Computing Scientometrics in Large-Scale Academic Search Engines with MapReduce

Leonidas Akritidis and Panayiotis Bozanis

Dpt. of Computer & Communication Engineering, Univ. of Thessaly, Volos, Greece

Abstract. Apart from the well-established facility of searching for research articles, the modern academic search engines also provide information regarding the scientists themselves. Until recently, this information was limited to include the articles each scientist has authored, accompanied by their corresponding citations. Presently, the most popular scientific databases have enriched this information by including *scientometrics*, that is, metrics which evaluate the research activity of a scientist. Although the computation of scientometrics is relatively easy when dealing with small data sets, in larger scales the problem becomes more challenging since the involved data is huge and cannot be handled efficiently by a single workstation. In this paper we attempt to address this interesting problem by employing MapReduce, a distributed, fault-tolerant framework used to solve problems in large scales without considering complex network programming details. We demonstrate that by setting the problem in a manner that is compatible to MapReduce, we can achieve an effective and scalable solution. We propose four algorithms which exploit the features of the framework and we compare their efficiency by conducting experiments on a large dataset comprised of roughly 1.8 million scientific documents.

1 Introduction

Following the evolution of the Web search engines, the scientific databases and academic search engines have significantly enriched the content of their result pages. Therefore, the results of a query for a research paper are now accompanied by information regarding the articles' authors. Some of the most popular scientific search engines such as Google Scholar[1], Microsoft Academic[2], and Scopus[3] extended this information by constructing author profiles where they compute and present their associated *scientometrics*.

The scientometrics are single, scalar values introduced with the aim of evaluating the research work of a scientist. The first and most widespread among them is *h-index*, devised by J.E. Hirsch [9]. This metric assigns a value to each scientist by taking into account not only the number of publications he/she has authored, but also, by considering the number of citations each article received.

[1] http://scholar.google.com
[2] http://academic.research.microsoft.com
[3] http://scopus.com

After the introduction of h-index, an entirely new line of research was drawn and multiple variants were proposed.

The computation of scientometrics is relatively easy when it is performed on small, well-controlled collections of research papers. For instance, in the case of h-index, the evaluation mechanism just needs to determine the articles each researcher has authored and then enumerate all their incoming citations. However, when the size of the collection increases the evaluation becomes more complex since a single workstation cannot accommodate all the involved data (i.e. documents, authors and citations). Therefore, we either have to use a secondary (and slower) type of storage, or solve the problem in parallel by distributing the data to a number of interconnected machines.

MapReduce is a distribution framework designed for solving problems in large scales. It is mainly oriented towards fault-tolerance, distributed storage, and simple implementation without requiring network programming details. This model has been used extensively by the Web search engines to develop a wide range of parallel algorithms. Examples include data mining tasks, information extraction from graphs, data structures construction, text processing, and others.

In this paper we propose four methods based on MapReduce to compute in parallel the scientometrics in large scientific databases. To the best of our knowledge, this is the first work in the current literature attempting to address this problem in large scales. All previous bibliography does not study in depth the issue in question, since until recently the data collections were small and the problem was not very important. However, the introduction of the large scientific databases and their constantly expanding repositories in combination with the users' increased interest, has rendered the issue much more challenging.

The rest of the paper is organized as follows: In section 2 we describe the most important related work about MapReduce and we refer to some popular scientometrics. Section 3 contains the key elements deriving from the related theory. In section 4 we set the problem in a basis that renders it manageable by MapReduce and we design the execution plan of our proposed algorithms. The experimental evaluation of our methods is given in section 5 and finally, in section 6 we conclude our work.

2 Related Work

In this section we present some fundamental articles about MapReduce and its architecture and we discuss some remarkable works which introduce strategies for solving common problems in parallel. Finally, we refer to a number of scientometrics that have been proposed in the related bibliography.

MapReduce was initially introduced by two Google engineers in [7]. In [8] the authors described GFS, the distributed file system on which the framework operates. A more extended presentation of the components of MapReduce is provided in [4]. The most popular open source implementations of MapReduce and GFS are *Hadoop* and the *HDFS* respectively. A technical overview of their architectural logic and design is provided in [3].

Numerous works have proposed expansions and modifications which allowed the framework to be used in a wider variety of applications. For instance, [2] introduced *HadoopDB*, an architectural hybrid between Hadoop and database management systems. In [16] the authors appended a "Merge" phase to the execution plan of the system with the aim of joining the relational outputs of two separate MapReduce tasks.

