International Journal of Computer Science and Applications
© Technomathematics Research Foundation
Vol. 12 No. 2, pp. 81 - 98, 2015

SCALING UP A DISTRIBUTED COMPUTING OF SIMILARITY COEFFICIENT WITH MAPREDUCE*

MIREL COŞULSCHI, MIHAI GABROVEANU, FLORIN SLABU and ADRIANA SBÎRCEA

Department of Computer Science,
University of Craiova,
13 A.I. Cuza Street, Craiova, 200585, Romania
mirelc@central.ucv.ro, mihaiug@central.ucv.ro, f.slabu@yahoo.com, sbirceaadriana@yahoo.com

The work presented in this paper addresses the design and implementation of a Hadoop application and the experiments performed with this application in order to compute the Jaccard similarity metrics for two very large graphs. The algorithm involved uses the MapReduce programming model, whose aim is to distribute the computing process over several machines in order to reduce the overall running time. As a distributed programming model, MapReduce is one of the most important techniques behind Cloud computing metaphor, focused on data intensive computing in clustered environments.

Hadoop open source framework provides to developers a Java API for implementing applications based on MapReduce programming paradigm. In this philosophy, the main task is divided into several smaller subtasks that can be executed or re-executed on any node in the cluster.

The experimental results presented in this paper were obtained after performing various tests over two large data sets (WEBSPAM-UK 2007 and Slashdot) on a distributed cluster.

Keywords: Jaccard similarity; big data; MapReduce; Hadoop.

1. Introduction

As a result of the fact that almost all organizations produce an enormous amount of data which must be stored, processed and analyzed, the request for proper tools able with dealing with this large datasets becomes more pressing nowadays. The first businesses, which have been impacted by the exponential growth of data were the search engines like Yahoo! and Google, as well as the social networks like Facebook or Twitter. For example, lots of data is regularly generated on Facebook[a]:

- there was more than 864 million daily active users on average, 1.35 billion monthly active users (September 2014);

*A preliminary version of this paper was presented at the International Conference on Information, Intelligence, Systems and Applications (IISA 2014) [Coşulschi *et al.* (2014)]
[a] *By the Numbers: 200+ Amazing Facebook User Statistics*,
`http://expandedramblings.com/index.php/by-the-numbers-17-amazing-facebook-stats/`
[Online; accessed 15-January-2015]

- total number of Facebook friend connections were over 150 billion (February 2013);
- total number of uploaded photos exceeded on September 2013 over 250 billion;
- average daily uploaded photos was over 350 million (February 2013);
- there are more than 300 PB of user data (November 3013).

Big data processing cannot be performed by people or by the existing tools designed to process small sets of data, like database management systems or content management systems, because they are no longer efficient in case the amount of data exceeds a certain limit. Classical databases are useful especially for a structured-type data while big data does not mean only structured data, but also semi-structured or unstructured data. The amount of unstructured data is growing faster than the amount of structured data, about 80% of the public data turning out to be unstructured data.

In 2012, the IDC in its *Sixth annual study of the digital universe*[b] estimates that "*from* 2005 *to* 2020*, the digital universe will grow by a factor of* 300*, from* 130 *exabytes to* 40,000 *exabytes, or* 40 *trillion gigabytes. From now until* 2020*, the digital universe will about double every two years*".

The necessity of developing algorithms, which can process a significant amount of data, becomes more pressing given the data age we live in [White (2012)]. The implementation of such algorithms has become easier due to cloud-computing concept. The ability to store, aggregate, combine data and then use the results to perform a deep data analysis has now become more accessible as trends such as *Moore's Law*[c] in computing, its equivalent in digital storage and cloud computing, continue to lower costs and other technology barriers [McKinsey Global Institute (2011)].

Although the concept of "cloud computing" is old, the definition of this term dates back to 1997 when it was first given by Professor Ramnath Chellappa in his article entitled *Intermediaries in Cloud-Computing: A New Computing Paradigm* [Chellappa (1997)]. Nowadays, there are several definitions of cloud computing, one of most relevant being the one provided by NIST (National Institute of Science and Technology): "*a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*" [Mell and Grance (2011)].

In this paper we have analysed the connections and influences that some nodes have over others using the Jaccard similarity coefficients for two very large graphs, computed with the help of a cluster of machines and a MapReduce application

[b] `http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf` [Online; accessed 15-January-2015].

[c] Moore's law is the observation that throughout the history of computing hardware, the number of transistors on integrated circuits has doubled approximately every two years.

managed by a Hadoop distribution.

The paper is organised as follows: Section 2 presents related works, Section 3 describes the Apache Hadoop, a framework for distributed computation, Section 4 presents the MapReduce programming model while Section 5 introduces some measures of similarity focusing on the Jaccard similarity. In Section 6 is presented the algorithms used to compute the Jaccard index. Section 7 describes the experiments performed and provides an analysis of the results. The paper ends with the conclusions.

2. Related Work

The development and implementation of algorithms able to process huge amounts of data has become more affordable due to cloud computing [Armbrust *et al.* (2010)] and the *MapReduce programming model* [Zhao and Pjesivac (2009)], which enabled the development of some open-source frameworks, such as *Apache Hadoop*.

