PageRank using Hadoop

Due date: December 1st

Task Description

The goal of this programming assignment is to compute the PageRanks of a collection of hyperlinked Wikipedia documents using Hadoop. The PageRank score of a web page serves as an indicator of the importance of the page. Many web search engines (e.g., Google) use PageRank scores in some form to rank user-submitted queries. The goals of this assignment are to:

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

Wikipedia Data

The inputs to the program are pages from the English-language edition of Wikipedia. You will compute the "importance" of various Wikipedia pages/articles as determined by the PageRank metric.

The English-language Wikipedia corpus is about 15 GB, spread across ~ 2.5 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 and then run a MapReduce process on this, Hadoop would need to perform 2.5 million file open–seek-read–close operations – which is very time consuming. Instead, you will 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_Title </title>
   <text> Page_body_goes_here </text>
</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:

```
<page><title> Page_Title </title><text> Page_body_goes_here </text></page>
```

The body text of the page also has all newlines converted to spaces to ensure it stays on one line in this representation. Links to other Wikipedia articles are of the form

```
[[Name of other article]].
```

MapReduce Steps

These steps present the high-level requirements of what each phase of the program should do.

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.

You'll need to look up and understand the Wikipedia Link format, which is described at

http://en.wikipedia.org/wiki/Help:Links#Wikilinks and
http://en.wikipedia.org/wiki/Wikipedia:Manual_of_Style/Linking

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

Hint: You will need to build the graph in two steps (see Cloud code and the hints/specifications towards the end of the assignment description). In the first step, you will emit all edges (source_page, destination_page), including some dummy edges that will help identify pages with no outgoing links. In the second step, you will emit an adjacency list for each node in the graph (i.e., page with an entry in the original wiki file). Pages that appear among outgoing links, but not in the original file, will be ignored.

Note: Only Wikipedia articles should be considered as links. Links that contain a semi-column, e.g. [[text : text]] should be ignored, as they don't correspond to articles (you also need to ignore input pages/titles that contain a semi-column, e.g. `<title>MediaWiki:Imghistory</title>`). Among article pages, we can have redirect, disambiguation, empty or stub pages. All should be included in the graph, if they have an entry in the original file.

Redirect pages are defined here:
http://en.wikipedia.org/wiki/Wikipedia:Redirect

Disambiguation pages are defined here:
http://en.wikipedia.org/wiki/Wikipedia:Disambiguation

Stub pages are defined here:
http://en.wikipedia.org/wiki/Wikipedia:Stub

Step 2. Corpus Size: Find the size of the Wikipedia corpus, N. (Hint: for the small development data set the size should be 3455).

Step 3. Process PageRank: This is the component that will be run in your main loop. *We assume that the graph does not fit in the memory at any time.* An idea for how this can be implemented is provided below. The output of this step should be directly readable as the input of this same step, so that you can run it multiple times. In this step, 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. Assuming that Y is a page, PR(Y) is current PageRank of Y, and $Z_1,\ldots,Z_n$ are outgoing links from Y, the Map and Reduce operations have the following inputs/outputs:

Map input: $\left(Y,[PR(Y),\{Z_1,...,Z_n\}]\right)$

Map output: $\left(Z_i,\frac{PR(Y)}{n}\right),\left(Y,\{Z_1,...,Z_n\}\right)$

Reduce input: $\left(Y,[S_1,...,S_m,\{Z_1,...,Z_n\}]\right)$

Reduce output: $\left(Y,\left[\frac{\varepsilon}{N}+(1-\varepsilon)\sum_{i=1}^{m}S_m,\{Z_1,...,Z_n\}\right]\right)$

Step 4. Normalize the PageRank scores: This component will be run after every PageRank iteration. It finds the sum of all current scores and normalizes scores by sum.

Step 5. Cleanup and Sorting: The goal of this step 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 3, so that the output is just a mapping between page names and PageRank values. We would like the 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.

Implementation Recommendations and Suggestions

Implement the PageRank algorithm using Hadoop as outlined above. You will need a driver class, which should (a) run the link graph generator; (b) calculate PageRank and normalize scores for 10 iterations, and (c) finally run the cleanup step. For extra challenge, write a job that determines whether or not PageRank has converged, rather than using a fixed number of iterations (5 extra credit points). Run PageRank, and find out what the top ten highest-PageRank pages are.

Make sure you optimize your code as much as possible (e.g., use in-mapper-combining whenever that is possible).

Make sure you don't hardcode input files or parameters – read them as command line arguments.

Avoid using large data structures that could lead to memory issue (e.g., you should not store all your graph in a hashtable at any time).

Overall advice

- As for the previous assignment, you can take a look at Cloud[9], an open source MapReduce library for Hadoop developed by a group at UMD, to get ideas about how you could implement various classes/methods in Hadoop. The Cloud[9] source code is available also on KSOL.

