Assignment 3
ITCS-6190/8190: Cloud Computing for Data Analysis

Due by 11:59:59pm on Friday, November 4, 2016

This assignment is based on a similar assignment developed at the University of Washington.

Running PageRank on Wikipedia

The goal of this programming assignment is to compute the PageRanks of an input set of hyper-linked Wikipedia documents using Hadoop MapReduce and Spark. 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 the PageRank algorithm and how it works in MapReduce and Spark.
2. Implement PageRank and execute it on a large corpus of data.
3. Examine the output from running PageRank on Simple English Wikipedia to measure the relative importance of pages in the corpus.

To run your program on the full Simple English Wikipedia archive, you will need to run it on the dsba-hadoop cluster to which you have access.

PageRank: The Algorithm

For the PageRank algorithm itself, refer to the Class 11 notes, or see
Section 2.1.1 of `http://infolab.stanford.edu/~backrub/google.html` .

This assignment is more involved than the first two assignments, and will involve several different MapReduce passes used sequentially. The inputs to the program are pages from the Simple English Wikipedia. You will compute the "importance" of various Wikipedia pages/articles as determined by the PageRank metric.

The Simple English Wikipedia corpus is about 500 MB, spread across 200,000 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 200,000 file open–seek-read–close operations – which is very time consuming. Instead, you will be using a pre-processed version of the Simple 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 Simple 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:

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

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">(Page 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. Links to other Wikipedia articles are of the form "`[[Name of other article]]`".

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 assignment.)

1. **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 $\frac{1}{N}$ as your initial PageRank value, and assume the damping factor d is 0.85. N is the total pages in the corpus (You need to calculate its value dynamically from the given corpus).

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

2. **Step 2: Process PageRank:** This is the component that will be run in your main loop. The output of this phase should be directly readable as the input of this same step, so that you can run it multiple times.

   In this phase, 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.

3. **Step 3: 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 2 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.

Task 1: Implementation using MapReduce

Implement the PageRank algorithm described above using Hadoop MapReduce. 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 100 highest-PageRank pages are.

Overall advice:

- Two very simple graphs, *graph1.txt* and *graph2.txt*, and a very small test data set *wiki-micro.txt.gz* are posted on Canvas. Please first test your code on your local machine using your local file system before testing on the course Hadoop cluster.
- We have provided the above test data sets and the SimpleWiki data set on the cluster in the `/projects/cloud/pagerank` directory. Use the data from this source as your first input directory to test your system. Move up to working on the SimpleWiki data when you are convinced that your system works.
- Test run 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 Job to values that make sense for this flow.
- Create a new Job and Configuration 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.
- Select an appropriate number of map tasks and reduce tasks.
- The final cleanup step should have 1 reduce task, to get a single list of all pages.
- Remember, this is "real" data. Tha data has been cleaned up for you 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!

- 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 SequenceFile input and output format for faster processing, if you don't need the intermediate values for debugging.
- **Start early.** This project represents considerably more programming effort than Assignments 1 and 2.
- Use a small test input before moving to larger inputs.
- The Hadoop API reference is at `http://hadoop.apache.org/core/docs/current/api/` – when in doubt, look it up here first!

Task 2: Implementation using Spark

Reimplement the PageRank algorithm using Spark. As you know, a major differentiator between Hadoop MapReduce and Spark is the use of RDDs. While developing your code, you should notice and be able to contrast the differences between MapReduce and Spark codes.

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 the two simple graphs and a *very* small data set on Canvas, *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 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 the cluster. After it works there, then and only then should you run on the full dataset. 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. (Do not forget to pre-test on smaller datasets again!)

Extra Credit Ideas

- 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.

Maximum credit for implementing one of the above extensions is 10 points. (The required tasks on the assignment are for a total of 100 points.)

Writeup

In your README (plain text) file, provide instructions for execution, any assumptions you made, and any extra credit ideas you implemented. In addition, please list from (at least) **four** prespectives how your Hadoop MapReduce and Spark implementations differ and why you think this might be good/bad. As an example:

- From development time prespective, MapReduce needed more time coding PageRank than Spark. This is because Spark provides transformation functions like map(), which allowed me to ....

Note: Items that address the same perspective would be counted as only **one** out of four. Think about perspectives other than development time!

Submission

1. Please submit all source code, properly commented, along with a README providing instructions for exceution and comparison between MapReduce and Spark implementations.
2. All code must compile. Compilation errors will result in a grade of 0.
3. Place your Hadoop and Spark source files in separate directories (e.g., Hadoop and Spark directories respectively).
4. Put both in a directory named as your UNCC email login ID together with your README writeup and tar/gzip it up before submitting.

Assignments are due by 11:59:59pm on Friday, November 4, 2016. Submission will be through Canvas as for the previous assignments.

Assignments are to be done individually. See course syllabus for late submission policy and academic integrity guidelines.