Codelab 1
Introduction to the Hadoop Environment (version 0.17.0)

**Goals:**
1. Set up and familiarize yourself with the Eclipse plugin
2. Run and understand a word counting program

**Setting up Eclipse:**

**Step 1: Install Eclipse and the IBM MapReduce Plugin**
Download the installable packages and install them according to the instructions in the README file.
http://net.pku.edu.cn/~course/cs402/2008/resource/cc_software/client/

**Step 2: Start Eclipse**
**Step 3: Change Perspective**
You should now have a new "perspective" available. In the upper-right corner of the workspace, you will see the current perspective, probably set to Java.

(Fig 1: Java perspective open)

Click the "Other perspective" button, which is circled in the figure above, then select "Other" from the list. If installed correctly, "MapReduce" will be an option in the window which pops up. You will then see the MapReduce perspective open, as signified by the corner changing to look like so:

(Fig 2: MapReduce perspective open)

**Step 4: Configure a new Hadoop server**
This perspective will add a new snap-in to your bottom pane (along with Problems and Tasks), like so:

(Fig 3: Servers pane available)

If this pane does not appear, go to Window * Show View * Other * MapReduce Tools *MapReduce Servers.
This view creates two (identical, unfortunately) elephant-shaped buttons. The tool
tips for these buttons will say "Edit Server" and "New Server."
Click "New Server" (circled in figure 3)

You can then add the name and location of a hadoop server to the configuration. The
fields are as follows:

Server name: Any descriptive name you want. e.g., "Hadoop Server"
Hostname: The host name or IP address of the JobTracker node
Installation directory: The path to the location of hadoop-0.XX.Y-core.jar on the
JobTracker node.
Username: Your account name on this machine

(Fig 4: An example of the settings used on my system. After validating the location,
the circled information note appears)

The settings we will be using are:
Hostname: 222.29.154.22
Installation directory: /POOL_TEMP_space/SUR/hadoopforCourse
Your username and password for the hadoop server itself are on another sheet of
paper.
After you enter the appropriate values, click the "Validate location" button to allow Eclipse to confirm that you have entered everything correctly. If so, it will inform you of the Hadoop version it finds at that location. You will be prompted to enter your UNIX password to allow the login process to continue.

When you click "Finish," the server should be visible in the Servers pane. In the "Project Explorer" view, you should now be able to browse the Hadoop DFS.

![Project Explorer](image)

(Fig 5: DFS visible)

You can now take a moment and explore the DFS. If you try to upload anything, you should place all your files in /user/$USERNAME.

**Running the Word Count Example**

Given an input text, WordCount 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. The Map code extracts words one at a time from the input and maps each of them to a counter '1' that indicates that the word was seen once. The Reduce code sums all the data (a long list of '1's) for each word, giving the total count.

Note that the reducer is associative and commutative: it can be composed with itself with no adverse effects. To lower the amount of data transferred 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**

If you have not already done so, switch to the MapReduce perspective (see the tutorial above).
Click File * New * Project
Select "MapReduce Project" from the list
Give the project a name of your choosing (e.g., "WordCount")
There is an entry spot for the MapReduce Library Installation Path. Click "Configure
Hadoop install directory", and navigate to the path to your local hadoop installation. When you are ready, click "Finish."

If you need to change Java Compiler, in the Project Explorer, right click on the top-level project for "WordCountExample," go to Properties * Java Compiler. Click "Enable project specific settings," then set the compiler compliance level to "x.0."

Step 2: Create the Mapper Class
In the Project Explorer, right click on your project name, then click New * Other Under MapReduce, select "Mapper."
Give this class a name of your choosing (e.g., Map)

Word Count Map:

A Java Mapper class is defined in terms of its input and intermediate <key, value> pairs. To declare one, simply subclass from MapReduceBase and implement the Mapper interface. The Mapper interface provides a single method:

```java
public void map(WritableComparable key, Writable value,
                OutputCollector<Text, IntWritable> output, Reporter reporter).
```

The map function takes four parameters which in this example correspond to:
1. WritableComparable key - the byte-offset of the current line in the file
2. Writable value - the line from the file
3. OutputCollector<Text IntWritable> 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). It is necessary to wrap the more basic types (e.g., String, Int) because all input and output types for Hadoop must implement WritableComparable, which handles the writing and reading from disk. (More precisely: key types must implement WritableComparable, and value types must implement Writable. Text, IntWritable, and other box classes provided by Hadoop implement both of these.)

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:

```java
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class Map extends MapReduceBase implements Mapper<WritableComparable, Writable, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(WritableComparable key, Writable 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 element. This time, add a "Reducer" named "Reduce".

Defining a Reducer is just as easy. Simply subclass MapReduceBase and implement the Reducer interface: `public void reduce(WritableComparable key, Iterator<Writable> values, OutputCollector<Text, IntWritable> 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:

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

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
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 Reduce extends MapReduceBase implements Reducer<WritableComparable, Writable, Text, IntWritable> {
    public void reduce(WritableComparable _key, Iterator<Writable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
        // replace KeyType with the real type of your key
        Text key = (Text) _key;

        int sum = 0;
        while (values.hasNext()) {
            // replace ValueType with the real type of your value
            IntWritable value = (IntWritable) values.next();

            // process value
            sum += value.get();
        }
        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 "MapReduce Driver" object to your program named "WordCount". This will include a skeleton main() function that includes stubs for you to set up the MapReduce job.

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:

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

public class WordCount {
    public static void main(String[] args) throws Exception {
        JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);

        FileInputFormat setInputPaths(conf, new Path("input");
        FileOutputFormat setOutputPath(conf, new Path("output");

        // specify a mapper
        conf.setMapperClass(Map.class);

        // specify a reducer
        conf.setCombinerClass(Reduce.class);
        conf.setReducerClass(Reduce.class);
        JobClient.runJob(conf);
    }
}

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 A MapReduce Task
Before we run the task, we must provide it with some data to run on.

(We have preinstalled needed data for all labs, the path is /data/. You can use them directly without upload them manually as the following instructions tell.)

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 use the provided shakespeare.tar.gz, it will inflate to a directory named "input/" containing several text files.) You can download this file at:


In the project explorer, browse the DFS to find your home directory. e.g., /user/ $USERNAME. Right click this element, and select "Import from local directory". Find your "input" directory, and add it. You should now see a tree that looks like:

```
user
    -- YourUserName
    -----input
    ------(some text files)
```

To start running your program, select the driver class in the Project Explorer. Then in the menu at the top of the screen, click Run * Run As * Run on Hadoop. Click "Choose an existing server", and select the server entry you configured earlier. Click "Finish."

You should now see some output developing in the "console" window which appears at the bottom of the screen. In the "MapReduce Servers" pane, you can also watch the progress of your job. Eventually, if all goes well, it will say "Completed."

**Step 6: View Output**

In the DFS explorer, click "Refresh" on your home directory. You should now see an "output" folder as well. There should be a file there named "part-00000". Double click the file to view it in Eclipse. You should see several lines which start with a word on the left, and then on the right are the number of occurrences for each word across the set of files.

If you want to re-run your job, remember to delete the output directory first, or else your program will exit with an error message.