Fabio Luchetti – 492470

Distributed Enabling Platforms

The PageRank problem: a Hadoop implementation

Project Report

Table of Contents

Project Specification

The aim of this project is the design, implementation and evaluation of the PageRank algorithm, a well-known mathematical procedure at the core of many big-data problems dealing with information network analysis.

The implementation has been done using the Hadoop/Java framework, thus leveraging MapReduce programming model. A private cluster on top of AWS EC2 free tier instances has been built and configured on purpose, in order to run tests and experiments in a real distributed scenario.

General problem definition

A complete treatment of the PageRank algorithm is beyond the scope of this report, so the interested reader is referred to the rich literature available on the web. In order to tackle the problem, for us a simple - but not trivial - definition will be sufficient. Let consider a set of N webpages $\{p_0, p_1, \dots, p_{n-i}\}$ and their linking structure $\{l_0, l_1, l_2, \dots\}$ (the hyperlinks). This can be viewed/modeled as a directed graph with vertices P: $\{p_0, p_1, \dots, p_{n-i}\}$ and edges L: $\{l_0, l_1, l_2, \dots\}$. We enumerate the webpages and associate with page p_j a number between 0 and 1, called the PageRank of p_j: the vector r: $\{r_{p0}, r_{p1}, \dots, r_{p\_n-1}\}$ of PageRanks is a probability distribution, and in a broad sense it measures the importance of the pages in the collection; the bigger the PageRank, the more important the page. The values r_{pj} are defined as:

$$r_{p_i} = d \sum_{p_j \in B_{p_j}} \frac{r_{p_j}}{|p_j|} + (1 - d)$$

where

- d is the so-called damping factor
- B_{pj} is the set of backward links of p_i (i.e. links to p_i)
- $| p_j |$ is the number of forward links of p_j (i.e. links from p_j)

The mathematical foundations of this formula bank on the theory of stochastic processes, by interpreting as an ergodic Markov chain the likelihood that a random surfer – navigating the web through hyperlinks – will arrive at any particular page. But more in general the algorithm may be applied to any collection of entities with reciprocal references.

Proposed algorithm: design choices and parallelization

To calculate the ranks vector, we adopt an iterative approach, in the limit of the number of iterations. A short comment is due before proceeding: please, note that Hadoop is definitely not suitable for iterative processing. The design goals of its creators were different and meant primarily for resilient batch processing of data with (possibly) a single linear scan of the input files, on commodity machines. Instead, here we require that the results of each iteration are written to disk and then read again (even on the same node) over and over the next iterations. By contrast, other technologies (such as Spark or similar) do exist and are best suited for faster Map Reduce iterative processing, ***provided you have*** sufficient memory and are not working on too large datasets.

In the following, the trivial details of the implementation are voluntarily avoided and, if ever necessary, a proper explanation of the business logic of the application is contained in the source code.

Input and output of the whole job are stored in files as a list of pair of numbers (k,v), one for each line, expressing respectively:

IN: *Page_ID <tab> Outlink_ID*
OUT: *Pagerank <tab> Page_ID*

where the output is sorted in ascending order with respect to the pageranks. The work is actually split in four different MapReduce jobs: ParseGraph, CalculateRank, CheckConvergence, SortRank. Let take the images below as a reference example and look at the mentioned jobs with more detail:

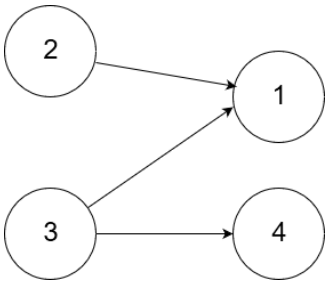


Fig. 1: Simple Web-Graph

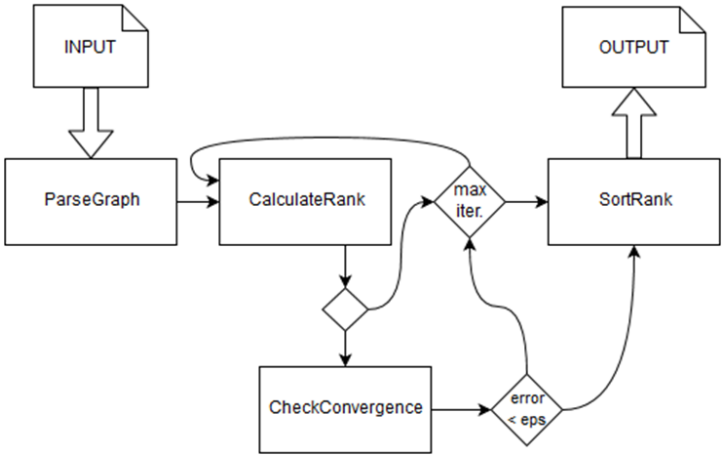
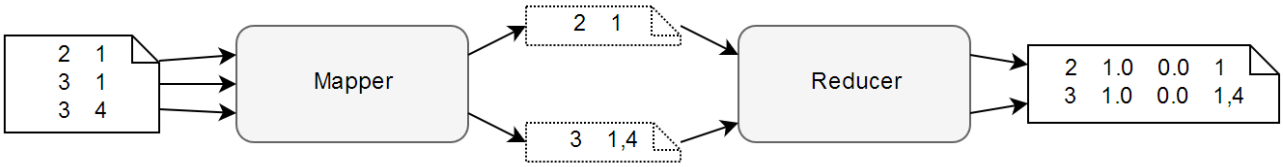


