CS 549—Fall 2024
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` *`inputDir`*$_1$ *`inputDir`*$_2$ *`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.
- `join` *`ranksInput namesInput outputDir #reducers`*: This job reads data in your intermediate format from the input directory, as well as the names of the vertices (both inputs should be sorted by vertex id), and outputs the join of the two data sets on vertex id, using the specified number of reducers.
- `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 interimDir`*$_1$ *`interimDir`*$_2$ *`namesFile 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. The `diff` task leaves its output in the `diffDir` directory, which is deleted after the difference is computed. The file *`namesFile`* contains the names of the vertices, that should be one of the inputs to a last step that joins the result of page rank calculations for the vertices with their names.

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 Intellij IDEA project that is provided, that gives you some code to get you started. This is a Maven project, as before for the assignments. The pom file already imports the requisite Hadoop libraries from global Maven repositories[1]. You should still install Hadoop on your local machine, since you will need its libraries in your classpath when testing your app. The pom file specifies Hadoop 3.3.6 (EMR 6.15.0).

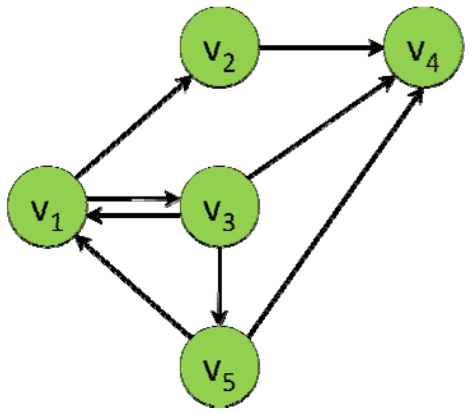
A good way to make sure that your implementation is correct is to test it in pseudo-distributed mode, using the Hadoop installation locally (perhaps in a Linux VM[2]) and the example graph from below. You should only test the composite task after you are sure that the other tasks work properly.

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. The scalable way to do this is to perform a join of the output page ranks and the page names, where the page name table includes node identifiers as keys. The code you are provided with shows how this join can be done in a penultimate map-reduce step (before sorting by pagerank).

In this assignment, you should run your Hadoop program for the Wikipedia data set on Amazon Elastic Map-Reduce (EMR). 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):



[1] The project uses dependencies on the Amazon EMR Artifact Repository, to ensure that you will compile against the same API as is running on EMR. More information here and here. Here is information on using EMR. Here is information on running Hadoop in Docker.
[2] See below for information about installing and running the Virtual Box hypervisor.

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-stevens/links-simple-sorted.tgz

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

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

Submitting

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

- 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 complete IDEA project, 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, and report on the results of your experiments with different numbers of reducers. **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. For testing with the Wikipedia data, try testing with a different number of reducers (say, five versus twenty), and comment on the difference in the results. 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 setting up and starting a run on the Wikipedia data. You should test your app on the Wikipedia data using Amazon AWS Elastic Map Reduce (EMR). Make sure your name is visible 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. Do **not** upload the Wikipedia data as part of your submission.

Running Hadoop in Virtual Box

If you have a Windows machine, you may want to run Hadoop in a Linux virtual machine. You are provided with an image that you can import into the Virtual Box hypervisor.

Download Virtual Box

Importing a VM into Virtual Box

Running a VirtualBox virtual machine

You are provided with a Linux VM that has Hadoop installed (username hadoop, password "abc123!"), configured to run as a single-node cluster. Once you have imported and started the machine, you should start a terminal (xterm) window, and start the cluster:

```
$HADOOP_HOME/sbin/start-dfs.sh
```

Once the namenode, secondary namenode and datanodes are up and running, start the resource and node managers:

```
$HADOOP_HOME/sbin/start-yarn.sh
```

Verify that everything is up and running:

```
$HADOOP_HOME/sbin/jps
```

Open a browser (Firefox) window in the running VM, and access the Hadoop UI:

Namenode: http://localhost:9870

Datanode: http://localhost:9864

YARN Resource Manager: http://localhost:8088

To share files with the VM, you can use a cloud storage service such as Dropbox or Google Drive. Virtual Box also supports file sharing between guest and host machines, but it requires the installation of Guest Additions on the VM.