Several other research articles have described important problems which were efficiently solved by using MapReduce. For instance, Web search engines have used the framework extensively in data intensive tasks such as inverted index construction [13], and PageRank computation [12]. Text intensive applications include duplicate and near-duplicate document detection [6], language processing algorithms [11] and numerous others. Finally, [17] introduced Pregel, a computational model for processing large graphs. Pregel programs are expressed as a sequence of iterations, in each of which a vertex can receive messages sent in the previous iteration. However, unlike PageRank computation, the evaluation of scientometrics can be performed in a single MapReduce job without requiring multiple iterations.

Now let us refer to some papers which are related to scientometrics. The pioneering article in the area is [9], where J.E. Hirsch introduced *h-index*, a metric which evaluates each scientist by rewarding both productivity and influence. Motivated by the success of h-index, several other metrics were proposed. Examples include the *SCEAS* system [15], *g-index* [5] and *f-index* [10]. Additionally, in [14] the authors describe two time-aware variations of h-index, namely the *contemporary h-index* and the *trend h-index*. The former takes into consideration the elapsed time since an article was published, whereas the latter considers the date an article received each of its citations.

3 Preliminaries

Now we briefly present the principles which characterize the framework and we describe its basic components. Furthermore, we revise some of the most popular scientometrics and we examine their attributes.

3.1 MapReduce Basics

MapReduce builds on the key idea of simplicity; that is, its users should not deal with complex network programming issues [7, 4]. Instead, the system provides an abstraction that requires from the algorithm developers to express their solutions by using only two functions: *map* and *reduce*.

The co-ordination of the parallel execution is performed by a single machine, the *Master*. The Master splits the input data into multiple fragments and assigns the processing of each fragment to a number of m *Workers*. The Workers (called *Mappers* in this phase) apply the map function to every key/value pair of their input and generate an arbitrary number of intermediate key/value pairs. When the input is exhausted, the system employs a number of r Workers (now called

Reducers) that apply the reduce function to all values associated with the same intermediate key. Their final output is the solution of the assigned task, also formatted in key/value pairs and partitioned in r shards.

There are two more optional components which can be involved in a MapReduce task: The *Partitioner* and the *Combiner*. The former is used to determine how the intermediate files produced by the Mappers should be transferred to the local file systems of the Reducers. The latter, is used to improve the efficiency of the execution by limiting the size of the data to be transferred from the Mappers to the Reducers by merging the values associated with the same key into associative arrays. The Combiner is deployed by either explicitly declaring a *combine* function, or by properly implementing it within the Mapper itself (*in-Mapper Combiner*). According to [12], the second option is usually preferable.

The MapReduce jobs are executed on top of a distributed file system [7] which transparently addresses all the problems that may occur (e.g., fault tolerance). For instance, in case a worker dies due to a hardware malfunction, Master assigns the job it was processing to another worker without any data loss.

3.2 Scientometrics

Here we provide brief descriptions of some important metrics that have been proposed for evaluating the research work of a scientist. The first and most popular metric among them is *h-index*, defined as follows:

A researcher a has h-index M_h^a, if M_h^a of his/her $|P^a|$ articles have received at least M_h^a citations each and the rest $|P^a| - M_h^a$ articles have received no more than M_h^a citations.

This metric calculates how broad the research work of a scientist is, since it accounts for both productivity and impact. Two interesting generalizations of h-index are the *contemporary* and the *trend* h-indices, both introduced in [14].

The *contemporary* h-index is an attempt to introduce temporal aspects in the evaluation of a scientist's work by taking into account the age of an article. According to its definition, each paper p_i of an author a is assigned a score $S_c^{p_i}$ determined by the following formula:

$$S_c^{p_i} = \gamma \frac{|P_c^{p_i}|}{(\Delta Y_i)^\delta} \tag{1}$$

where γ and δ are two constant coefficients; typical values for them are 4 and -1 respectively. ΔY_i symbolizes the number of the years elapsed since the publication of p_i, whereas $|P_c^{p_i}|$ is the number of the articles citing p_i. This way an old article gradually loses its value, even if it still gets citations. Based on the score $S_c^{p_i}$ the contemporary h-index is defined as follows:

A researcher a has contemporary h-index M_c^a, if M_c^a of his/her $|P^a|$ articles get a score $S_c^{p_i} \geq M_c^a$ and the rest $|P^a| - M_c^a$ articles get a score $S_c^{p_i} < M_c^a$.

