# PAGERANK SPARK IMPLEMENTATION

## Introduction and main steps

The goal of this work is to compute the page rank value for a set of nodes (pages) stored in an input file. For each node, the page rank is determined by the number of **ingoing** links of the considered node, but we can get from the inputs only the list of **outgoing** links for each node. Thus, the steps to compute the page ranks are the following:

1. Decode the input file to get the title of the page and the set of outgoing links
2. Assign to each node the initial page rank value of 1/N (N is number of considered nodes)
3. Compute the contribution that the considered node gives to the pages addressed by its outgoing links
4. Discard the contributions destined to pages that are out of the scope (linked nodes that are not in the original set of considered nodes)
5. Sum the contributions received by each node, apply the page rank formula
6. Go to 3 for a certain number of iterations, then:
7. Sort results by page rank and output them in a text file

## Pseudocode and DAG

Starting from these steps, we can write the Spark pseudocode considering that:

* We decided to save the network structure and the computed page ranks into 2 different RDDs, because the first one will be static for the whole application lifespan, so it is better to modify frequently a small RDD instead of a big one the same number of times
* Every node contributes to itself with a 0 factor, this record is required not to lose the nodes that have no incoming links
* To avoid spurious links, for which concerns the *wiki-micro.txt* dataset we consider only the links in the text section

## Python implementation

## Java Implementation

## Performance optimization

* 2 computing nodes -> partizione iniziale dei dati in 2, impatto sui join

#### Caching

Once every worker has computed its own partition of the graph structure (nodes + outgoing links), it will be going to need this data multiple times across the iterative computation. For this reason, workers are asked to **cache** in memory the ***nodes***RDD (containing the graph structure), so that to avoid the recomputing of it across the iterations. This choice lead to a sensible performance boosting and required time reduction (more than 3 times less in a run with 5 iterations).

With small datasets, partitions of this RDD can fit completely in memory. On the contrary, huge datasets could degrade the performance gain, but in this case the disk persistency of the RDD is still a better choice w.r.t. the full re-computation from scratch.

* Minimizzazione degli shuffle: filter e collect
* MapValue quando le chiavi sono costanti per lasciare intatto il partizionamento