- SSH -- Tunnels. Type "2600" in the "Source Port" box, click "Dynamic," then click "Add." An entry for "D 2600" should appear in the Forwarded Ports panel. Go back to the Session category, and type **hadoop.cs.washington.edu** in the Host Name box. Log in with your username and password. When this has logged in, you do not need to do anything else with this window; just minimize it, and it will forward SOCKS traffic in the background.

If you are using Windows and do not have an ssh client, download PuTTY. It is free.

Download putty.exe from this page:

http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

You need to set up a proxy ssh connection any time you need to access the web interface (see steps 5 and 6), or if you are trying to connect directly to the cluster from your own machine.

Common Setup Step 5: Configure FoxyProxy so that you can see the system state

The Hadoop system will expose information about its health, current load, etc, on two web services that are hosted behind the firewall. You can view this information, as well as browse the DFS and read the error log files from services and jobs via this interface.

You must use Firefox to access these sites.

Download the FoxyProxy Firefox extension at:
http://foxyproxy.mozdev.org/downloads.html

Install the extension and restart FireFox. If it prompts you to configure FoxyProxy, click "yes." If not, go to Tools * FoxyProxy * Options.

Set the "Mode" to "Use proxies based on their pre-defined patterns and priorities"
In the "Proxies" tab, click "Add New Proxy"

- Make sure "Enabled" is checked
- Proxy Name: "UW Hadoop" (or something reasonable)
- Under "Proxy Details," select Manual Proxy Configuration.
    - hostname: localhost.
    - port: 2600
    - SOCKS proxy? should be checked
    - Select the radio button for SOCKS v5
- Under URL Patterns, click Add New Pattern
    - Pattern name: UW Private IPs:
    - URL Pattern: http://10.1.133.*:*/*
    - Select "whitelist" and "Wildcards"
- Click Add New Pattern again
    - Pattern name: xenhosts
    - URL pattern: http://xenhost-*:*/*
    - Select "whitelist" and "Wildcards"

In the "Global Settings" tab of the top-level FoxyProxy Options window, select "Use SOCKS proxy for DNS lookups"
Click "OK" to exit all the options.

You will now be able to surf the web regularly, while still redirecting the appropriate traffic through the SOCKS proxy to access cluster information.

Common Setup Step 6: Test FoxyProxy configuration

Visit the URL http://10.1.133.0:50070/
It should show you information about the state of the DFS as a whole. This lets you know

that the "10.1.133..." rule was set up correctly. Click on one of the "XenHost-???" links in the middle of the page. If it takes you to a view of the DFS, you have set up the "xenhosts" patterns correctly. Hurray!

Lab-Machine Only Step 7: Log in to the submission node

If you are using a lab machine, then you do not have the permissions to edit the hosts file, which is necessary for a direct connection to the Hadoop service. Therefore, you will need to perform all your actual Hadoop instructions on the **submission node**, which is configured to talk directly to the Hadoop system.

Log in to *username*@hadoop.cs.washington.edu. Use the password you set earlier.
From there, log in to the submission node: *username*@10.1.133.1. Use the (second) password you set earlier.

You are now logged in to a machine where you can run Hadoop. Hadoop is installed in:
    /hadoop/hadoop-0.18.2-dev
We will refer to this directory as $HADOOP, below.

Common Setup Step 8: Test your connection to Hadoop itself by browsing the DFS

Open up a terminal (Windows users: cygwin window).
Change to the $HADOOP directory where you installed Hadoop (This is /hadoop/hadoop-0.18.2-dev on the submission node)

Type the following command (assuming "$" is the prompt):

```
$ bin/hadoop dfs -ls /
```

It should return a listing similar to this:

```
Found 2 items
drwxr-xr-x   - hadoop supergroup          0 2008-10-01 17:56 /tmp
drwxr-xr-x   - hadoop supergroup          0 2008-10-02 05:10 /user
```

Troubleshooting tips (Self-administered machine):

- When running the command above if you see the following error: "JAVA_HOME is not set" you need to make sure you have a java jdk version 1.5.x or higher installed. Then you need to set the JAVA_HOME environment variable to point to the place where the jdk is installed. Use the following command with the actual path. (ideally you can put it in your .bashrc file):

```
$ export JAVA_HOME=/path/to/your/jdk
```

- If running under cygwin you encounter the following message (or something similar):

"cygpath: cannot create short name of c:\hadoop\hadoop-0.18.1\logs"
Just create the directory $HADOOP\logs

Hurray! You are now able to directly access the cluster.