Another mechanism for ranking scientists is the *trend* h-index. Here the idea is to assign scores to each paper by considering the year an article acquired a particular citation. This idea is expressed by the following equation:

$$S_t^{p_i} = \gamma \sum_{n=1}^{|P_c^{p_i}|} \frac{1}{(\Delta Y_n)^\delta} \quad (2)$$

where γ and δ are defined as previously. The scores $S_t^{p_i}$ are now used to define the trend h-index:

A researcher a has trend h-index M_t^a, if M_t^a of his/her $|P^a|$ articles get a score of $S_t^{p_i} \geq M_t^a$ and the rest $|P^a| - M_t^a$ articles get a score of $S_t^{p_i} < M_t^a$.

In this paper we examine how the three aforementioned metrics can be evaluated in large scales by employing the MapReduce framework. Nevertheless, the algorithms we present here can also be applied to compute other types of scientometrics (i.e. g-index, f-index) without any additional effort.

4 Computing Scientometrics with MapReduce

In this section we identify the key components of the problem and we design the Map and Reduce functions. In the sequel, we optimize our approaches by introducing in-Mapper Combiners.

4.1 Problem Formulation

Let us begin by introducing P which is the set containing all papers, and A which includes all authors. Each paper $p_i \in P$ contains a reference section encountered towards the end of the manuscript. From this section we extract $P^{p_i} \subset P$ which contains all papers referenced by p_i; for each reference $p_j^{p_i} \in P^{p_i}$ we retrieve all the contributing authors $A^{p_j^{p_i}}$. In Table 1 we summarize all the above notations.

The input of the problem can be considered as a set of (p_i, C^{p_i}) pairs, where p_i represents the integer identifier of an article and C^{p_i} symbolizes its content. Our objective is to construct a list of (a, M_x^a) pairs, where x identifies the metric

Symbol	Meaning
P	The set containing all papers
A	The set containing all authors
p_i	An arbitrary paper $p_i \in P$
C^{p_i}	The textual content of p_i
a_j	An arbitrary author $a_j \in A$
A^{p_i}	The authors who created p_i
P^{a_j}	The papers authored by a_j
P^{p_i}	The papers referenced by p_i
$S_x^{p_i}$	The score of a paper p_i with respect to the metric x
$M_x^{a_j}$	A metric evaluating the work of a_j $M_h^{a_j}$: h-index of a_j $M_c^{a_j}$: contemporary h-index of a_j $M_t^{a_j}$: trend h-index of a_j

Table 1. List of the most frequent symbols

M employed to evaluate each scientist (see last row of Table 1). According to the definitions of all three metrics, it is required that we decompose the required (a, M_x^a) pairs and construct for each author, one pair of the following form:

$$\Big(a, SortedList\Big[\big(p_1, S_x^{p_1}\big), ..., \big(p_N, S_x^{p_N}\big)\Big]\Big) \tag{3}$$

where $S_x^{p_i}$ is the score of p_i with respect to the metric x. In case we are interested in computing h-index, this score merely represents the total number of the incoming citations that p_i received. A detailed analysis on how these scores are computed is provided in subsection 4.2.

According to 3, to calculate the metric values for an author, we first need to identify all the publications he/she has authored, and then compute their corresponding scores. Notice that the elements of this $(paper, score)$ list must be sorted in descending score order to enable fast metric evaluation with a single iteration. In the following section we present four methods to solve this interesting problem by using MapReduce.

4.2 Basic Algorithm Design

We start by feeding the system with the given set of the publications P. According to our previous discussion, we express the input of the Map function in a $(key, value)$ manner by defining (p_i, C^{p_i}) pairs. Within the Mapper, we parse the textual content of each paper $p_i \in P$ and we retrieve all its outgoing references P^{p_i}. For each reference $p_k^{p_i} \in P^{p_i}$ we compute a score $S_x^{p_k}$, according to the metric x we need to evaluate.

For the plain h-index metric, we set the score equal to 1 for all references, thus denoting that the paper $p_k^{p_i}$ has one incoming citation (which of course, is p_i). For the other two metrics, we need to consult equations 1 and 2. Our goal is to properly set the partial scores in the map phase in order to compute the final scores in the reduce phase. For this reason, during the map phase, we set the partial scores recorded in Table 2.

In the sequel, each reference is again parsed and its authors $A^{p_k^{p_i}}$ are identified. For each extracted author, we create one tuple that will be sent to the Reducer and there are two options to format this tuple. The first one (called *method 1*) dictates that we set the author as the key, and create a pair $(paper, score)$ for the value field. Our second option (called *method 2*) is to generate a composite key of the form $(author, paper)$, and place the paper score within the value field. Algorithm 1 illustrates a pseudocode for the map function implementing these two methods.