Additional Information

Installing Hadoop on Ubuntu (Single Node Cluster)

Setting up a Single Node Hadoop Cluster

Inputs and Outputs for Mappers and Reducers

Here are some hints on the inputs and outputs for mappers and reducers:

InitMapper:

- The input line is a vertex followed by a ": " and space-separated adjacency list
- The output key is first vertex, value is second vertex [or adjacency list]

InitReducer:

- Input key is vertex
- Input list is adjacent vertices
- Output key is "vertex;rank" (initial rank is 1)
- Output value is comma-separated list of vertices

The reducer will write a key and value as tab-separated strings on an output line

IterMapper:

- The input key is the line number, which will be ignored
- The input value is "vertex;rank" and comma-separated list of friend vertices, separated by a tab
- Output (friend vertex, weight) pairs for each friend
- Output (vertex, adjacency list as comma-separated list of friend vertices); mark the value as an adjacency list with a special character in front

IterReducer:

- Input key is the vertex id
- Input list is adjacent vertices (with weights), as well as a comma-separated adjacency list
- Output is key is "vertex;rank" with rank computed from weights
- Output value is comma-separated adjacency list

DiffMap1:

- The input key is the line number, which will be ignored
- The input value is "vertex;rank" and comma-separated list of friend vertices, separated by a tab
- Output a single (vertex, rank) pair

DiffRed1:

- Input key is a vertex
- Input list is two rank values for that vertex (there must be exactly two)

- Output the difference between the ranks (as a key), null value

DiffMap2:

- The input key is the line number, which will be ignored
- The input value is a rank difference
- Output the string "Difference" and the difference value (as a Text)

DiffRed2:

- The input key is the string "Difference"
- The input list is the differences in ranks for each vertex
- Output the maximum difference value (as a key), null value

JoinNameMapper:

- The input key is the line number, which will be ignored.
- The input value is a colon-separated pair of vertex id and vertex name.
- Output key is a pair (vertex id, "0") and output value is the vertex name.

JoinRankMapper:

- The input key is the line number, which will be ignored.
- The input value is a tab-separated pair of vertex id and vertex rank.
- Output key is a pair (vertex id, "1") and output value is the vertex rank.

JoinReducer:

- The input key is the vertex id. The list of values is the vertex name followed by the vertex rank.
- Output key is vertex name and output value is vertex rank.

FinMapper:

- The input key is the line number, which will be ignored
- The input value is "vertex;rank" and comma-separated list of friend vertices, separated by a tab
- Output key is -rank (to sort in reverse order)
- Output value is vertex

FinReducer:

- Input key is page rank (which is negative)
- Input list is vertices with that rank
- Output (vertex, -rank) for each vertex in the list

Adding Vertex Names In The Output

An additional step in this assignment is to produce (vertex name, rank) pairs as output, instead of just (vertex, rank) where the vertex id is not meaningful for human consumption. The mapping from vertex to vertex name is stored in a separate file. These are the strategies you might follow to replace vertex ids with vertex names:

Distributed Cache

If the metadata is not too large, it can be put in the Hadoop distributed cache and then be available to be loaded by mappers and reducers. You would need to copy the file(s), mapping vertex identifiers to vertex names, to a shared file system (such as S3 or HDFS). Then use the `job.addCacheFile(uri)` operation to add this file to the distributed cache. Hadoop ensures that this file is available on the local file system of the task node wherever a mapper or reducer runs, so it can read the file into memory and do an in-memory lookup of a vertex name from a vertex id. You could for example use this approach in `FinReducer`, to convert a vertex id to its vertex name in the output. The challenge is that the mapping file is quite large and the reducer may run out of memory trying to load it.

The alternative to an in-memory lookup is to perform a join of the page rank outputs with the vertex name mapping. There are basically two alternatives: map-side join and reduce-side join.

Map-Side Join (Equipartitioned Join)

A map-side join performs the join in the mapper, and avoids the overhead of a reduce phase to sort the outputs. However, this does have the requirement that the two input data sets have been partitioned in a similar way (same number of sorted partitions, with the same keys in the corresponding partitions). Typically this would be achieved when the data sets are produced from map-reduce jobs that partitioned data in the same way. You can find out more about this option here and here.

Reduce-Side Join

A reduce-side join performs the join in the reducer. It is somewhat simpler than the map-side join, avoiding the restrictions on the input data, but it does incur the overhead of sorting in the shuffle-reduce phase. The idea is to have the mapping phase tag the inputs depending on whether they are coming from the page rank or the vertex name data set. The shuffle-reduce phase combines these tagged inputs for each vertex, and it then emits each vertex name with its page rank. The example code provided tags the keys. Alternatively, and more flexibly, you could tag the values. There is a Hadoop library that does this, tagging values with the name of the data set they come from (it requires adding the datajoin library to the classpath), You can see an example of the use of this library here, and you can see another example of tagging the values here.