In future sessions where you want to execute commands on the Hadoop server, make sure you perform step 7 (logging in to submission node) if you are operating on a lab machine, and step 4 (set up local SOCKS proxy) if you need to view log files or server status.

The remainder of this document describes the actual body of the assignment.

Counting Words With Hadoop

Hadoop excels at processing line-oriented data (i.e., plain text files). A straight-forward example of how to use this is to count occurrences of words in files.

An example:
Suppose we had two files named "A.txt" and "B.txt". Their contents are below:

```
A.txt
--------------
This is the A file, it has words in it

B.txt
--------------
Welcome to the B file, it has words too
```

The algorithm we use to count the words must fit in MapReduce. We will implement the following pseudo-code algorithm:

Mapper: takes as input a line from a document

```
foreach word w in line:
    emit (word, 1)
```

Reducer: takes as input a key (word) and a set of values (all of which will be "1")

```
sum = 0
foreach v in values:
    sum = sum + v
emit (word, sum)
```

The mappers are given all of the lines from each of the files, one line at a time. We break them apart into words and emit (word, 1) pairs -- indicating that at that instant, we saw a given word once. The reducer then collects all of the "1"s arranged by common words, and sums them up into a final count for the word.

So the mapper outputs will be:

Mapper for A.txt:
< This, 1 >
< is, 1 >
< the, 1 >
< A, 1 >
< file, 1 >
< it, 1 >
< has, 1 >
< words, 1 >
< in, 1 >
< it, 1 >

Mapper for B.txt:
< Welcome, 1 >
< to, 1 >
< the, 1 >
< B, 1 >
< file, 1 >
< it, 1 >
< has, 1 >
< words, 1 >
< too, 1 >

Each word will have its own reduce process. The reducer for the word "it" will see:
key="it"; values=iterator[1, 1, 1]
as its input, and will emit as output:
< it, 3 >
as we expect.

Running the Word Count Example

Given an input text, WortCount uses Hadoop to produce a summary of the number of words in each of several documents.

The Hadoop code below for the word counter is actually pretty short. It is presented in this document.

Note that the reducer is associative and commutative: it can be composed with itself with no adverse effects. To lower the amount of data transfered from mapper to reducer nodes, the Reduce class is also used as the *combiner* for this task. All the 1's for a word on a single machine will be summed into a subcount; the subcount is sent to the reducer instead of the list of 1's it represents.

Step 1: Create a new project

Click File * New * Project
Select "Java Project" from the list
Give the project a name of your choosing (e.g., "WordCountExample")

Then click "Next." You should now be on the "Java Settings" screen. Click the "Libraries" tab.
Now click "Add External Jars"
In the dialog box, navigate to where you unzipped Hadoop ($HADOOP). Select hadoop-0.18.1-core.jar. Click Ok.
Click "Add External Jars" again.
Navigate to $HADOOP again (it should already be there), and then to the "lib" subdirectory. Select all of the jar files in the directory and click Ok.

When you are ready, click "Finish." It will now create the project and you can compile against Hadoop.

Step 2: Create the Mapper Class

The listings below put all the classes in a package named 'wordCount.' To add the package, in the Project Explorer, right click on your project name and then click New * Package. Give this package a name of your choosing (e.g., 'wordCount')

Now in the Project Explorer, right click on your package, then click New * Class
Give this class a name of your choosing (e.g., WordCountMapper)

Word Count Map:

A Java Mapper class is defined in terms of its input and intermediate <key, value> pairs. To declare one, subclass from MapReduceBase and implement the Mapper interface. The Mapper interface provides a single method: public void map(LongWritable key, Text value, OutputCollector output, Reporter reporter). The map function takes four parameters which in this example correspond to:

1. LongWritable key - the byte-offset of the current line in the file
2. Text value - the line from the file
3. OutputCollector - output - this has the .collect() method to output a <key, value> pair
4. Reporter reporter - allows us to retrieve some information about the job (like the current filename)

The Hadoop system divides the (large) input data set into logical "records" and then calls map() once for each record. How much data constitutes a record depends on the input data type; For text files, a record is a single line of text. The main method is responsible for setting output key and value types.