Fig. 2: Workflow chart of the application

ParseGraph, Job 1

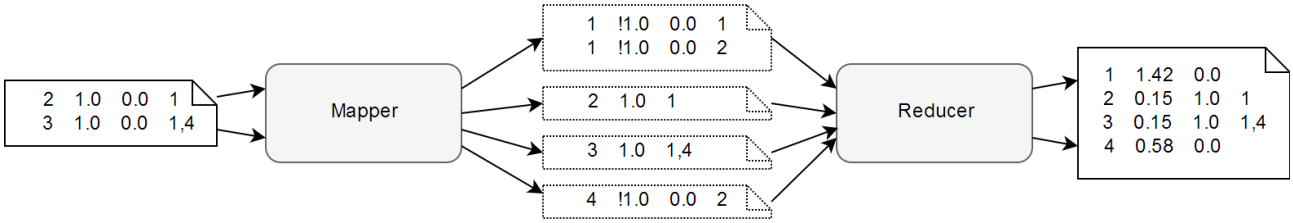


IN: *Page <tab> Outlink*
INTERMEDIATE: *Page <tab> CommaSeparatedOutlinks*
OUT: *Page <tab> Rank <tab> prevRank <tab> CommaSeparatedOutlinks*

Given a web graph, each directed link l=(k,v) going from k to v is represented in the input file by a line with two integer values, representing the ids of the pages. The ParseGraph job captures the outgoing links of each page of the collection and as output it stores, for each page: the page itself, its initial ranks and its adjacency list.

The Mapper phase emits, for each k encountered, the corresponding v; then the Reducer phase stores for each k its ranks - using an initial value of 1.0 for every page (to be coherent with the definition of probability distribution it should have been 1/N, but this unnecessarily over-complicates the algorithm without affecting the final result) - and the list of its adjacent pages separated by commas.

CalculateRank, Job 2



IN:	*Page <tab> Rank <tab> prevRank <tab> CommaSeparatedOutlinks*
INTERMEDIATE:	*OutLink_j <tab> !sourceRank <tab> sourcePrevRank <tab> sourceOutlinkCount (page is not necessary)*
INTERMEDIATE	*Page <tab> Rank <tab> CommaSeparatedOutlinks*
OUT:	*Page <tab> newRank <tab> oldRank <tab> CommaSeparatedOutlinks*

The CalculateRank job constitutes the most expensive part of the program, and it should calculate the pageranks using a simple iterative algorithm.

As we can see above, the job make use of two types of key-value "intermediate" result. In order to calculate the rank contribution of every outlink p_j of a page p, I mapped each p_j to its source rank and total outgoing link count. But it could happen that if some p has no incoming links (thus not being itself the outlink of any page), then p would never be emitted, but rather forgotten as soon as the next iteration. Therefore, we distinguish two possibilities: in one case the key-value entry emitted represents an actual share for the updating pagerank calculation, and we tag this value with the special character '!'; in the other case, the key-value map serves to reproduce correctly the input for the next iterations and to keep track of the ranks. In the reduce phase the new pageranks are evaluated and stored, using as output format the same input format of the mapper phase.

The job keeps being iterated unless a specific convergence criteria is met, or the maximum number of iterations has been reached. For a page p, its PageRank at iteration k+1 is calculated as

$$r_{k+1}(p_i) = d \sum_{p_j \in B_{p_j}} \frac{r_k(p_j)}{|p_j|} + (1-d)$$

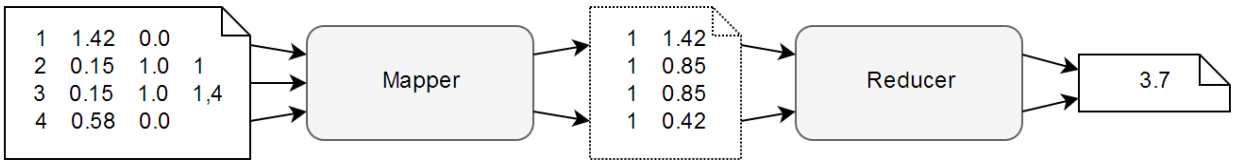
$$r_0 = 1.0$$

Let's break it down into sections:

- r_k: each page has a notion of its own self-importance.
- $|p_j|$: each page spreads its vote out evenly amongst all of its outgoing links. $|p_j|$ represents the count of outgoing links for page p_j.
- $r_k/|p_j|$: so, if a page p has a backlink from page j, the share of the vote it will get is $r_k/|p_j|$

- d: is a double constant (by default set to 0.85), called damping factor. All these fractions of votes are added together but the total vote is "damped down" by multiplying it by d.
- (1-d): if divided by N, this term should make the sum of all the ranks of the pages - meant as a probability distribution - sum to one. It also accounts for the pages that have no links to it (no backlinks), so that they still get a small rank of (1-d).

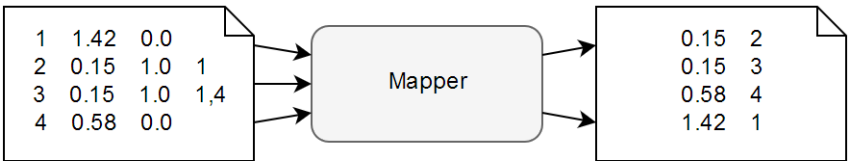
CheckConvergence, Job 3



IN: *Page <tab> Rank <tab> prevRank <tab> CommaSeparatedOutlinks*
INTERMEDIATE *1 <tab> abs(Rank-prevRank)*
OUT: $\sum|r_{k+1}(p_j) - r_k(p_j)|$

As long as the computation keeps going, it could happen that before the last iteration no significant changes applies to the ranks vector. So, to shorten the whole completion time, a convergence test is done periodically upon the current and previous value of the ranks: if the distance in norm L_1 is below a certain accuracy value ($\sum|r_{k+1}(p_j) - r_k(p_j)| < \varepsilon$), the PageRank algorithm stops iterating. Trivially, the map emits a dummy (always identical) key and a value (one per input split) indicating a partial sum, whose terms are the absolute value of the difference among ranks of consecutive iterations. The reduce just sums these quantities and stores the total result. The compliance of this result with the convergence criteria will be checked thereafter, before launching a new iteration of the CalculateRank job.

SortRank, Job 4



IN: *Page <tab> Rank <tab> prevRank <tab> CommaSeparatedOutlinks*
OUT: *Rank <tab> Page*

The final step is quite straightforward. The SortRank job lacks the Reduce phase and takes advantage of the Shuffle-and-Sort subroutine provided by the Hadoop RTS to simply reorder and associate each page with its final PageRank value. As input, it takes the output of the CalculateRank job.

User manual

You will find in the zipped folder all the necessary files to compile and run the developed solution, provided you have settled up a working Hadoop environment. To compile the sources, we exploit Apache Maven building facilities.

Compile
In order to compile, cd' in the project directory and simply run:

```
mvn clean package
```

Run
The program makes use of a set of configurable variables, which have to be passed in input at application launch through the proper flag. These are:

--input (-i)	<input>	The directory of the input file (or the file itself). [REQUIRED]
--output (-o)	<output>	The directory of the output result. [REQUIRED]
--damping (-d)	<damping>	The damping factor [OPTIONAL]. Default is 0.85.
--count (-c)	<max iterations>	The maximum amount of iterations [OPTIONAL]. Default is 3.
--accuracy (-a)	<accuracy>	The estimate of error in norm 1 for the rank vector [OPTIONAL]. Default is 1.0.
--periodicity (-p)	<periodicity>	Checks for convergence every <p> rank-calculation iterations [OPTIONAL]. Default is 2.
--help (-h)		Display a help text.

Finally, to run the application:

```
hadoop jar target/pagerank-1.0-SNAPSHOT.jar pad.luchetti.pagerank.PageRank -i <input>
-o <output> […]
```

You can pick a dataset with the proper input format from the publicly available Stanford web graphs collection.

Tests and datasets

This project includes some bash test scripts that execute an autonomous instance the jobs so far described, in order to test separately each phase of the algorithm. The scripts are stored in the bin/ folder, while some default input files are provided in the data/ folder, together with some other files representing the expected output results. This test dataset could easily be expanded provided that the input/output files added are properly named as:

- datainJ*i*_*x*
- dataoutJ*i*_*x*

where *i*: {1,2,3,4} indicates the Job they refer to and *x* is any alphanumeric string.

The business logic of the scripts is that for every input file found in data/, they load it in the Hadoop hdfs, execute the job, compute and store the result, get-merge it and compare it with the appropriate expected output.