Metric	**Partial Score**
h-index	$S_h^p = 1$
contemporary h-index	$S_c^p = \gamma/(\Delta Y_p)^\delta$
trend h-index	$S_t^p = \gamma/(\Delta Y_{p,c})^\delta$

Table 2. Setting the partial paper scores in the map phase for various scientometrics

Algorithm 1 Mapper class(es): In case method 1 is used, the framework executes the step 8a. If we use method 2, we need to execute the step 8b.

```
 1: class Mapper
 2:     method map (integer p_i; string C^{p_i})
 3:         P ← ExtractReferences(C^{p_i})
 4:         for all references p ∈ P
 5:             S^p ← ComputeScore(p)
 6:             A^p ← ExtractAuthors(p)
 7:             for all authors a ∈ A^p
8a:                 emit (a, pair[p, S^p])
8b:                 emit (pair[a, p], S^p)
```

Although the map phase is almost identical for methods 1 and 2, the reduce phases must implement a different strategy. Notice that the MapReduce framework guarantees that the data sent from the Mappers arrives at the Reducers in a sorted key order. This gives method 2 an advantage; method 2 implements a *secondary sort*, that is, the data not only is brought to the Reducers in an ascending author order (as holds for method 1), but also, in ascending paper order. This allows a more robust approach of the reduce phase, since we save the cost of *searching* for the incoming papers. To make our state clearer, we provide Algorithms 2 and 3 for the reduce phase of methods 1 and 2 respectively.

Let us discuss Algorithm 2 first. Since the $(paper, score)$ pairs are brought to the Reducers in arbitrary order, we need to store these pairs into a data structure H which will allow us to accumulate the partial paper scores. More specifically, for each value field of the Reducer input, we search in H for the input paper. In case this search fails, we insert the paper along with its corresponding score. In the opposite case, we just accumulate the incoming score to the one which is already stored in H. After all the tuples have been processed, we sort H in a descending score order and we compute the desired metric by iterating through its entries (steps 11–16). The sorting of H can be performed within the main memory of the Reducer since it stores at most a few hundreds entries; the vast majority of the authors has published fewer than 1000 articles.

On the contrary, in Algorithm 3 there is no need for searching; instead, it is only required to allocate an array H to store the paper scores. Since the tuples arrive in sorted order, we just need to compare the paper we are currently processing to the previous one (step 9). In case their identifiers are equal we accumulate their partial scores and update the last record of H. In the opposite case, we store the new paper score in a new position at the end of H. When all the papers of an author have been processed, we repeat the steps 9–17 of Algorithm 2 to compute the desired metric and we proceed with the next author. The final (a, h^a) tuple must be written out in the close method.

Finally, notice that the pair values of Algorithm 2 and the pair keys of Algorithm 3 are not included in the basic data types of MapReduce. Consequently, it is required that we implement additional classes which explicitly define how these data types must be read and written by the framework. Nevertheless, the

Algorithm 2 Method 1, Reducer class

```
1: class Reducer
2:     method reduce (string a; pairs[integer p, float S^p])
3:         H ← new AssociativeArray
4:         for all pair v ∈ pairs[integer p, float S^p]
5:             if v.p ∈ H
6:                 H^p.S ← H^p.S + v.S^p
7:             else
8:                 H.add(v)
9:         sort H in descending S order
10:        integer papers ← 0, metric ← 0
11:        for all pairs ∈ H
12:            papers ← papers + 1
13:            if H^p.S ≥ papers
14:                metric ← metric + 1
15:            else
16:                stop iteration
17:        emit (a, metric)
```

complexity for Algorithm 3 is increased since the custom data type is used in the key; hence, it is required to determine how the system will compare the keys to each other to achieve sorted Mapper output (compareTo method). However, the increased complexity of Algorithm 3 is rewarded with improved execution performance.

4.3 Optimizing the Performance

Despite their difference in tuples formatting, the Mappers of both methods 1 and 2 still emit data to the Reducers each time an author of a paper reference is extracted. Since we do not check whether the key we are currently processing has been previously sent, it is inevitable that we transmit the same key multiple times. This leads to a performance bottleneck due to the increased network traffic caused among the nodes of the system. Here we attempt to address this problem with the support of the Combiners.

