Assignment 2
Running PageRank on Wikipedia

Due Date: Thursday, Oct 30 at 4:30 pm

Goals:

1. Understand the PageRank algorithm and how it works in MapReduce
2. Implement PageRank and execute it on a large corpus of data
3. Examine the output from running PageRank on Wikipedia to measure the relative importance of pages in the corpus

PageRank:

The Algorithm --

For the PageRank algorithm itself, refer to the lecture notes, or see:
http://infolab.stanford.edu/~backrub/google.html (section 2.1.1)

This lab is more involved than the first, containing several different MapReduce passes used sequentially. The input to the program are pages from the English-language edition of Wikipedia. You'll compute the "importance" of various Wikipedia pages/articles as determined by the PageRank metric.

The English-language Wikipedia corpus is about 20 GB, spread across 2 million files -- one per page. However, the Hadoop DFS, like the Google File System, is designed to operate efficiently on a small number of large files rather than on a large number of small files. If we were to load the Wikipedia files into Hadoop DFS individually, then run a MapReduce process on this, Hadoop would need to perform 2 million file open--seek--read--close operations -- very time consuming!
Instead, you'll be using a pre-processed version of the Wikipedia corpus in which the pages are stored in an XML format, with many thousands of pages per file. This has been further preprocessed such that all the data for a single page is on the same line. This makes it easy to use the default InputFormat, which performs one map() call per line of each file it reads. The mapper will still perform a separate map() for each page of Wikipedia, but since it is sequentially scanning through a small number of very large files, performance is much better than in the separate-file case.

Each page of Wikipedia is represented in XML as follows:

```
<page>
  <title> Page_Name </title>
  (other fields we do not care about)
  <revision optionalAttr="val">
    <text optionalAttr2="val2"> (Page body goes here)
    </text>
  </revision>
</page>
```

As mentioned before, the pages have been "flattened" to be represented on a single line. So this will be laid out on a single line like:

```
<title>Page Name</title>(other fields)<revision optionalAttr="val"><text
```

optionalAttr="val2">(body)</text></revision>

The body text of the page also has all newlines converted to spaces to ensure it stays on one line in this representation.

MapReduce Steps:

This presents the high-level requirements of what each phase of the program should do: (While there may be other, equivalent implementations of PageRank, this suggests one such method that can be implemented in this lab.)

Step 1: Create Link Graph

Process individual lines of the input corpus, one at a time. These lines contain the XML representations of individual pages of Wikipedia. Turn this into a link graph and initial page rank values (Use *1-d* as your initial PageRank value).

Think: What output key do you need to use? What data needs to be in the output value?

Step 2: Process PageRank

This is the component which will be run in your main loop. The output of this section should be directly readable as the input of this same step, so that you can run it multiple times.

In this section, you should divide fragments of the input PageRank up among the links on a page, and then recombine all the fragments received by a page into the next iteration of PageRank.

Step 3: Cleanup and Sorting

The goal of this lab is to understand which pages on Wikipedia have a high PageRank value. Therefore, we use one more "cleanup" pass to extract this data into a form we can inspect. Strip away any data that you were using during the repetitions of step 2 so that the output is just a mapping between page names and PageRank values. We would like our output data sorted by PageRank value.

Hint: Use only 1 reducer to sort everything. What should your key and value be to sort things by PageRank?

At this point, the data can be inspected and the most highly-ranked pages can be determined.

CodeLab Exercise:

Implement the PageRank algorithm described above. You will need a driver class to run this process, which should run the link graph generator, calculate PageRank for 10 iterations, and then run the cleanup pass. Run PageRank, and find out what the top ten highest-PageRank pages are.

Overall advice:

- Our local copy of wikipedia is stored in "/shared/wikipedia" on the DFS. Use this as your first input directory. **Do NOT use it as your output directory.**