A very important role in cloud computing evolution is played by Amazon, which offers the mostly accessed online library worldwide. This company introduced two cloud computing services, initially intended only for internal use, and which, in time, become very popular within the IT community thanks to their clever design, which makes web-scale computing easier for developers: `EC2`[d] - for computations and `S3`[e] - for storage, to manage their resources.

MapReduce has gained a widespread popularity mainly due to the development of Apache Hadoop. MapReduce applications are extremely broad, being tested on general distributed computing frameworks with security applications such as botnet detection, spam classification [Caruana *et al.* (2011)] and spam detection [Indyk *et al.* (2013)]. The authors of the last paper experimented their algorithms on the WEBSPAM-UK2007 dataset, which we also considered in this work.

In [Kunegis *et al.* (2009)] the authors analyse the corpus of user relationships of the Slashdot technology news site. The paper explores, among other characteristics, link-level characteristics such as distances and similarity measures.

Many similarity measures having been defined and studied, researchers are now aware of their advantages and disadvantages due to the various comparative studies performed [Rajaraman and Ullman (2012)]. One of these measures, the Jaccard similarity index has various applications [Bank and Cole (2008)], [Engen *et al.* (2011)], [Mulqueen *et al.* (2001)]. In [Machaj *et al.* (2011)] the Jaccard coefficient was chosen among different similarity measures for evaluating their proposed Rank Based Fingerprinting (RBF) localization algorithm. In another work [Leydesdorff (2008)], Leydesdorff reports the results of an empirical comparison of a number of direct and indirect similarity measures: the Spearman rank correlations between the association strength (referred to as the probabilistic affinity or the activity index), the

[d]Amazon Elastic Compute Cloud - `http://aws.amazon.com/ec2/`
[e]Amazon Simple Storage Service - `http://aws.amazon.com/s3/`

cosine, and the Jaccard index. Measures are applied to a smaller data set than the one considered in this paper, consisting of the co-citation frequencies of 24 authors.

Our implementation for the Jaccard similarity index computing is inspired from the work of Bank and Cole [Bank and Cole (2008)].

3. Hadoop Framework

Apache Hadoop[f] is one of the most popular open-source framework offering a Java API, which can be used to develop applications following MapReduce paradigm. Hadoop handles the data processing details, allowing the developer to focus on application logic.

Today, this framework gained an increased popularity. Companies such as Yahoo! [Irving (2010)], *New York Times*, *IBM*, *Facebook*, *Hulu*[g] etc. are now using it. Its adoption is enhanced by the fact that Hadoop based applications can be executed on all OSs: Windows, Linux, Unix and Mac OS, despite that Linux is its official platform.

The base Apache Hadoop framework is composed of the following modules[h]:

- *Hadoop Common* - contains libraries and utilities needed by other Hadoop modules.
- *Hadoop Distributed File System (HDFS)* - a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster.
- *Hadoop YARN* - a resource-management platform, added staring with Hadoop 2.0, responsible for managing compute resources in clusters and using them for scheduling of users' applications.
- *Hadoop MapReduce* - a programming model for large scale data processing.

4. MapReduce Programming Model

MapReduce is a programming model for processing and generating large data sets introduced by two Google's researchers, Jeffrey Dean and Sanjay Ghemawat, in a paper published in 2004, entitled *MapReduce: Simplified Data Processing on Large Clusters* [Dean and Ghemawat (2004)]. This programming model was designed and introduced by Google's researchers earlier, but its implementation was not published, remaining private. The operational element is *job* which in MapReduce splits the input data into many independent blocks of data that can be processed independently.

There are two seminal ideas which can be encountered in *MapReduce*:

[f] `http://hadoop.apache.org/`
[g] `http://www.hulu.com`
[h] `http://en.wikipedia.org/wiki/Apache_Hadoop`

- There is a previous well-known strategy in computer science for solving problems, named *divide-et-impera*: when we have to solve a problem whose general solution is hard to compute, then the problem is decomposed in smaller subproblems, apply the same strategy for it, and combine the results/partial solutions corresponding to each subproblem into a general solution. The decomposing process continues until the size of the subproblem is small enough to be handled individually.
- The basic data structures are $(key, value)$ pair and list. The design of MapReduce algorithms involves usage of $(key, value)$ pairs over arbitrary datasets.

Map and *Reduce* concepts were inspired from functional programming, Lisp having two functions with similar meaning, *map* and *fold*. Functional operations have the property of preserving data structures such that the original data remains unmodified and new data structures are created.

The *map* function takes as arguments a function f and a list, applies the function f to each element of the input list, resulting a new output list.

The *fold* function has an extra argument compared to the *map*: despite the function f, and the input list, there is an accumulator. The function f is applied to each element of the input list, the result is added to previous accumulator obtaining the accumulator value for the next step.

Although the idea seems to be very simple, the implementations details of a *divide-and-conquer* approach are not. There are many details with which the practitioners must deal:

- decomposition of the problem and allocation of subproblems to workers;
- synchronization between workers;
- communication of the partial results and their merge for local solutions;
- management of software/hardware failures;
- management of workers' failures.

MapReduce was designed around two important concepts, the *mapper* and the *reducer*, while the lists of $(key, value)$ pairs are the basic data structure for interprocess communications. *Mapper* and *reducer* are designed as two user implemented functions with the following signatures:

- `Map`: $(k1, v1) \rightarrow [(k2, v2)]$
- `Reduce`: $(k2, [v2]) \rightarrow [(k3, v3)]$.