Since in this example we want to output <word, 1> pairs, the output key type will be Text (a basic string wrapper, with UTF8 support), and the output value type will be IntWritable (a serializable integer class). (These are wrappers around String and Integer designed for compatibility with Hadoop.)

For the word counting problem, the map code takes in a line of text and for each word in the line outputs a string/integer key/value pair: <word, 1>.
The Map code below accomplishes that by...

1. Parsing each word out of value. For the parsing, the code delegates to a utility StringTokenizer object that implements hasMoreTokens() and nextToken() to iterate through the tokens.
2. Calling output.collect(word, value) to output a <key, value> pair for each word.

Listing 1: The word counter Mapper class:

```
package wordCount;

import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

public class WordCountMapper extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable> {

    private final IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value,
            OutputCollector<Text, IntWritable> output, Reporter
reporter) throws IOException {

        String line = value.toString();
        StringTokenizer itr = new
StringTokenizer(line.toLowerCase());
        while(itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
        }
    }
}
```

When run on many machines, each mapper gets part of the input -- so for example with 100 Gigabytes of data on 200 mappers, each mapper would get roughly its own 500 Megabytes of data to go through. On a single mapper, map() is called going through the data in its natural order, from start to finish. The Map phase outputs <key, value> pairs, but what data makes up the key and value is totally up to the Mapper code. In this case, the Mapper uses each word as a key, so the reduction below ends up with pairs grouped by word. We could instead have chosen to use the word-length as the key, in which case the data in the reduce phase would have been grouped by the lengths of the words being counted. In fact, the map() code is not required to call output.collect() at all. It may have its own logic to prune out data simply by omitting collect. Pruning things in the Mapper is efficient, since it is highly parallel, and already has the data in memory. By shrinking its output, we shrink the expense of organizing and moving the data in preparation for the Reduce phase.

Step 3: Create the Reducer Class

In the Package Explorer, perform the same process as before to add a new class. This time, add a class named "WordCountReducer".

Defining a Reducer is just as easy. Subclass MapReduceBase and implement the Reducer interface: public void reduce(Text key, Iterator<IntWritable> values, OutputCollector output, Reporter reporter). The reduce() method is called once for each key; the values parameter contains all of the values for that key. The Reduce code looks at all the values and then outputs a single "summary" value. Given all the values for the key, the Reduce code typically iterates over all the values and either concats the values together in some way to make a large summary object, or combines and reduces the values in some way to yield a short summary value.

The reduce() method produces its final value in the same manner as map() did, by calling output.collect(key, summary). In this way, the Reduce specifies the final output value for the (possibly new) key. It is important to note that when running over text files, the input key is the byte-offset within the file. If the key is propagated to the output, even for an identity map/reduce, the file will be filed with the offset values. Not only does this use up a lot of space, but successive operations on this file will have to eliminate them. For text files, make sure you don't output the key unless you need it (be careful with the IdentityMapper and IdentityReducer).

Word Count Reduce:

The word count Reducer takes in all the <word, 1> key/value pairs output by the Mapper for a single word. For example, if the word "foo" appears 4 times in our input corpus, the pairs look like: <foo, 1>, <foo, 1>, <foo, 1>, <foo, 1>. Given all those <key, value> pairs, the reduce outputs a single integer value. For the word counting problem, we simply add all the 1's together into a final sum and emit that: <foo, 4>.

Listing 2: The word counter Reducer class:

```
package wordCount;

import java.io.IOException;
import java.util.Iterator;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;

public class WordCountReducer extends MapReduceBase implements
Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values,
            OutputCollector<Text, IntWritable> output, Reporter
reporter) throws IOException {

        int sum = 0;
        while (values.hasNext()) {
            IntWritable value = (IntWritable) values.next();
            sum += value.get(); // process value
        }
        output.collect(key, new IntWritable(sum));
    }
}
```

Step 4: Create the Driver Program

You now require a final class to tie it all together, which provides the main() function for the program. Using the same process as before, add a class to your program named "WordCount".

Given the Mapper and Reducer code, the short main() below starts the Map-Reduction running. The Hadoop system picks up a bunch of values from the command line on its own, and then the main() also specifies a few key parameters of the problem in the JobConf object, such as what Map and Reduce classes to use and the format of the input and output files. Other parameters, i.e. the number of machines to use, are optional and the system will determine good values for them if not specified.

