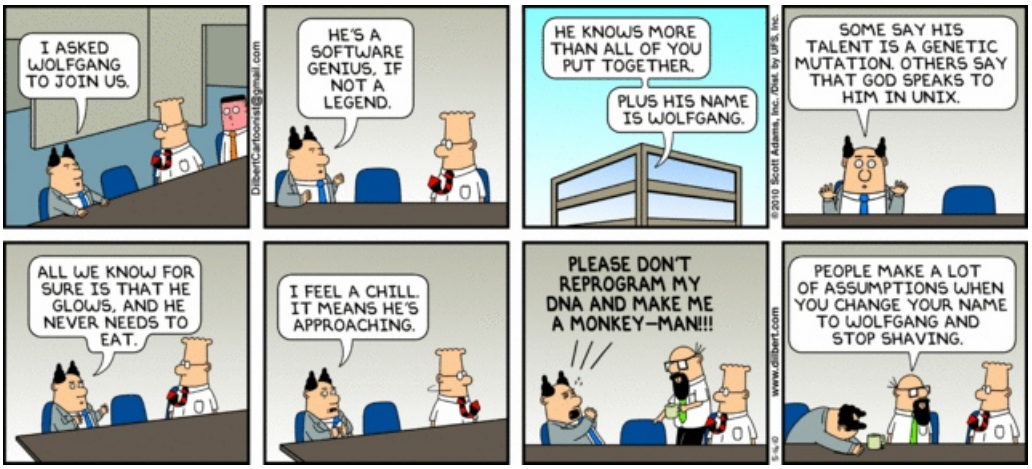
To reduce or not to reduce, *that* is the question



1 Running jobs on the Hadoop cluster

For part 1 of assignment 8, you should have gotten the word counting example from class compiling. To start with, let's walkthrough the process of running code you've written on the hadoop cluster.

1. Get the code compiled

   - **If you're working on a lab machine:**
     Eclipse already compiles the code for you, so you just need to `cd` into the `bin` directory of the project directory within the Eclipse workspace folder where the `WordCount` code resides. If you're don't have separate src and bin directories[1] then just `cd` into the base of the project directory.
   - **If you're working on your own computer:**
     * Go to the `src` directory of the project directory within the Eclipse workspace folder where the `WordCount` code resides. If you're don't have separate src and bin directories[1] then just `cd` into the base of the project directory.
     * Use `scp` or whatever approach you use to copy files to `basin` and copy all of the source files in their package directory structure (in this case, `demos`) to a directory on `basin`.

[1] There's an option to select this when you setup a new project. In general, it's a good idea to do this.

* Login to `basin` and then `cd` to the folder where you copied the source files. Compile your code from the command-line:

  ```
  javac -cp /usr/local/hadoop/hadoop-core-1.2.1.jar:. demos/*.java
  ```

  In this case, there is just one package and one file, however, in general, you will need to list all of the packages that have files you want to compile at the end of the `javac` call.

2. Create a jar file

   Hadoop requires that you generate a jar file with all of your code to run it. To create a jar file, make sure you're in the base directory where you just compiled your code (or where Eclipse compiled it for you) and type the following:

   ```
   jar -cvf demos.jar demos/*.class
   ```

   The first argument is the name of the jar file to create and then everything else following is what should go into the jar file. In general, you will need to list all of the packages that have files you want to include at the end of the call to `jar`

3. Running your code on the cluster

   Now that you have a jar file you can run your code on the cluster. When running hadoop programs you use the `jar` flag to specify the jar file that contains the code and then you specify which classes main class to run (including the package structure):

   ```
   hadoop jar demos.jar demos.WordCount
   ```

   You should see the usage statement printed out:

   ```
   WordCount <input_dir> <output_dir>
   ```

   Let's run the full job with some very simple data:

   ```
   hadoop jar demos.jar demos.WordCount /user/dkauchak/lab/ output
   ```

   This uses some data in my directory with just a couple of files as input and will output the results to your own `output` directory.

   Take a look in the `output` directory and you should see a single file called `part-00000`. If you look at the contents of this file you should see:

   ```
   !        1
   .        1
   a        1
   also     2
   and      2
   another  1
   banana   1
   but      1
   file     3
   ```

```
has      2
is       2
more     1
some     2
text     2
the      1
this     3
with     1
word     1
words    2
```

2 WordCount variations

Now that we can compile and run our hadoop programs, let's try out a few variants of our `WordCount` program.

NoOpReducer

As I mentioned in class, one of the best ways to debug your program (and to develop it incrementally) is to use a reducer that simply copies its input to the output. In the `demos` directory on the course web page I have included the `NoOpReducer.java` class. Download this file and put it into your Eclipse project.

Now, change the line in the `run` that sets the reducer to be:

```
conf.setReducerClass(NoOpReducer.class);
```

Recompile your code and then run this variant on the cluster. A few thoughts on this:

- Don't forget to delete your output directory before you run it. Otherwise, you're going to get an exception.
- Both for this lab and when you're writing your own programs, you want to make the process of compiling, copying the files, generating the jar, etc. as simple as possible. My advice is to setup a few terminal windows and designate each window as doing a few commands. You can then simply use the up-arrow to rerun these commands for each variation. For example, my setup is:
  1. one terminal session for copying over the .java files
  2. one terminal session for compiling and creating the jar file
  3. one terminal session for deleting old output directory, executing hadoop command and peeking at output
  4. one terminal session for doing more involved things with the hdfs, etc.