A mapper is created for every *map task* and a reducer is created for every *reduce task*. The *map* function is applied by mapper to every input pair $(k1, v1)$ generating as output a list of pairs $[(k2, v2)]$ which, at their turn, represent the input data for the reducer. Between the two tasks there is performed a *group by* operation, such as the output from the first task reaches the second task sorted and grouped by a key. Usually, a reducer receives data from more than one mapper leading to a number

of reducers smaller than the number of mappers. The *reduce* function receives a $(k2, [v2])$ pair, where $[v2]$ is a list containing all intermediate values associated with the same intermediate key $k2$, aggregates them and outputs a list of pairs $[(k3, v3)]$.

The *map* and *reduce* instances are distributed across multiple machines from the cluster, in order to be processed in parallel. The keys and values can be of any type, user-defined or not. There is one rule that applies for the mapper's output and reducer's input: the types of keys and values from the mapper's output should match with the types of keys and values of the reducer's input.

A few examples of algorithms developed using MapReduce consist of *pi*, *distributed grep*, *inverted index*, *count of URL frequency* computation etc. [Borthakur (2009)], [Lin and Dyer (2010)], [Rajaraman and Ullman (2012)], [White (2012)].

One of the key features of the MapReduce programming model is that it allows everyone to develop fast algorithms for big data processing [Zikopoulos *et al.* (2011)].

5. Jaccard Similarity Measure

From a simple point of view, measures of similarity denote the closeness among two or more objects. An important class of measures of similarity is represented by distance measures. Let A be a set of elements, called *space*. A *distance measure* defined on A is a function $d(x, y)$, $d : A \times A \to \mathbb{R}$, having as arguments two elements from this space and returning as output a real number, which satisfies the following four axioms:

a) $d(x, y) \geq 0$ (the distance between two elements is always a positive number).
b) $d(x, y) = 0 \Leftrightarrow x = y$.
c) $d(x, y) = d(y, x)$ (symmetry).
d) $d(x, y) \leq d(x, z) + d(z, y)$ (known as the triangle inequality).

The similarity notion is very flexible, so there are many distance measures formulas such as the followings: the *Euclidian distance*, the *Jaccard distance*, the *Cosine distance*, the *Edit distance*, *the Tanimoto distance* etc. [Rajaraman and Ullman (2012)].

One of the most popular measure used to evaluate the similarity between two data sets is *Jaccard index* (or *Jaccard similarity coefficient*) defined as the ratio between the cardinal of the intersection of two data sets and the cardinal of the reunion of the same two data sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}. \tag{1}$$

Jaccard index is not a distance measure and it can be applied to objects having binary attributes.

The function defined as the difference between 1 and the Jaccard index is a real measure distance known as the *Jaccard distance*. It measures the degree of dissimilarity between two data sets, defined as the complementary of the Jaccard

coefficient:

$$J_\delta(A,B) = 1 - J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}. \quad (2)$$

A few examples of how the Jaccard similarity coefficient can be used, are the documents similarity, the online shopping, the movie ratings, the similarity of the users from social networks etc.

Example 5.1. *Let's consider two web sites denoted with A and B. We want to evaluate the degree of similarity for these two web sites influenced by the fact that they share same host links in their web pages. We will consider only the host name for a page-to-page URL link present in the anchor tag from a web page .*

Let's suppose that Host1, Host2, Host3, Host4 are all host names referenced by the URL links from web pages from both websites. Table 1 shows the links encountered in the web pages from the web sites A and B.

Table 1. Links distribution for sites A and B

Site	Host1	Host2	Host3	Host4
A	yes	yes	yes	yes
B	no	yes	no	no

The site A is having the neighborhood set $N_A = \{Host1, Host2, Host3, Host4\}$ *while the site B is having the following neighborhood set:* $N_B = \{Host2\}$. *Thus we:*

$$J(A,B) = = \frac{|N_A \cap N_B|}{|N_A \cup N_B|} = \frac{1}{4}$$
$$J_\delta(A,B) = 1 - J(A,B) = 1 - \frac{1}{4} = \frac{3}{4}$$

6. Algorithm Design

An informed reader would have been noticed the presence of graphs data structures almost everywhere starting with social networks like Facebook[i], MySpace[j], NetFlix[k] and ending with transportation routes or Internet infrastructure architecture. These graphs keep growing in size and there are a lot of new things people would like to infer from the many facets of information stored in this data structures.

The graphs can have billions of vertices and edges. Citation graphs, affiliation graphs, instant messenger graphs, phone call graphs are part of the so called social

[i] `http://developers.facebook.com/`
[j] `http://wiki.developer.myspace.com/index.php?title=Main_Page`
[k] `http://developer.netflix.com/docs`

network analysis. In the paper [Leskovec *et al.* (2005)] it is shown that despite the previously supposed sparse nature of those graphs, their density increase over time.

In this section, we will follow the algorithms steps as they appear in Bank and Coles work. The developed application is composed of two parts aiming to compute the Jaccard index and its frequency.

