

**CS 549—Fall 2015**  
**Distributed Systems and Cloud Computing**  
**Assignment Five—Page Rank in Hadoop**

Implement the PageRank algorithm to find the most popular pages in a set of Web pages. You will use your implementation of PageRank to find the 'most popular' pages in a dump of Wikipedia pages that you are provided with.

*Input format:* We will assume that the link graph is initially available as a list of vertices and their adjacency lists (a list of “friend” vertices for each vertex). Each entry is a line in a file of the form:

```
node-id: to-node1 to-node2 ...
```

In addition, for output purposes, there is separately information about page names for each node, of the form:

```
node-id: page-name
```

*Output format:* The goal is to produce a file that contains the page rank of each Wikipedia page. There should be one line for each page, and each line should contain the Wikipedia page name (**not** the vertex identifier) and the page rank. The file should be *sorted by rank in reverse order*.

As described in the lectures, PageRank is an iterative algorithm, so you will implement this as an iterative MapReduce. Thus, the output of round  $k$  will be used as the input of round  $k+1$ . In addition, we will need three additional types of jobs: One for converting the input data into our intermediate format, one for computing how much the rank values have changed from one round to the next (actually made up of two MapReduce jobs), and one for converting the intermediate format into the output format.

The intermediate format for iterative PageRank computations should include the node identifier, page rank, and the adjacency list for that node. Remember that the adjacency list should be propagated through each iteration, having been read from the initial data. You will want to initialize the intermediate data with page ranks of 1 for each page. Note also that Web page names do not appear until the output of the final result; you will need to modify the code to provide this. Use the node identifiers (long integers) to identify nodes during PageRank computations.

You are provided with a driver that will read the command-line arguments and, depending on the first argument, implement the following four functions (you will need to finish the definitions of the mapper/reducer pairs for each):

- `init inputDir outputDir #reducers`: This job read the file(s) in the input directory, convert them into your intermediate format, and outputs

- the data to the output directory, using the specified number of reducers.
- `iter inputDir outputDir #reducers`: This job performs a single round of PageRank by reading data in your intermediate format from the input directory, and writing data in your intermediate format to the output directory, using the specified number of reducers.
- `diff inputDir1 inputDir2 outputDir #reducers`: This job reads data in your intermediate format from *both* input directories and outputs a single line that contains the maximum difference between any pair of rank values for the same vertex. You must use absolute values for the differences, i.e., the change from 0.98 to 0.97 is 0.01. While diff runs as a single task, it is implemented using two different MapReduce jobs run successively.
- `finish inputDir outputDir #reducers`: This job reads data in your intermediate format from the input directory, converts the data to the output format, and outputs it to the output directory, using the specified number of reducers.

Additionally, the driver provides a composite function that needs to be submitted only once and runs the entire PageRank algorithm from beginning to end, i.e., until convergence has occurred.

- `composite inputDir outputDir intermDir1 intermDir2 diffDir #reducers`: This function runs the init task, then it alternates between running the iter and diff tasks until convergence has occurred (**diff ≤ 30 in the case of the test data**), at which point it runs the finish task and places the output into <outputDir>. Note, running the diff task after every iteration could add considerable time to your job, so this function runs the diff task after every two or three iterations to save computation time.

Each job must delete the output directory if it already exists, and *it must also output your name to System.out every time it is invoked*. The main class of your job is `edu.stevens.cs549.hadoop.pagerank.PageRankDriver`. You should document your code sufficiently to enable the grader to understand how you are processing inputs and outputs in mappers and reducers.

You should start with the Eclipse project that is provided, that gives you some code to get you started. This is a Maven project, so be careful to import it as such into your Eclipse workspace. The pom file already imports the requisite Hadoop libraries from global Maven repositories. You should still install Hadoop on your local machine, since you will need its libraries in your classpath when testing your app locally.

A good way to make sure that your implementation is correct is to test it in pseudo-distributed mode, using the Hadoop installation locally (perhaps in a Linux VM) and the example graph from below. This will save your AWS credits for the final job with the Wikipedia data. You should only test the composite task after you are sure that the other tasks work properly.

Most of the code is already provided to you. You need to customize the driver to display your name and userid when a job runs, and fill in the missing details in the mappers and reducers. There are comments in the code explaining what the missing code should do, so this is largely an exercise in understanding the Hadoop API, and the implementation of PageRank using Hadoop.

The code you are provided with just outputs the vertex identifiers (with their page ranks). For a full grade, you should output Wikipedia page names at the end. There are two ways to do this. The easiest way is to store the mapping of node identifiers to page names as one or more files in Hadoop's cache, and read this into the mapper or reducer that provides the final output. In your driver, add the mapping file(s) to the cache:

```
job.addCacheFile(...URI in HDFS...);
```

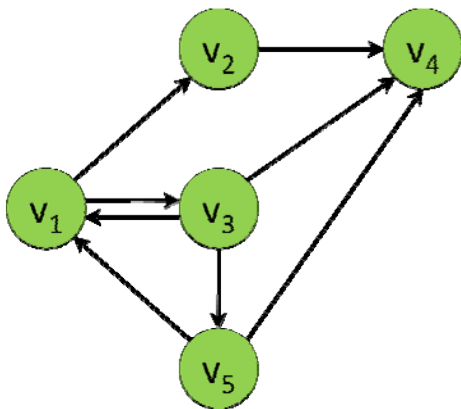
In your mapper/reducer, read this file(s) into memory by overriding the setup method:

```
public void setup(Context context) {  
    super(context);  
    URI[] files = context.getCacheFiles();  
    Path path = new Path(files[0]);  
    ...  
}
```

A more scalable way to do this would be to perform an equipartitioned join of the output page ranks and the page names, where the page name table includes node identifiers as keys. However it is sufficient to use the cache for this assignment.