Listing 3: The driver class:

```
package wordCount;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;

public class WordCount {
    public static void main(String[] args) {
        JobClient client = new JobClient();
        JobConf conf = new JobConf(wordCount.WordCount.class);

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);

        // input and output directories (not files)
        FileInputFormat.addInputPath(conf, new Path("input"));
        FileOutputFormat.setOutputPath(conf, new Path("output"));

        conf.setMapperClass(wordCount.WordCountMapper.class);
        conf.setReducerClass(wordCount.WordCountReducer.class);
        conf.setCombinerClass(wordCount.WordCountReducer.class);

        client.setConf(conf);
        try {
            JobClient.runJob(conf);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

You will note that in addition to the Mapper and Reducer, we have also set the **combiner class** to be WordCountReducer. Since addition (the reduction operation) is commutative and associative, we can perform a "local reduce" on the outputs produced by a single Mapper, before the intermediate values are shuffled (expensive I/O) to the Reducers. e.g., if machine A emits <foo, 1>, <foo, 1> and machine B emits <foo, 1>, a Combiner can be executed on machine A, which emits <foo, 2>. This value, along with the <foo, 1> from machine B will both go to the Reducer node -- we have now saved bandwidth but preserved the computation. (This is why our reducer actually reads the value out of its input, instead of simply assuming the value is 1.)

Step 5: Run the Program Locally

We can test this program locally before running it on the cluster. This is the fastest and most convenient way to debug. You can use the debugger integrated into Eclipse, and use files on your local file system.

Before we run the task, we must provide it with some data to run on. In your project workspace, (e.g., c:\path\to\workspace\projectname) create a directory named "input".(You can create directory in the Project Explorer by right clicking the project and selecting New * Folder, then naming it "input.") Put some text files of your choosing or creation in there. In a pinch, a copy of the source code you just wrote will work.

Now run the program in Eclipse. Right click the driver class in the Project Explorer, click "Run As..." and then "Java Application."

The console should display progress information on how the process runs. When it is done, right click the project in the Project Explorer and click Refresh (or press F5). You should see a new folder named "output" with one entry named "part-00000". Open this file to see the output of the MapReduce job.

If you are going to re-run the program, you must delete the output folder (right click it, then click Delete) first, or you will receive an error.

If you receive the following error message:

```
08/10/02 14:56:14 INFO mapred.MapTask: io.sort.mb = 100
08/10/02 14:56:14 WARN mapred.LocalJobRunner: job_local_0001
java.lang.OutOfMemoryError: Java heap space
    at
org.apache.hadoop.mapred.MapTask$MapOutputBuffer.<init>(MapTask.java:371)
    at org.apache.hadoop.mapred.MapTask.run(MapTask.java:193)
    at
org.apache.hadoop.mapred.LocalJobRunner$Job.run(LocalJobRunner.java:157)
java.io.IOException: Job failed!
    at
org.apache.hadoop.mapred.JobClient.runJob(JobClient.java:1113)
    at wordCount.WordCount.main(WordCount.java:30)
```

... then add the following line to your main() method, somewhere before the call to runJob():

```
conf.set("io.sort.mb", "10");
```

and run the program again. Remember to remove or comment this line out before running on the cluster. An equally valid alternative approach is to edit the "Run Configuration" used by Eclipse. In the "VM Arguments" box, add the argument "-Xmx512m" to give it more memory to use.

When run correctly, you should have an output file with lines each containing a word followed by its frequency.

Step 6: Upload Data to the Cluster

Now that we are confident that our program runs correctly on a small amount of data, we are ready to run it on the cluster.

We must upload some files to the cluster to operate on. We will do so by creating a file hierarchy on our local machine, and then upload that to the DFS.
Create a local directory somewhere named "input". In the input directory, add a few text files you would like to index.

If you are running this on a lab machine, you should then zip up the directory (use WinZip or something to make it into a .zip file), and then copy it to the cluster. (Self-administered machine users, skip down a little ways) To do this, execute the following command locally:

```
$ scp my-zip-file.zip  username@hadoop.cs.washington.edu:
```

(don't forget the ":" at the end)
This will copy the file to the gateway machine.
If you are using windows, and cygwin is unavailable, you may not have scp. Download **pscp** from the PuTTY web site: http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

Now ssh to hadoop.cs.washington.edu, and repeat this process to copy the file to 10.1.133.1:

```
$ scp my-zip-file.zip 10.1.133.1:
```

Since your username is the same on both machines, you don't need to provide that twice.

Now log in to the submission node (ssh 10.1.133.1).
Unzip the file you just created:
```
$ unzip my-zip-file.zip
```
This should create your input/ directory with all the files in it on the submission node.

Users on both self-administered machines and lab machines should **all do the following:** We must now copy the files into the Distributed File System (HDFS) itself.

In your terminal, change to the parent directory of your input directory. Execute:
```
$ /path/to/hadoop-installation/bin/hadoop dfs -copyFromLocal input input
```

Where path/to/hadoop-installation/ is "$HADOOP" referred to earlier. This will copy the local "input" folder (first argument) to the target directory (second argument) also named "input" on the cluster. This path is relative to /user/$USERNAME/ if you didn't suppy an absolute path here.

If you're using bash as your terminal (submission node users: you are), you can actually save yourself repeated typing by executing:
```
$ HADOOP=/path/to/hadoop-installation
```

from now on, in this session, you can actually write commands like:
```
$ $HADOOP/bin/hadoop dfs -copyFromLocal input input
```

and "$HADOOP" will be substituted with /path/to/hadoop-installation. The instructions from here on assume that you've done this.

Verify that your upload has completed successfully by running:
```
$ $HADOOP/bin/hadop dfs -ls .
```

It should show you a line that contains a directory named "input"
If you then run
```
$ $HADOOP/bin/hadoop dfs -ls input
```
it should show you all the files in that directory.

Step 7: Run the MapReduce Program on the Cluster

MapReduce requires your program to be compiled into a "jar" file containing all your classes. Open a command line terminal and switch to your project directory under your workspace. type dir (windows) or ls (linux/OSX). By default, you should see a "src" directory where your source files are, the "input" directory where you put your test files, and a "bin" directory where the compiled classes are.

We will jar up the bin directory with the following command:

```
$ jar cvvf wordcount.jar -C bin .
```

This will create an output jar named "wordcount.jar" that contains all your classes. It will print the names of the classes it is adding to the command line. If you configured Eclipse to co-mingle .java and .class files together in the src directory, just execute the above with "-C src" instead.

Lab machine users: scp (or pscp) the wordcount.jar file from your current machine to the gateway (hadoop.cs) and again from the gateway to the submission note (10.1.133.1).

Everyone:
To run the program, execute the following line on the command line (on your self-administered machine, or on the submission node):

```
$ $HADOOP/bin/hadoop jar wordcount.jar wordCount.WordCount
```

This will tell Hadoop to run a jar-based program. The first argument is the name of the jar file, and the next argument is the full name of the class (including the package) where it will find the main() method. Any subsequent command-line arguments are delivered to your main() method.

You should now see some progress output from running the job.

Step 8: View Output

On your locally administered machine or the submission node, execute the following:

```
$ $HADOOP/bin/hadoop dfs -ls .
```

You should see two entries: the "input" folder you created yourself, and a new "output" folder. This is the result of running your job. If you then look at the output folder:

```
$ $HADOOP/bin/hadoop dfs -ls output
```

You should see a file named part-00000. You can view a snippit of this file (the last 1 KB) by executing:

```
$ $HADOOP/bin/hadoop dfs -tail output/part-00000
```

You can copy the whole file to the (locally administered machine or) submission node by running:

$ $HADOOP/bin/hadoop dfs -copyToLocal output/part-00000 *local-file-name*

This will copy the contents into *local-file-name* in the current directory. You can view the file by typing 'less *local-file-name*'. If you're on the submission node, you probably don't want to do this there.

To retrieve the file from the submission node, first execute the following on the gateway (hadoop.cs):

```
$ scp 10.1.133.1:path/to/local-file-name gateway-file-name
```

If you were running this in your home directory, then you can omit the 'path/to/'. This will copy local-file-name on the submission node to gateway-file-name on the gateway node.

Then execute the following on your local machine:

```
$ scp username@hadoop.cs.washington.edu:gateway-file-name final-file-name
```

This will copy the file to "final-file-name" on your local machine. You can now view this in the text editor of your choice on your local machine.

Remember to delete files you leave lying around on the gateway node and submission node with 'rm', e.g.,

```
$ rm somefile
```

To recursively delete a whole directory:

```
$ rm -rf somedirectory/
```

Important: If you want to re-run your job, you must delete the output dierctory first, or else your program will exit with an error message. To delete directories in HDFS:

```
$ $HADOOP/bin/hadoop dfs -rmr dirname
```

e.g.,

```
$ $HADOOP/bin/hadoop dfs -rmr output
```

Step 9: Run on our inputs

Instead of using some files on your local machine, we have also provided a few larger text files for you. Set the input directory in the driver method to "/shared/large_texts" and re-run the program. Don't forget to delete your output directory before rerunning the job.

An Inverted Index

Given an input text, offset indexer uses Hadoop to produce an index of all the words in the text. For each word, the index has a list of all the locations where the word appears, and optionally a text excerpt of each line where the word appears. Running the line indexer on the complete works of Shakespeare, the following input lines:

lucrece.txt, offset 38624: To cipher what is writ in learned books,
orlando.txt, offset 66001: ORLANDO Which I take to be either a fool or a cipher.

would produce the following output for the word "cipher:"

```
cipher    lucrece.txt@38624,orlando.txt@66001,...
```

Step 1: Create a new project

Create a new project using the same instructions as for Word Count (step 1). Give the project a name of your choosing (e.g., "LineIndexExample"). Don't forget to add the libraries to the build path.

Step 2: Create the Mapper Class

Create a new package (named, for example "lineindex"). Then create a new class in the package.
Give this class a name of your choosing (e.g., LineIndexMapper)

Line Indexer Map:

Review: The map function takes four parameters which by default correspond to:

1. LongWritable key - the byte-offset of the current line in the file
2. Text value - the line from the file
3. OutputCollector - output - this has the .collect method to output a <key, value> pair
4. Reporter reporter - allows us to retrieve some information about the job (like the current filename)

The map function should output <"word", "filename@offset"> pairs. Despite the name of the task (Line Indexer) we will actually be referring to locations of individual words by the byte offset at which the line starts -- not the "line number" in the conventional sense. This is because the line number is actually not available to us. (We will, however be indexing a line at a time--thus the name "Line Indexer.") Large files can be broken up into smaller chunks which are passed to mappers in parallel; these chunks are broken up on the line ends nearest to specific byte boundaries. Since there is no easy correspondence between lines and bytes, a mapper over the second chunk in the file would need to have read all of the first chunk to establish its starting line number -- defeating the point of parallel processing!

Your map function should extract individual words from the input it is given, and output the word as the key, and the current filename & byte offset as the value. You need only output the byte offset where the line starts. Do not concern yourself with intra-line byte offsets.

Hints:

1. Since in this example we want to output <"word", "filename@offset"> pairs, the types will both be Text.
2. The word count program from code lab 1 had an example of extracting individual words from a line
3. To get the current filename, use the following code snippit:

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

Step 3: Create the Reducer Class

In the Package Explorer, perform the same process as before to add a new element. This time, add a class named LineIndexReducer.

Line Indexer Reduce Code:

The line indexer Reducer takes in all the <"word", "filename@offset"> key/value pairs output by the Mapper for a single word. For example, for the word "cipher", the pairs look like: <cipher, shakespeare.txt@38624>, <cipher, shakespeare.txt@12046>, <cipher, ... >. Given all those <key, value> pairs, the reduce outputs a single value string. For the line indexer problem, the strategy is simply to concat all the values together to make a single large string, using "," to separate the values. The choice of "," is arbitrary -- later code can split on the "," to recover the separate values. So for the key "cipher" the output value string will look like "shakespeare.txt@38624,shakespeare.txt@12046,shakespeare.txt@34739,...". To do this, the Reducer code simply iterates over values to get all the value strings, and concats them together into our output String.

Important: The following Java programming advice is important for the performance of your program. Java supports a very straightforward string concatenation operator:

```
String newString = s1 + s2;
```

which could just as easily be:

```
s1 = s1 + s2;
```

This will create a *new string* containing the contents of s1 and s2, and return it to the reference on the left. This is O(|s1| + |s2|). If this is performed inside a loop, it rapidly devolves into O(n^2) behavior, which will make your reducer take an inordinately long amount of time. A linear-time string append operation is supported via the **StringBuilder** class; e.g.:

```
StringBuilder sb = new StringBuilder();
sb.append(s1);
sb.append(s2);
sb.toString(); // return the fully concatenated string at the end.
```

Step 4: Create the Driver Program

You now require a final class to tie it all together, which provides the main() function for the program. Using the same process as before, add a class to serve as the driver. Name this something like "LineIndexer." It should look extremely similar to the one in WordCount, but the class names should obviously be the class names for your Mapper and Reducer.

Set the input and output types to Text.class, set the input and output paths to some of your liking. (If you use an absolute path, specify a subdirectory of "/user/$USERNAME". If you specify a relative path, it will always be relative to this base.) Finally, set it to use your Mapper and Reducer classes.

You should now run your program locally on a small amount of test input. When it is ready for larger-scale testing, upload some input data, run your program and verify that it works. Refer back to instructions for steps 5--8 from the previous part of this lab if you need help with this part. When it seems to all work, run with the input directory set to /shared/large_texts.

Extra credit:

Add a summary of the text near each instance of each word to the output.

What to Turn In:

A) All code necessary to implement LineIndexer. It should be legible and with comments.

B) A writeup in a file named "writeup.txt" (send plain text only) answering the following questions:

1) What does the word count program do with the key provided to the mapper? What does the line indexer do with this key? Explain the different usages and why they're both okay.

2) Explain in your own words the purpose of the combiner in the word counter. Does it save bandwidth? How?

Given two documents:
A.txt: This is a file and it is a perfectly good file and it is my best one
B.txt: This is a file as well and it is just as good.

List the <key, val> pairs associated with the word "is" that would be generated by the word count mapper. Then list the <k, v> pairs that would be emitted by the combiners (assume the documents are processed on separate nodes), and by the reducer.

3) Should you be using a combiner in line indexer? Why or why not? (Explain either how it helps, or why it wouldn't.)

4) How many instances of the word "sword" does the word counter find when run on the large_texts corpus?

5) List at least 3 filenames and offsets for the word "sword" found by the line indexer.

6) Did you write line indexer from scratch, or did you modify word counter? If the former, describe your thought process as you designed the code. If the latter, tell us what parts of it you had to change.

7) If you did the extra credit, mention it here to make sure we notice :) Describe what changes you had to make over the baseline implementation.

Hints and Reminders:

- You don't need to always run on the cluster. For small debugging inputs, you can run on your local machine, using your local file system. Just launch the program the normal way in Eclipse (hit the "play" button).

- This will allow you to see System.out.println()'s etc. Anything the mappers print on the cluster will not be sent back to you.
- This will also let you use the local Java debugger, since it will run everything in a single process. You can then set breakpoints in your Mapper and Reducer code.
- Use a small test input before moving to larger inputs
- If your job takes more than 10 minutes to run on the cluster, you've done something wrong. Please kill your job to allow other students to use the resources.
- You can check cluster JobTracker status by viewing the web page at http://10.1.133.0:50030/
- If your job on the cluster doesn't work right, go back to running on your local machine to get everything working before you move back to the cluster. **Running buggy code on the cluster may prevent other students from running anything.**
- You must always delete your output directory on the cluster before starting the job. If there's a previous output directory present, your job will fail to launch.
- The API reference is at http://hadoop.apache.org/core/docs/r0.18.1/api/index.html -- when in doubt, look it up here first!
- If either your mapper or reducer is > 50 lines of code, you're overthinking the problem.

IMPORTANT: How to Kill Jobs

If you run a job which starts "running away" (e.g., running for more than 5 or ten minutes), you should kill it to prevent it from blocking other students.

From your locally administered-machine or the submission node, run:
```
$ $HADOOP/bin/hadoop job -list
```

This will list all running jobs. The left-most column will be the job name (which is usually a combination of "job_", a timestamp and a counter). The right-most column will be the username of the launching user. Find your job by your username, then execute:
```
$ $HADOOP/bin/hadoop job -kill job_id
```

Where *job_id* is whatever was the left-most column for that row.

Note: Don't forget, jobs are enqueued for a while before they run. You should check to make sure your job is actually "running" as opposed to "pending" (e.g., just waiting for someone else's job to finish up first). Jobs are enqueud in a FIFO (first-in, first-out) fashion. If you are behind a lot of other students, you might be waiting a while.

Moral of the story: Do not wait until the last minute! You might find that there's a 2 hour (or longer!) delay before your job runs. **We will not be granting extensions** based on the excuse that "my job didn't run because it was enqueued behind lots of other people." It is your responsibility to start projects early enough that this is not an issue. Careful debugging on your local machine (in the lab or at home) should ensure that you only need to run on the cluster once or twice. Using the cluster itself for your debugging runs is a great way to waste your time and probably turn things in late.