The first part is represented by the computation of the Jaccard index between any two vertices of the input graph. This process can be decomposed into three stages:

(1) In the first stage, the input data is extracted from a file: each line represents one edge of the graph described by its two vertices. As in MapReduce special notations, the first vertex represents the *key* k, while the second one is the *value* v. The first job's mapper (see Algorithm 1) just parse each line of the input file and emits (k, v) pairs.

Algorithm 1 XYMapper

INPUT:

k - id of the current node
v - node's information

```
1: class MAPPER
2:     method MAP(key = k, value = v)
3:         EMIT(k, v)                          ▷ emit the updates for current node
4:     end method
5: end class
```

The *CountCardinalReducer* (see Algorithm 2) computes for each input *key* k the cardinality of its *values'* set. The set L ($L = [v_1, v_2, \ldots]$) stands for the neighborhood of the input vertex (whose *value* is kept by the key k). For each pair (k, v_i) it is emitted a new pair $(k, \{v_i, size(L)\})$, where v_i represents a vertex from the neighborhood list of the vertex k, and will be further used for neighborhood's lists intersection of any two graph's vertices.

Algorithm 2 CountCardinalReducer

INPUT:

k - id of the current node
L - neighborhood list of node k

```
1: class REDUCER
2:     method REDUCE(key = k, value = L = [v_1, v_2, ...])
3:         for all v ∈ L do
4:             EMIT(k, {v, size(L)})
5:         end for
6:     end method
7: end class
```

(2) In the second stage of the algorithm, for each key-value (k, v) pair, the *key* is inverted with the *value*, contributing to the computation of the intersection between the neighbors' sets for any two vertices, in the last stage of the algorithm.

The second job's mapper (see Algorithm 3) receives as input the values $(k, \{v, count\})$ computed by the reducer of the first job. The vertices' positions inside the input are switched: the mapper emits a new *key-value* pair having the format $(v, \{k, count\})$ where v is the vertex contained in the *value* part processed by the previous reducer, while the *value* of this new pair $(value \leftarrow \{k, count\})$ is composed of the *key* k of the input pair and the cardinal *count*.

Algorithm 3 FromXYToYXMapper

INPUT:

k - id of the current node
v - a node from the neighborhood set of the current node
count - cardinal of the neighborhood set of the current node

```
1: class Mapper
2:     method Map(key = k, value = {v, count})
3:         Emit(v, {k, count})
4:     end method
5: end class
```

The second reducer (see Algorithm 4) receives as input a vertex of the graph as the *key* and a list of pairs containing vertices and sets' sizes. A maximal number of values are grouped together on a string. For each string, either having maximum size or with the remainder values, the reducer emits a pair consisting of an empty string as the *key*, and the built string as the *value*. The *key* will not be used in the next step.

Algorithm 4 SplitReducer

INPUT:

k - id of the current node
L - node's list information

```
1: class Reducer
2:     method Reduce(key = k, value = L = [{v1, n1}, {v2, n2}, ...])
3:         str ← ""
4:         for all {vi, ni} ∈ L do
5:             str ← str ⊕ (vi, ni)
6:         end for
7:         newValue ← str
8:         newKey ← ""
9:         Emit(newKey, newValue)
10:    end method
11: end class
```

(3) The last job from this processing part computes the Jaccard index for each pair of vertices.

The map method receives an empty string as the *key*, and a string *txt* as the *value* (see Algorithm 5). The elements from the string *txt* are extracted and stored in a list L. Further on, it will be constructed all pairs $\{u_1, u_k\}$, $k > 1$ from the first element of the list L, u_1, and each other element of the list L, u_k, $k > 1$. The method emits pairs having as the *key* a pair of graph nodes $\{u_1, u_k\}$ and as the *value*, the sum of the cardinals of the neighborhood sets $n_1 + n_k$.

Algorithm 5 CollectAllPairsMapper

INPUT:

k - id of the current node
txt - text string

```
1: class Mapper
2:     method Map(key = k, value = txt)
3:         L ← ∅
4:         while (txt.hasMoreTokens()) do
5:             x ← parse(txt.nextToken())
6:             c ← parse(txt.nextToken())
7:             L ⇐ {x, c}
8:         end while
9:         L ⇒ {x, n}
10:        while (L ≠ ∅) do
11:            L ⇒ {y, m}
12:            if (x < y) then
13:                Emit({x, y}, n + m)
14:            else
15:                Emit({y, x}, n + m)
16:            end if
17:        end while
18:    end method
19: end class
```

The last reducer (see Algorithm 6) receives a key which contains a pair with two vertices $\{u, v\}$ whose neighborhood lists have a non-empty intersection, and a list of values, each value representing the sum of the neighborhood sets' cardinals $L = [m, m, \ldots]$. It is counted the cardinal of the list L - this value represents the cardinal of the set obtained from the intersection between the neighborhood of the vertex u and the neighborhood of the vertex v. The Jaccard index of the pair $\{u, v\}$ is the ratio between the cardinal of the list L and the difference between the sum of the cardinals of the neighborhood sets and the cardinal of the intersection.