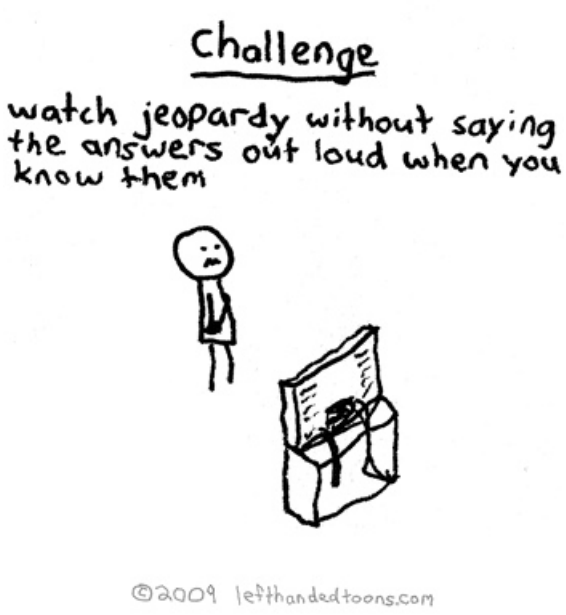
It can take a minute or two to setup, but I can go from a code change on my laptop to running on the hadoop cluster in about 10 seconds.

Assuming this runs correctly, you should see something like:

```
!       1
.       1
a       1
also    1
also    1
and     1
and     1
another 1
banana  1
but     1
file    1
file    1
file    1
has     1
has     1
is      1
is      1
more    1
some    1
some    1
text    1
text    1
the     1
this    1
this    1
this    1
with    1
word    1
words   1
words   1
```

In addition to just a map and reduce phase, hadoop also allows for a "combiner" phase. A combiner implements the reducer interface and runs in between the map phase and the reduce phase. However, there's a catch! Let's run the WordCount example with a combiner and see if you can figure out what the catch is.

Uncomment the line `conf.setCombinerClass(WordCountReducer.class);` in `WordCount.java`, recompile, etc. and then rerun on the hadoop cluster. What is the combiner doing?



(Really think about it for a minute! Scroll down when you're ready for the answer.)

The combiner phase is often referred to as a "mini-reduce" phase. It does run a full reduce (notice above we just used our `WordCountReducer`), however, it only runs it locally on the same machine that the map process was run. This means that it only reduces on some of the data. Why might this be useful? Why is this useful for the word count example?

3 Inverted Index

Also in the demos directory online I have included another of the classic MapReduce examples called `LineIndexer`. Download this file into the demos package in your Eclipse workspace and look through the code to see if you can figure out what it does.

The `LineIndexer` example includes one thing we haven't seen before: the use of the `Reporter` object. Here we're using it to get the split that we're working on and then using that to get the particular filename.

Once you think you've figure out how it works, set it up to run. To run this program, you'll need to change your call to hadoop to:

```
hadoop jar demos.jar demos.LineIndexer
```

Look at the output. Did it do what you expected?

This function generates what is called an inverted index, that is a mapping from words to the documents that they occur in. This is probably one of the most important structures that enables web search (and many "index-based" search techniques). This was one of the early motivating examples that led to the development of the MapReduce framework by Google.

A few things to think about:

- Could we use the `LineIndexerReducer` as a combiner for this example? Try it out and see if you're right. If you don't think you can, you should either get the wrong answer, or more likely, an error. If you do think you can, you should get the same output as before.
- In this example I have used a `HashSet` to filter out repeated entries. While this works, it does require populating and clearing the hash set each time reduce is called. If the list of documents that a word occurs in gets very large, this can be problematic (think web scale).

  Can you think of an alternative using the MapReduce framework (*Hint: it involves using a two phase MapReduce.*)?

4 grep

`grep` is a great command-line utility that allows you to search for text (and even regular expressions) in files. If the text is found in a line in the file (or the regex matches), that entire line is printed out. `grep` is very fast, however, if you had a very large file (or a number of very large files) it could still take a long time.

In the demos directory online, I have included a basic version of grep implemented on MapReduce. Download it and take a look at it. See if you can figure out how it works. There's one new thing for this program and that is passing information to map function. How is this done? The `configure` function is called whenever a mapper/reducer is constructed.

Run the program on the cluster. Again, to do this you'll have to call `hadoop jar demos.jar` and then tell it to run the grep example. Notice this example take three command-line arguments (try searching for "text" or "banana").

Assuming all goes well, you should get an answer back. What are the numbers that get printed before the lines?

Here are a few things to try:

- Can you use the reducer as a combiner? If so, try it out :)
- Modify the code so that it prints out both the number as well as the filename. Can you get it to group the results by filename?
- Modify the code to take a regular expression to match, not just a word. `java.util.regex` should be useful. To pass a regular expression as a parameter on the command-line you'll probably need to wrap it in quotes.