Lab 0 - Introduction to Hadoop/Eclipse/Map/Reduce

CSE 490h - Winter 2007

To Do

1. Eclipse plug in introduction.
2. Read this hand out.
3. Get Eclipse set up on your machine.
4. Write the word counter example in Eclipse
5. Load a small text corpus onto the cluster.
6. Run word counter on the corpus.
7. Convince Slava you have it working.
8. That's all, folks.

This Document

1. Hadoop concepts
2. The Word Counter Example
    a. Word Counter Map & Code
    b. Word Counter Reduce & Code
    c. Word Counter Main & Code
3. Running a MapReduce on Hadoop
    a. Dfs
    b. Running a MapReduce locally
    c. Seeing job progress

Hadoop Concepts

To use Hadoop, you write two classes -- a Mapper and a Reducer. The Mapper class contains a map function, which is called once for each input and outputs any number of intermediate <key, value> pairs. What code you put in the map function depends on the problem you are trying to solve. Let's start with a short example.

Suppose the goal is to create a word count of a body of text – we are given the text files, and we want to output a list of words annotated with their counts. For that problem, an appropriate Map strategy is: for each word in the input, output the pair <word, 1>.

For example, suppose we have this five-line High school football coach quote as our input data set:

We are not what
we want to be,
but at least
we are not what
we used to be.

Running the Map code that for each word, outputs a pair <word, 1>, yielding the set of pairs...
<we, 1>
<are, 1>
<not, 1>
<what, 1>
<we, 1>
<want, 1>
<to, 1>
<be, 1>
<but, 1>
etc...

For now we can think of the <key, value> pairs as a nice linear list, but in reality, the Hadoop process runs in parallel on many machines. Each process has a little part of the overall Map input (called a map shard), and maintains its own local cache of the Map output. (For a description of how it really works, see Hadoop/MapReduce or the Google White Paper linked to at the end of this document.)

After the Map phase produces the intermediate <key, value> pairs they are efficiently and automatically grouped by key by the Hadoop system in preparation for the Reduce phase (this grouping is known as the Shuffle phase of a map-reduce). For the above example, that means all the "we" pairs are grouped together, all the "are" pairs are grouped together like this, showing each group as a line...

<we, 1> <we, 1> <we, 1> <we, 1>
<are, 1> <are, 1>
<not, 1> <not, 1>
<what, 1> <what, 1>
<want, 1>
<to, 1> <to, 1>
<be, 1> <be 1>
<but, 1>
<at, 1>
<least, 1>
<used, 1>

The Reducer class contains a reduce function, which is then called once for each key -- one reduce call for "we", one for "are", and so on. Each reduce looks at all the values for that key and outputs a "summary" value for that key in the final output. So in the above example, the reduce is called once for the "we" key, and passed the values the mapper output, 1, 1, 1, and 1 (Note that the values going into reduce are not in any particular order). Suppose reduce computes a summary value string made up of the number of values the mapper output for the given key, then the output of the Reduce phase on the above pairs will produce the pairs shown below. The Reduce phase also sorts the output <key,value> pairs into increasing order by key:

<are, 2>
<at, 1>
<be, 2>
<but, 1>
<least, 1>
<not, 2>
<to, 2>
<we, 4>
<what, 2>
<want, 1>
<used, 1>

Like Map, Reduce is also run in parallel on a group of machines. Each machine is assigned a subset of the keys to work on (known as a reduce shard), and outputs its results into a separate file.

Word Count Example

Let’s take the above map reduce algorithm and implement it on the hadoop framework. First, we are going to tackle the map portion:

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:

public void map(WriteableComparable key, Writeable value,
OutputCollector output, Reporter reporter).

Note: these inner classes probably need to be declared "static". If you get an error saying *ClassName*.<init>() is not defined, try declaring your class static. The map function takes four parameters which in this example correspond to:

- WriteableComparable key - the byte-offset
- Writeable value - the line from the file
- OutputCollector - output - this has the .collect method to output a <key, value> pair
- Reporter reporter - you can ignore this for now
-

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 types will both be Text (a basic string wrapper, with UTF8 support). It is necessary to wrap the more basic types because all input and output types for Hadoop must implement WritableComparable, which handles the writing and reading from disk.

For the word count problem, the map code takes in a line of text and for each word in the line outputs a string key/value pair <word, 1>.

The Map code below accomplishes that by...