Algorithm 6 ComputeJaccardReducer

INPUT:

$\{u, v\}$ - a pair with two vertices
L - list with values representing the sum of the cardinals of the neighborhood sets

```
1: class Reducer
2:     method Reduce(key = {u, v}, value = L = [m, m, ...])
3:         n ← size(L)
4:         Emit({u, v}, n/(m − n))
5:     end method
6: end class
```

The second part is represented by an algorithm that computes the frequency of the Jaccard index values for a specified graph. The algorithm consisting of a single MapReduce job.

The mapper (see Algorithm 7) gets its data from the input file, as $(key, value)$ pairs, where each line from the input file is sent as the *value* element and contains the values correspondind to id's of two vertices and their Jaccard index value, $u\ v\ r$. The method emits a pair $(k, newValue)$ where k represents the value of the Jaccard index associated to the pair of vertices $u\ v$, and $newValue$ is a string consisting of the two nodes, separated by a comma.

Algorithm 7 NodesToJaccardMapper

INPUT:

k - dummy key

txt - input text

```
1: class Mapper
2:     method Map(key = k, value = txt)
3:         u ← parse(txt.nextToken())
4:         v ← parse(txt.nextToken())
5:         r ← parse(txt.nextToken())
6:         Emit(r, u + "," + v)
7:     end method
8: end class
```

The reducer (see Algorithm 8) receives for a value of the *key* k all pairs of nodes having the same value for the Jaccard index. It counts the elements from the input list and emits a pair $(k, count)$ where k contains the value of Jaccard index and *count* represents the cardinal of the input list.

Algorithm 8 CountJaccardValuesReducer

INPUT:

k - a value of Jaccard index

L - list of strings

```
1: class Reducer
2:     method Reduce(key = k, value = L = [v_1, v_2, ...])
3:         count ← size(L)
4:         Emit(k, count)
5:     end method
6: end class
```

7. Experimental Results

The first experiments were performed on a large set of data named WEBSPAM-UK2007[l]. From this set we have used the *hostgraph*, which represents a very large collection of annotated spam/non-spam hosts, containing 114, 529 hosts, the node numbering starting from 0. The input file for the MapReduce algorithm implementation contains the hosts graph, which summarizes the URL to URL links by converting from multiple links on different web pages into a single link between two hosts. A line from the input file has the following generic structure:

```
dest_1:nlinks_1 dest_2:nlinks_2, ..., dest_k:nlinks_k
```

where `dest_i` represents the destination host, and `nlinks_i` represents the number of page-to-page links between the host having the *id* equal with the line number. When a host does not have any link with another host, its corresponding line is empty.

We have processed it in order to obtain a new file containing the adjacency list of the graph of hosts. Every line from the original file is parsed having as a result a set of lines, each one corresponding to the pair [current host - every host on the original line].

The application was run on a virtual cluster, consisting of six virtual machines. The master machine, hosted the *NameNode*, the *Secondary NameNode* and the *JobTracker* daemons, while the *DataNode* and the *TaskTracker* daemons ran on the other five machines from the cluster, the slaves. Every virtual machine had 1 GB of RAM, a 1.99 GHz dual core processor and a hard drive with a capacity of 25 GB. The virtual machines were launched and managed using *VMWare ESX* 5.0[m]. The Hadoop version installed on these machines was *Hadoop 0.18.0* [Coşulschi *et al.* (2012)], [Coşulschi *et al.* (2014)].

The size of the input file for the application was 16 MB, while the size of the result file was over 2 GB. A few lines from the output file are listed in table 2.

The Jaccard index histogram is represented in figure 1, where one can see that

[l] "Web Spam Collections", Crawled by the Laboratory of Web Algorithmics, University of Milan, `http://chato.cl/webspam/datasets/` [Online; accessed 15-January-2015]

[m] `http://www.vmware.com/products/esxi-and-esx/`

Table 2. Values of the Jaccard index for sample pairs of nodes from the WEBSPAM-UK2007 graph.

$Node_1$	$Node_2$	Jaccard index
100024	100012	0.023121387
100024	100021	0.018365473
100059	100007	0.1
100062	100018	0.045454547
100064	100021	0.001897533
100083	100077	0.33333334
100088	100010	0.03448276
100107	100104	0.25
100110	100007	0.1
100110	100012	0.008264462
100110	100092	0.030303031