You should test your PageRank implementation on two sets of data.

First, get it to work on the following example graph. It is sufficient to demonstrate your implementation working locally with Hadoop (perhaps using a VM on which you have installed Hadoop):



Second, use PageRank to find the fifty highest-ranked nodes in Wikipedia, based on

information about page links that is provided to you. This data is considerably cleaned up from the dumps provided by Wikipedia, removing dead links and links outside Wikipedia or to images or remarks. It has already been split into multiple files and has been imported into Amazon S3:

<http://s3.amazonaws.com/cs549/links-simple-sorted.tgz>

In addition, there is a collection of files that map node identifiers to page names:

<http://s3.amazonaws.com/cs549/page-names.tgz>

For this case, you should use Amazon's Elastic MapReduce (EMR) to run the job.

### **Step-by-step guide: Running on Amazon's Elastic MapReduce**

The following is a step-by-step guide for setting up your solution to run on Amazon's Elastic MapReduce. To conserve your AWS credit, only use the full Wikipedia data and the maximum number of machines after you are sure that your solution works on the sample graph with one or two machines.

1. Go to the Amazon S3 Management Console (<https://console.aws.amazon.com/s3/home>) and create a new bucket with a unique name. This bucket will store input, output, MapReduce logs, and your exported jar file for use with Elastic MapReduce.
2. Within your created bucket, create two new folders named "jars" and "in". Upload your exported jar file into the "jars" folder, and upload the test graph data into the "in" folder. When you are ready to test with the Wikipedia data, download the .tgz file, extract it, and upload the files to your "in" folder in S3. You will also need to upload the files containing page names to another folder (you choose a name) in S3.
1. Now we are ready to run an Elastic MapReduce job. Go to the EMR Management Console (<https://console.aws.amazon.com/elasticmapreduce/home>) and click "Create New Job Flow".
2. Give your job flow any name, select "Run your own application -> Custom JAR", and then Continue.
3. Under "Jar Location\*:" Type in the location of your uploaded jar file (containing your compiled code). If you followed the instructions from step 2, your path should be "*yourbucketname*/jars/PageRank.jar"
4. "Jar Arguments\*:" will have the arguments to run your composite task. Input and output folders should be referred to as S3 paths. Here is a sample set of arguments:

```
edu.stevens.cs549.hadoop.pagerank.PageRankDriver composite  
s3n://bucketname/in s3n://bucketname/out intermediate1  
intermediate2 diffdir 20
```

Note that only the input and output need to be S3 paths. Intermediate data can be stored in HDFS folders (This will happen automatically when no `s3n://` is specified).

5. On the next page, specify the number of instances that will run your MapReduce job. To start, set the instance count for “Core Instance Group” to 1, and leave the instance count for “Task Instance Group” at 0. This will run your job with two machines—one master and one slave node. When you are ready to run your job on the full Wikipedia dataset, you can set the Core Instance count to 19 for a total of 20 machines.
6. On the next page, make sure to enable logging and set the log path to “`s3n://yourbucketname/log`”. This will be very useful for debugging your job if anything goes wrong.
7. On the next page, select “Proceed with no bootstrap actions”
8. On the final page, review your settings to make sure they are correct. When you are ready, click “Create Job Flow” to start your job. This can take a long time on the Wikipedia data, but it should complete in a couple of hours using the full 20 machines. On the sample graph, expect your job flow to take a maximum of 30 minutes.

## What if things go wrong?

You can look at the Hadoop logs in the S3 Management Console under `yourbucketname/log/jobflowid/`. The most valuable log information will likely be under steps (2), since these are the logs generated by your jar file.

Please watch your resource consumption while working on this assignment; if you cannot get to the convergence threshold within a reasonable budget (say, \$15-\$20), just submit the top 10 you have, and document the number of rounds in your README file for partial credit.

## Submitting

Once you have your code working, please follow these instructions for submitting your assignment:

- Export your Eclipse project to the file system.

- Create a zip archive file, named after you, containing a directory with your name. E.g. if your name is Humphrey Bogart, then name the directory Humphrey\_Bogart.
- In that directory you should provide the zip archive that contains your sources, your Hadoop app jar file, PageRank.jar, and the test data that you used locally.
- Also include in the directory a report of your submission. This report should be in PDF format. **Do not provide a Word document.**

**The content of the report is very important.** A missing or sloppy report will hurt your final grade, even if you get everything working. In this report, describe how you completed the code for M/R, with a particular focus on the representations that you used for inputs to and outputs from the map and reduce steps. You should also describe how you tested the code. Again, it is very important for your grade that you do adequate testing.

You should also provide a video of your local testing, on the small test graph. Remember that your name should be displayed on each iteration of the job.

~~You should also provide a video showing your use of EMR to set up and start a run of your MR task. Show that you have set up a bucket in S3 as prescribed (no need to show the upload), and show the steps in the EMR console. Make sure your name is visible in the video, but don't provide your AWS credentials in the video! There is obviously no point in showing the entire run in EMR, since it will take some time, the video should just show you starting the program running.~~

Remember the format of the submission: A zip archive file, named after you, with a directory named after you. In this directory, provide a zip archive of your source files and resources, a jar file of your submission, a report for your submission, and videos (unless you have uploaded the videos to Google Drive).

Do **not** upload the Wikipedia data as part of your submission.