- 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.
- For each word, calling *output.collect(word, value)* to output a <key, value> pair for each word.

```
public class wordCountMapper extends MapReduceBase implements Mapper {
      Text word         = new Text();
      Text oneText      = new Text("1");
      public void map(WritableComparable key, Writable values,
                  OutputCollector output, Reporter reporter)
                  throws IOException {
            String line = values.toString();
            StringTokenizer iter = new StringTokenizer(line);
            while(iter.hasMoreTokens()) {
                  word.set(iter.nextToken());
                  output.collect(word, oneText);
            }
      }

}
```

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 line-length as the key, in which case the data in the reduce phase would have been grouped by line length. 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.

Word Count Reduce
Defining a Reducer is just as easy. Simply subclass MapReduceBase and implement the Reducer interface:

public void reduce(WriteableComparable key, Iterator 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 propogated 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 Code
The word count Reducer takes in all the <word, 1> key/value pairs output by the Mapper for a single word. Given all those <key, value> pairs, the reduce outputs a single value string. For the word count problem, the strategy is simply to count all the values and set the summary to this number.

To do this, the Reducer code simply iterates over values and counts them.

```
public class wordCountReducer extends MapReduceBase implements Reducer
{
      Text count = new Text();
      public void reduce(WritableComparable key, Iterator values,
                  OutputCollector output, Reporter reporter)
                  throws IOException {
            int wordcount = 0;
            while (values.hasNext()) {
                  values.next();
                  wordcount++;
            }
```

```
            count.set(String.valueOf(wordcount));
            output.collect(key, (Writable)count);
      }
}
```

Word Count Main Program

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, ie. the number of machines to use, are optional and the system will determine good values for them if not specified.

Note one set of parameters in particular, *InputPath* and *OutputPath*. The *InputPath* must be a valid directory on the DFS. The map reduce will then be run over all files in that directory. The *OutputPath* must be a new directory that does not yet exist. The execution of the job will create the directory. (If the directory already exists, hadoop will abort the job)

```
public class wordCountDriver {

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

            // specify output types
            conf.setOutputKeyClass(Text.class);
            conf.setOutputValueClass(Text.class);

            // specify input and output DIRECTORIES (not files)
            // on the DFS.
            conf.setInputPath(new Path("src"));
            conf.setOutputPath(new Path("out"));

            conf.setMapperClass(wordCountMapper.class);
            conf.setNumMapTasks(10); // Number of machines for map

            conf.setReducerClass(wordCountReducer.class);
            conf.setNumReduceTasks(2); // Num of machines for reduce

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

Running A Map-Reduction Manually

To run a Hadoop job, simply ssh into any of the JobTracker nodes on the cluster. To run the job, it is first necessary to copy the input data files onto the distributed file system. If the data files are in the localInput/ directory, this is accomplished by executing:

```
./bin/hadoop dfs -put localInput dfsInput
```

The files will then be copied onto the dfs into the directory dfsInput. It is important to copy files into a well named directory that is unique. These files can be viewed with

```
./bin/hadoop dfs -ls dir
```

where dir is the name of the directory to be viewed.

You can also use

```
./bin/hadoop dfs -lsr dir
```

to recursively view the directories. Note that all "relative" paths given will be put in the

/users/$USER/[dir]

directory. Make sure that the dfsOutput directory does not already exist, as you will be presented with an error, and your job will not run (This prevents the accidental overwriting of data, but can be overridden).

Now that the data is available to all of the worker machines, the job can be executed from a local jar file:

```
./bin/hadoop jar wordCount.jar
```

The job should be run across the worker machines, copying input and intermediate data as needed. Assuming the output of the reduce stage will be left in the dfsOutput directory, copy these files to your local machine in the directory localOutput by:

```
./bin/hadoop dfs -get dfsOutput localOutput
```

Running A Map-Reduction Locally

During testing, you may want to run your Map-Reduces locally so as not to adversely affect the compute clusters.

This is easily accomplished by adding a line to the main method:
`conf.set("mapred.job.tracker", "local");`

Seeing Job Progress
When you submit your job to run a line will be printed saying:

```
Running job: job_12345
```

where 'job_12345' will correspond to whatever name your job has been given. Further status information will be printed in that terminal as the job progresses. However, it is also possible to monitor a job given its name from any node in the cluster. This is done by the command:

```
./bin/hadoop/ job -status job_12345
```

Jobs can also be killed if necessary with

```
./bin/hadoop/ job -kill job_12345
```

A small amount of status information will be displayed, along with a link to a tracking URL (eg, http://jobtrackermachinename:50030/). This page will be a job-specific status page, and provide links to main status pages for other jobs and the Hadoop cluster itself.