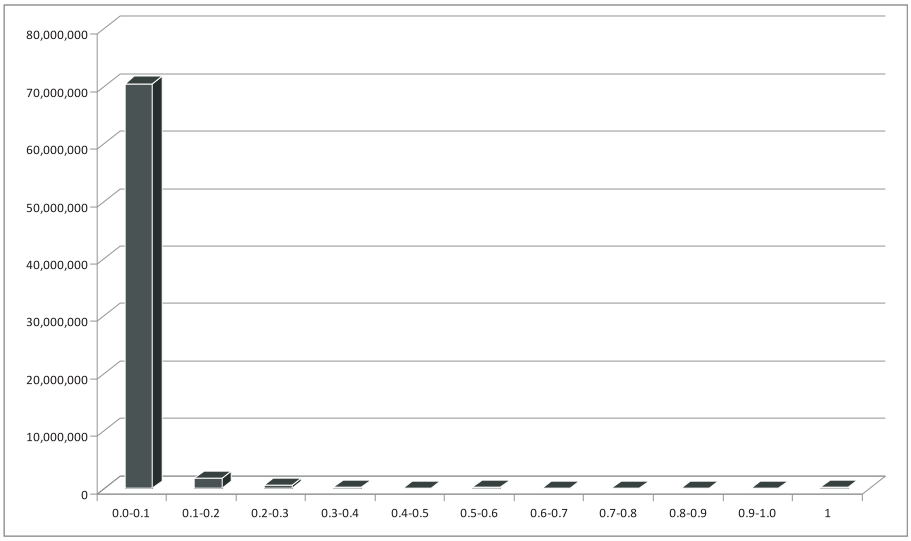

Fig. 1. The Jaccard index histogram for the WEBSPAM-UK2007 data collection

most nodes from the host graph have the Jaccard similarity coefficient included in the $[0.0, 0.1)$ interval. This interval contains over $70,000,000$ indexes, thus most nodes from our graph are completely different. As regards the identical nodes from the similarity point of view, there are more than $150,000$ indexes having values 1, while over $50,000$ are almost identical, with the Jaccard coefficient included in

$[0.9, 1.0)$ interval. In order to further explore the $[0.0, 0.1)$ interval, it was split into smaller intervals, the results being displayed in figure 2.

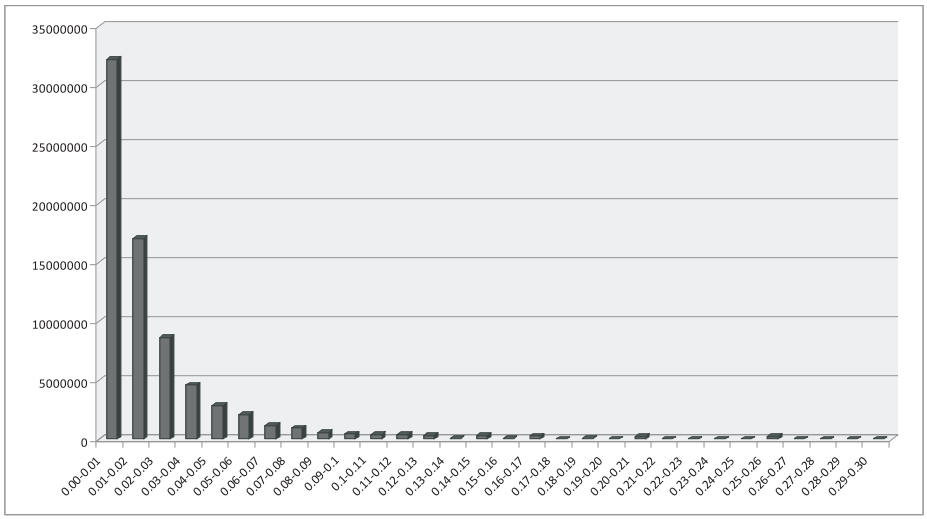

Fig. 2. The Jaccard index values distribution over $[0.0, 0.3)$ interval for the WEBSPAM-UK2007 dataset

In this figure, one can notice that most of the sites from the chosen dataset have one or two common links for every 100 adjacent sites. The distribution of the Jaccard index is very heterogeneous due to the chosen dataset. Lower values are obtained depending on the number of sites that have the Jaccard similarity coefficient values from $0.13 - 0.14$ (for sites having 13-14 links in common on every 100 neighbors).

The second test was performed on a different dataset, i.e. Slashdot, which was collected in February 2009[n]. Slashdot is a technology-related news website[o], offering a Slashdot Zoo feature that enables users to tag each other as friends or foes. The corresponding graph contains friend/foe links between the Slashdot users. The file containing the dataset has on each line a pair of nodes, separated by space, with the structure $node_1$ $node_2$. The dataset contains $82,168$ nodes and $948,464$ edges [Leskovec *et al.* (2009)].

The histogram corresponding to the sequence of values for all the Jaccard indexes can be seen in figure 3. The results are similar to the ones obtained from the first dataset, the majority pairs of nodes from the graph having the values of the Jaccard index included in the $[0.0, 0.1)$ interval.

[n] `http://snap.stanford.edu/data/soc-Slashdot0902.html` [Online; accessed 15-January-2015]
[o] `http://slashdot.org/`

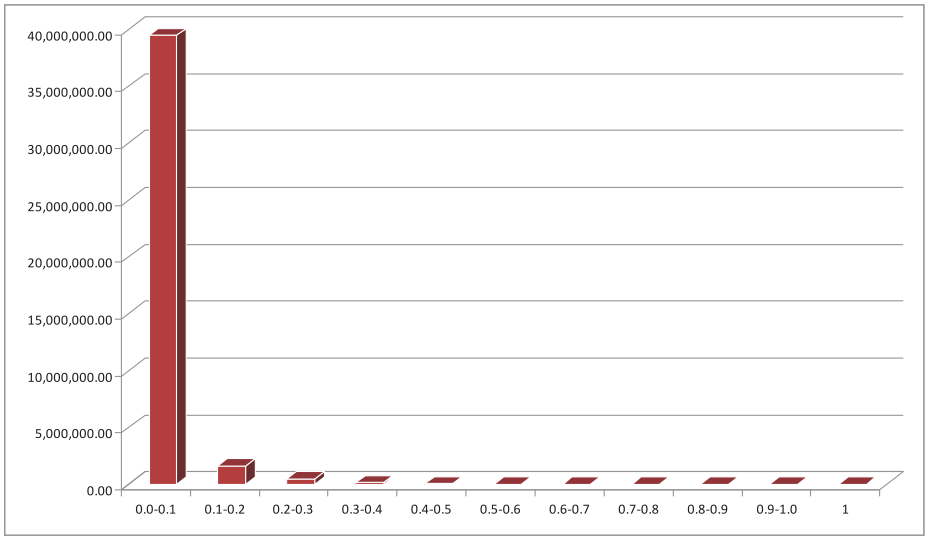

Fig. 3. The Jaccard index values distribution for the Slashdot graph.