- And, **do not use it at all until you are sure you've got everything working!** There is a **test data set** stored in "/shared/wiki_small" on the DFS--this contains about 100,000 articles. Use the data from this source as your first input directory to test your system. Move up to the full copy when you are convinced that your system works.
- SequenceFiles are a special binary format used by Hadoop for fast intermediate I/O. The output of the link graph generator, the input and output of the PageRank cycle, and the input to the cleanup pass, can all be set to org.apache.hadoop.mapred.SequenceInputFormat and SequenceOutputFormat for faster processing, if you don't need the intermediate values for debugging.
- Test running a single pass of the PageRank mapper/reducer before putting it in a loop
- Each pass will require its own input and output directory; one output directory is used as the input directory for the next pass of the algorithm. Set the input and output directory names in the JobConf to values that make sense for this flow.
- Create a new JobClient and JobConf object for each MapReduce pass. main() should call a series of driver methods.
- Remember that you need to remove your intermediate/output directories between executions of your program
- The input and output types for each of these passes should be Text. You should design a textual representation for the data that must be passed through each phase, that you can serialize to and parse from efficiently. Alternatively, if you're feeling daring, implement your own subclass of *Writable* which includes the information you need.
- Set the number of map tasks to 300 (this is based on our cluster size)
- Set the number of reduce tasks to 50.
- The PageRank for each page will be a very small floating-point number. You may want to multiply all PageRank values by a constant 10,000 or so in the cleanup step to make these numbers more readable.
- The final cleanup step should have 1 reduce task, to get a single list of all pages.
- Don't forget, this is "real" data. We've done most of the dirty work for you in terms of formatting the input into a presentable manor, but there might be lines which don't conform to the layout you expect, blank lines, etc. Your code must be robust to these parsing errors. Just ignore any lines that are illegal -- but don't cause a crash!
- **Start early.** This project represents **considerably more programming effort** than project 1.

Remember, project 1 provides instructions on how to set up a project and execute a program on the Hadoop cluster. Refer back to the instructions in that document for how to do this assignment as well.

Testing Your Code:

If you try to test your program on the cluster you're going to waste inordinate amounts of time shuttling code back and forth, as well as potentially waiting on other students who run long jobs.

Before you run any MapReduce code, you should **unit test** individual functions, calling them on a single piece of example input data (e.g., a single line out of the data set) and seeing what they do.

After you have unit tested all your components, you should do some **small integration tests** which make sure all the working parts fit together, and work on a small amount of representative data.

For this purpose, we have posted a *very* small data set on the web site at:
http://www.cs.washington.edu/education/courses/cse490h/08au/projects/wiki-micro.txt.gz
This is a ~2 MB download which will unzip to about 5 MB of text in the exact format you should expect to see on the cluster.

Download the contents of this file and stick it in an "input" folder on your local machine. Test against this for your unit testing and initial debugging. After this works, then move up to /shared/wiki_small. After it works there, then and only then should you run on the full dataset. If individual passes of your MapReduce program take more than 15 minutes, you have done something wrong. You should kill your job (see instructions in assignment 1), figure out where your bugs are, and try again. (Don't forget to pre-test on smaller datasets again!)

Extensions: (For the Fearless)

If you're finished early and want a challenge, feel free to try some/all of these. Karma++ if you do.

- Write a pass that determines whether or not PageRank has converged, rather than using a fixed number of iterations
- Get an inverted indexer working over the text of this XML document
- Combine the inverted indexer and PageRank to make a simple search engine
- Write a MapReduce pass to find the top ten pages without needing to push everything through a single reducer

(I haven't done these. No bets on how easy/hard these are.)

Writeup:

Please write up in a text file (no Word documents, PDFs, etc -- plain old text, please!) the answers to the following questions:

1) How much time did this lab take you? What was the most challenging component?

2) Describe the data types you used for keys and values in the various MapReduce stages. If you serialized data to Text and back, describe how you laid the contents of the Text out in the data stream.

3) What scalability hazards, if any, do you see in this system? Do you have any ideas how these might be overcome or mitigated?

4) What are the 10 pages with the highest PageRank? Does that surprise you? Why/why not?

5) Describe how you tested your code before running on the large data set. Did you find any tricky or surprising bugs? Include any test programs / JUnit test cases, etc, in your code submission.

6) List any extensions you performed, assumptions you made about the nature of the problem, etc.

Submission Instructions:

Again, use the "turnin" program on attu to submit your code. All code must compile. Compilation errors will result in a grade of "0." Put all your code in a directory together with your writeup and tar/gzip it up.

The preceding material is from the University of Washington Computer Science & Engineering senior undergraduate course:

CSE 490H
Scalable Systems: Design, Implementation and Use of Large Scale Clusters
Autumn 2008

For further information (including all lecture material) see:

http://www.cs.washington.edu/education/courses/490h/08au/