Codelab 3
Running PageRank on Wikipedia (Hadoop version 0.17.0)

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:
http://net.pku.edu.cn/~course/cs402/codelabs/google.pdf (section 2.1.1)
http://infolab.stanford.edu/~backrub/google.html

This lab is a more involved process containing several different MapReduce passes used sequentially. The input to the program are pages from the English-language edition of Wikipedia. Rather than operating on individual text file inputs, however, it is modified to work on the Wikipedia data, which is not stored in a large number of flat text files.

Storing large numbers of individual files in the Hadoop DFS is inefficient. The DFS can hold large amounts of data (many terabytes), but expects this data to be primarily in a small number of very large files. The English-language Wikipedia corpus is about 15 GB, spread across 2 million files. If we were to load these files into the DFS individually, and then run a MapReduce process on this, then Hadoop would need to perform 2 million file open--seek--read--close operations: very time consuming!

Instead, the pages are stored in an XML format, with several (many thousands) of pages per file. This has been further preprocessed such that all the XML for a single page is on the same line. This makes it easy for us 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 higher 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 is a straightforward one we can use for this lab.)

Step 1: Create Link Graph

The PageRank algorithm operates by distributing PageRank value from pages to their neighbors by traversing the link graph representation of the pages. This step parses all the page-to-page links out of the page bodies, and maps each page to a list of the pages it links to. The reducer is simply the identity function, as no data relevant to a particular page needs to be combined in from other pages: each page and its out-links is a self-contained unit.

As for the mapper:

- The input to the mapper is a key consisting of the byte offset into this large XML file of the current line, and a value consisting of a single line of flattened XML.
- The output of the mapper should have the page name (between <title>....</title>) as its key.
- The output value should be all of the links extracted from the line of text, as well as an initial PageRank value for the page equal to "d". (d is the "random jump probability"; use the value 0.15) The format of this output value

should be something you can easily parse into its individual components, later.

- Links are represented in the input as either "[[TargetPageName]]" or "[[PageName|DisplayText]]". You should discard any display text.
- If the current line of text in the mapper is not a <page>...</page>, there should be no output.
- Note that we are uninterested in the text of the article: while this is useful for indexing, this is orthogonal to PageRank.

Step 2: PageRank Distribution

The actual algorithm itself is encoded in the PageRankMapper and PageRankReducer classes. Because PageRank is an iterative algorithm, this MapReduce task is run several times over.

- You do not need to determine end criteria for the PageRank algorithm; just use a fixed loop of 10 iterations in your driver
- The mapper receives as input a slice of the link graph:
    - The key is the page name
    - The value is the outgoing page list and PageRank value from the previous iteration
    - (This is the same format as you emitted in step 1)

The mapper receives as input a Text object representing a slice of the link graph: one page and all its outbound links, as well as its current PageRank (carried over from the previous iteration, or an initial seed value). It then allocates equal fractions of the current PageRank of the input page to each of its outbound links. The outputs of the mapper are pairs of the form: <TargetPageName, PageRankFragment>. Because we want to preserve the link graph for future iterations, we also emit the link graph slice to our own PageName.

The reducer receives as input everything with the same PageName key. This is a set of PageRankFragments -- floating-point values representing fractions of the PageRanks of all pages which point to the current PageName -- as well as the outbound link set for this PageName. The reducer sums these fragments and calculates the new PageRank for the current page using the formula *(1 - D) + D*Sum(PR_fragment_inputs)*. It then formats a final output string using the same format as the mapper input, incorporating the link graph slice and the current PageRank.

Note that the reducer can be used as a combiner.

Hint for writing a MapReduce program to perform a single iteration of PageRank.

Map stage: $(Y,[PR(Y),\{Z_1,....,Z_n\}]) \rightarrow (Z_i,\frac{PR(Y)}{n}),(Y,\{Z_1,....,Z_n\})$

Where Y is a title, *PR(Y)* – title's current PageRank, and Z_i – i-th outgoing link from article Y.

Reduce stage: $(Y,[S_0,....,S_m,\{Z_1,....,Z_n\}]) \rightarrow (Y,[(1-d)+d\times\sum_{i=1}^{m}S_i,\{Z_1,....,Z_n\}])$

Where S_i are *PR(Y)/n* terms from the map stage, and *d* is damping factor. You can experiment with the values of *d*, but we recommand you set it at 0.85.

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. Write a PageRankCleanupMapper to do this step.

The mapper in this step receives a <PageName, (PageRank, OutboundLinkList)> datum, just like the PageRank calculator. It emits as output <PageRank, PageName>, discarding the link list. Note that the PageRank is now the key, and PageName is the datum. This implicitly sorts the values by PageRank, as keys passed to a reducer are processed in sorted order. The reducer is simply the identity function.

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 "/wiki" on the DFS. Use this as your first input directory. Do NOT use it as your output directory.
- There is a test data set stored in "/smallwiki" on the DFS--it contains about 100,000 articles. Use this 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, should all be set to org.apache.hadoop.mapred.SequenceInputFormat and

SequenceOutputFormat.

- 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.
- Set the number of map tasks to 30 (this is based on our cluster size of 15 nodes)
- Set the number of reduce tasks to 15.
- 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.

Extensions: (For the Fearless)

You're not expected to get this far, but if you are super-ambitious, you could try...

- 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 this with PageRank into a simple search engine

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