Table 3. Values of the Jaccard index for sample pairs of nodes from the Slashdot graph.

$Node_1$	$Node_2$	Jaccard index
10002	10001	0.04347826
10003	100	0.018072288
10003	10001	0.004651163
10004	0	0.011111111
10004	100	0.015957447
10004	10001	0.014705882
10005	1	0.005291005
10005	10001	0.060606062
10005	10004	0.012658228
10006	100	0.009615385
10006	10002	0.011494253
10006	10005	0.042105265
10007	10005	0.05882353
10008	100	0.011278195

Similarly, in order to have a closer look at the values distribution, the interval $[0.0, 0.3)$ was split into smaller intervals. Figure 4 presents the histogram for this interval. As a conclusion, the Slashdot graph has many pairs of nodes with the

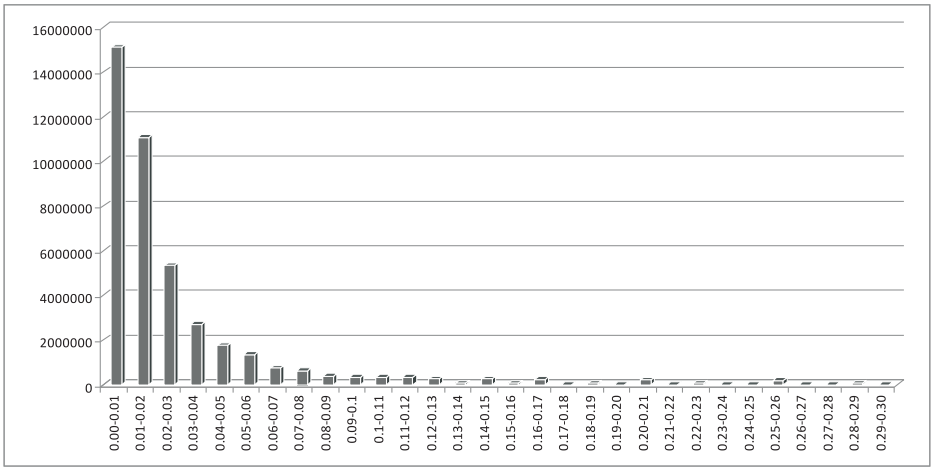

Fig. 4. The Jaccard index values distribution over $[0.0, 0.3)$ interval for the Slashdot data set

corresponding values of the Jaccard index belonging to the $[0.0, 0.1)$ interval, which means that most of its nodes are very different from the point of view of the similarity functions used: e.g. they have little nodes in common. The lower frequency values of histogram are located in the $[0.24, 0.25)$ and $[0.29, 0.3)$ intervals (for friends having $24 - 25$ and $29 - 30$ links in common on every 100 neighbors). A few values of the dataset are listed in table 3.

8. Conclusion

As regards the WEBSPAM-UK2007 data collection, let's consider two hosts u_1 and u_2 with A the set of neighbouring hosts with u_1 (hosts that have at least a web page pointed by a link from a web page from u_1) and B the set of neighbouring hosts with u_2. Jaccard index for the hosts u_1 and u_2 represents the measure of overlap that sets A and B share. In other words, this value represents the number of common neighbouring hosts reported to the total number of neighbouring hosts.

If two hosts u_1 and u_2 have the Jaccard index ≈ 1, then:

(1) we can conclude that their degree as *hub* is similar considering only the number of hosts they refer to (we consider the *hub* notion as introduced for the HITS algorithm [Kleinberg (1999)]);
(2) it could be a good measure for classifying hosts as spam/non-spam (if a host was already classified as *spam*, then all the hosts having the Jaccard index ≈ 1, are good candidates for the same labeling);
(3) the two hosts are connected from the semantic point of view (if they have a lot of neighbouring hosts in common, then the information that they are hosting is similar).

On the other side, a Jaccard index ≈ 0 means that the two hosts have less in common from the semantic point of view.

Regarding the Slashdot dataset, the set of neighbours of a node u can be interpreted as a group of people whom u is already familiar with. Thus the Jaccard index is an indicator of the level of commonly known people. A pair of persons having the value of the Jaccard index greater than 0.9, means that there are strong connections between them at the professional level or that the respective people have common interests. Also, a level lower than 0.1 can be seen as an indicator that the respective two persons have less in common.

We can create a new graph on the basis of the Jaccard index with the same set of vertices, while the set of edges consist of edges between two elements u and v if and only if the Jaccard index computed for the two nodes has a value greater than a threshold value ($J(u, v) > \alpha$, where e.g. $\alpha = 0.9$).