- Use the hints at the following link to pass objects to all mappers (e.g., to pass the number of documents, or the sum of page rank scores to mappers – assuming those are written to files, possibly by other MapReduce jobs):

  http://stackoverflow.com/questions/13501276/best-practice-to-pass-copy-of-object-to-all-mappers-in-hadoop

- Two test data sets are posted on KSOL: a *very* small test data set (scowiki-20090929-one-page-per-line, ~8Mb) and a slightly larger data set (afwiki-20091002-one-page-per-line, ~68Mb). First, test your code with the small data set, on your local machine, using your local file system (standalone mode). Then, test your code on Beocat, using the larger data set. Run your code on Beocat when you are convinced that your system works. *A file containing all Wikipedia English pages will be provided for your final test on Beocat.*
- 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 (you could use this when running your code on Beocat).
- Test 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 iteration 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.
- Select an appropriate number of map tasks and reduce tasks.
- 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 use one reducer, to get a single list of all pages.
- Remember, this is real data. The data has been cleaned up in terms of formatting the input into a presentable manner, but there might be lines that do not 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 do not cause a crash!
- Be aware that some things will work in the standalone version but not in the distributed version on Beocat. For example, you need to configure the jobs corresponding to the iterations in PageRank inside the iteration loop, in order to have you code work properly on Beocat – each iterations triggers a job, they need to be configured accordingly.
- You will need to copy files from HFDS to the local system, if you want to look at them, or to access them from the driver program.
- The Hadoop API reference is at http://hadoop.apache.org/core/docs/current/api/ – when in doubt, look it up here first!
- Start early! Beocat might be busy and you may not be able to run your code on Beocat if you wait until the last minute.

Testing Your Code

If you try to test your program on Beocat, you're going to waste inordinate amounts of time shuttling code back and forth, as well as potentially waiting on other people who run long jobs. Before you run any MapReduce code on Beocat, 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. This is the purpose of the *very* small data set posted online.

Download the contents of this file and place 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 slightly larger dataset and run it on Beocat. If individual passes of your MapReduce program take too long, you may have done something wrong. You should kill your job, figure out where your bugs are, and try again. When this is working fine, do a final run on the largest Wikipedia dataset (to be provided).

Submit: You should submit a zip archive, named **yourname.zip**, that contains the following (please use file upload in Canvas to submit your assignment):

(10 points) A short report that

- Briefly describes the functionality of your program (main classes/data structures you used in your implementation).
- List the map/reduce jobs in your code. For each map/reduce job, describe the inputs and the outputs (to map and reduce methods, respectively).
- Describe the data types you used for keys and values in the various MapReduce stages.
- Explains how to run your program (standalone mode and Beocat) and show proof that your program does what it's supposed to do in both cases.
- Include a discussion section that comments on your whole experience with this assignment, any lessons learned, or any issues left as open questions.
- How long did it take you to complete this assignment? What was the most challenging part of the assignment?
- Describe how you tested your code before running on the large data set. Did you find any tricky or surprising bugs?
- What are the 10 pages with the highest PageRank? Does that surprise you? Why/why not?
- Acknowledge any code you may have used or any resources that you may have consulted in order to complete your assignment.

(20 points) The source code of your program (including a short readme that explains how your code is organized). Make sure your code is properly commented.

More hints/specifications on graph construction (based on how Cloud[9] builds the adjacency graph)

First, for each page like this:

<page><title> Page_Title </title><text> Page_body_goes_here </text></page>

you need to find its adjacency list, if the page points to other pages.

Even if there are no links in the page_body, you should still include that page in the graph with an empty adjacency list.

If the adjacency list contains pages that don't have an entry in the original file, simply ignore them. One problem is that you can't ignore them, unless you have parsed the whole collection and found out that they don't have an entry in the original file. With a first parse, you could build a hashtable which holds all pages in the original file; with another pass, you could check to see what pages in the adjacency list are there or not. But that hashtable could get pretty large, and we might not be able to keep it in the memory.

Cloud provides a solution that avoids constructing the hashtable. It builds the graph using two MapReduce jobs as follows. In job 1, all edges (source_page, destination_page) are emitted, including dummy edges that will be used to identify pages that don't have outgoing links. More precisely, for a page p having outgoing links q1,...,qn, the mapper of the first job emits

((p,0), p) and ((q1,1), p), ..., ((qn,1), p)

If there are no outgoing links, it only emits ((p,0), p).

The data is partitioned using a custom partitioner (see lecture 3), which ensures that all entries having p in their key (both (p,0) and (p,1)) will go to the same reducer.

sCurrentArticle is a global variable per reducer.

If key.getRightElement is 0, as in ((p,0), p), then the reducer emits (p, p) and sCurrentArticle is set to p.

If the key.RightElement is 1, as in ((p,1), s), and sCurrentArticle is p, then the reducer emits (s, p).

If the key.RightElement is 1, as in ((p,1), s), but sCurrentArticle is different than p, those p pages are ignored (they don’t have an entry in the original file).

In job 2, edges (source_page, destination_page) are used to construct adjacency lists.

The mapper emits (source_page, destination_page). The reducer aggregates after source_node and constructs the adjacency list, by ignoring destination_pages that are identical with source_page. This way, we can end up with pages with empty adjacency list, i.e. no outgoing links.

That’s roughly the Cloud implementation for the wiki graph construction.