The main goal of this work was to develop a distributed application to compute one of the most popular similarity measure, the Jaccard coefficient, and to use it for evaluating the similarity or dis-similarity degree for vertices from very large graphs. The application is based on the MapReduce programming model, which allows processing of large datasets. The implementation was deployed into a Hadoop cluster where we performed experiments using two real world datasets, while the results were analyzed and interpreted.

Acknowledgments

This work was partially supported by the strategic grant POSDRU/159/1.5/S/133255, Project ID 133255 (2014), co-financed by the European Social Fund within the Sectorial Operational Program Human Resources Development 2007-2013.

References

Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, D. (2010). *A view of cloud computing*, Communication of the ACM, 53(4):50–58 .

Borthakur, D. (2009). *Hadoop Architecture and its Usage at Facebook*, `http://borthakur.com/ftp/hadoopmicrosoft.pdf` [Online; accessed 15-January-2015].

Caruana, G., Li, M. and Qi, M. (2011). *A MapReduce based parallel SVM for large scale spam filtering*, In: 8th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), vol. 4, IEEE: 2659-2662

Chellappa, R. (1997). *Intermediaries in cloud-computing: A new computing paradigm*, in INFORMS Dallas 1997 Cluster: Electronic Commerce, Dallas, Texas.

Bank, J. and Cole, B. (2008). *Calculating the Jaccard Similarity Coefficient with Map Reduce for Entity Pairs in Wikipedia*, `http://www.weblab.infosci.cornell.edu/papers/Bank2008.pdf` [Online; accessed 14-June-2014].

Coşulschi, M., Gabroveanu, M., and Sbîrcea, A. (2012). *Running Hadoop applications in virtualization environment*, Annals of the University of Craiova, Mathematics and Computer Science Series, 39(2): 322-333.

Coşulschi, M., Gabroveanu, M., Slabu, F. and Sbîrcea, A. (2014). *Experiments with comput-*

ing similarity coefficient over big data, In: 5th International Conference on Information, Intelligence, Systems and Applications (IISA 2014), IEEE: 112-117.

Dean, J. and Ghemawat, S. (2004). *MapReduce: simplified data processing on large clusters*, In: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation (OSDI04), 6: 137-150.

Engen, S., Grøtan, V. and Sæther, B.-E. (2011). *Estimating similarity of communities: a parametric approach to spatio-temporal analysis of species diversity*, Ecography, 34:220-231.

Indyk, W., Kajdanowicz, T., Kazienko, P. and Plamowski, S. (2013). *Web Spam Detection Using MapReduce Approach to Collective Classification*, In: International Joint Conference CISIS/ICEUTE/SOCO Special Sessions, Springer, 189:197-206.

Irving, B. (2010). *Big data and the power of Hadoop*, Yahoo! Hadoop Summit.

Kleinberg, J. (1999). *Authoritative sources in a hyperlinked environment*, Journal of the ACM, 46(5): 604-632.

Kunegis, J., Lommatzsch, A. and Bauckhag, C. (2009). *The Slashdot Zoo: Mining a Social Network with Negative Edges*, In: Procedings of World Wide Web Conference, pp. 741-750.

Leskovec, J., Kleinberg, J. and Faloutsos, C. (2005) *Graphs over time: Densification laws, shrinking diameters and possible explanations*, in Proceedings of the eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD'05), ACM, 2005, pp. 177–187.

Leskovec, J., Lang, K., Dasgupta, A. and Mahoney, M. (2009). *Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters*, Internet Mathematics, 6(1):29–123.

Leydesdorff, L. (2008). *On the Normalization and Visualization of Author Co-Citation Data: Saltons Cosine versus the Jaccard Index*, Journal of the American Society for Information Science and Technology, 59(1): 77-85.

Lin, J. and Dyer, C. (2010). *Data-Intensive Text Processing with MapReduce*, Morgan & Claypool Publishers.

Machaj, J., Pich, R. and Brida, P. (2011). *Rank Based Fingerprinting Algorithm for Indoor Positioning*, International Conference on Indoor Positioning and Indoor Navigation (IPIN), pp. 1-6.

McKinsey Global Institute. (2011). *Big data: The next frontier for innovation, competition, and productivity.*

Mell, P. and Grance, T. (2011). *The NIST Definition of Cloud Computing*, National Institute of Science and Technology.

Mulqueen, C. M., Stetz, T. A., Beaubien, J. M. and O'Connell, B. J. (2001). *Developing Dynamic Work Roles Using Jaccard Similarity Indices of Employee Competency Data*, Ergometrika, 2:26-37.

Rajaraman, A. and Ullman, J. D. (2012). *Mining of Massive Datasets*, Cambridge University Press, `http://www.mmds.org/` [Online; accessed 15-January-2015].

Zikopoulos, P., Eaton, C., DeRoos, D., Deutsch, T. and Lapis, G. (2011) *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*, McGraw-Hill.

White, T. (2012). *Hadoop: The Definitive Guide. Storage and Analysis at Internet Scale*, 3rd Edition, O'Reilly Media / Yahoo Press.

Zhao, J. and Pjesivac-Grbovic, J. (2009). *MapReduce – The Programming Model and Practice*, Sigmod 2009 Tutorial, `http://research.google.com/archive/papers/mapreduce-sigmetrics09-tutorial.pdf` [Online; accessed 15-January-2015].