The Algorithm 4 shows how we can extend method 1 with the aim of supporting an in-Mapper Combiner. We call this new approach as method 1-C, where the letter "C" signifies the presence of a Combiner. The cornerstone of method 1-C is to avoid multiple emissions of identical author names and thus, save valuable network bandwidth. To achieve this, we first initialize a container data structure H which shall allow us to emit $(author, list[paper, score])$ tuples instead of the simple $(author, (paper, score))$ tuples of method 1. During the references parsing process, each time an author is encountered we perform a look-up in the container (step 10); in case the author is not present in H we insert the record along with the corresponding $(paper, score)$ pair (steps 11–13). In the opposite case, we need to check whether the current reference belongs to the $(paper, score)$ list of the found author. If the search is unsuccessful we store

Algorithm 3 Method 2, Reducer class

```
 1: class Reducer
 2:     method initialize
 3:         string a_prev ← ""
 4:         integer p_prev ← 0
 5:         integer n ← 0
 6:         H ← new Array
 7:     method reduce (pair[string a, integer p]; float S^p)
 8:         if a = a_prev
 9:             if p = p_prev
10:                 H(n) ← H(n) + S^p
11:             else
12:                 H.add(S^p)
13:                 n ← n + 1
14:         else
15:             Perform steps 9–17 of Algorithm 2
16:             H.reset()
17:             a_prev ← a
18:             p_prev ← p
19:             n ← 0
20:     method close
21:         emit (a, metric)
```

the paper and its score in the list (step 16); otherwise, we update the corresponding list record by summing up the new paper score to the stored one (step 18). After all the input data has been processed, the Mapper emits the tuples stored within H to the Reducer via the Close method.

It is immediately obvious that the method 1 generates an immense number of key-value pairs compared to method 1-C. Method 1-C is much more compact since with method 1, the author is repeated for each reference we send to the Reducer. Nevertheless, we need to mention here that there are two side effects deriving from the usage of a Combiner. The first one is the increased memory footprint of the map function due to the allocation of the container data structure. The second is a possible delay in the execution of the map phase due to the double search we perform (one for the author and one for paper). However, in this specific application that we examine, our experiments reveal that this delay is infinitesimal due to the small length of the $(paper, score)$ lists, and that the usage of a Combiner definitely leads to significant acceleration of the entire task.

Finally, we introduce method 2-C where we inject the in-Mapper Combiner approach in method 2. The Algorithm 5 illustrates the basic steps which are similar to those of Algorithm 4. In this case however, the container data structure does not store a list of $(paper, score)$ pairs for each author, but a single cumulative score value per each distinct $(author, paper)$ pair. This minimizes the benefits of using a Combiner because the $(author, paper)$ keys are more numerous than the simple author keys of method 1-C. In addition, notice that the reduce phase in this case is identical to that of method 2.

Algorithm 4 Method 1-C: Improved version of method 1 with Combiners

```
1: class Mapper
2:     method initialize
3:         H ← new AssociativeArray
4:     method map (integer p_i; string C^{p_i})
5:         P ← ExtractReferences(C^{p_i})
6:         for all references p ∈ P
7:             S^p ← ComputeScore(p)
8:             A^p ← ExtractAuthors(p)
9:             for all authors a ∈ A^p
10:                if a ∉ H
11:                    L^a ← new Array
12:                    L^a.add(p, S^p)
13:                    H.add(a, L^a)
14:                else
15:                    if p ∉ H.L^a
16:                        H.L^a.add(p, S^p)
17:                    else
18:                        H.L^a.update(p, +S^p)
19:    method close
20:        for all authors a ∈ H
21:            emit (a, list(p, S^p))
22: class Reducer
23:    method reduce (string a; list[integer p, float S^p])
24:        H ← new AssociativeArray
25:        for all pair v ∈ list[integer p, float S^p]
26:            if v.p ∈ H
27:                H^p.S ← H^p.S + v.S^p
28:            else
29:                H.add(v)
30:        Perform steps 9–17 of Algorithm 2
```

5 Experiments

For the experimental evaluation of our theoretic analysis we employed Hadoop 0.20.2, an open-source implementation of the Google's MapReduce framework. We begin this section with a brief description of our test dataset and we proceed with data size measurements and efficiency assessments.

5.1 Dataset characteristics

Collecting bibliometric data is a challenging task, due to the strict data protection policies applied by the digital libraries. Since crawling is forbidden, we are limited in using only open access document collections. The largest among these collections is the CiteSeerX [1] dataset, an open repository comprised of approximately 1.8 million scientific articles. The dataset is available in three forms: The first one contains the raw text of the publications scattered in 1.8 million plain

Algorithm 5 Method 2-C: Improved version of method 2 with the introduction of in-Mapper Combiners. The Reducer is identical to the one of Algorithm 3.

```
1: class Mapper
2:     method initialize
3:         H ← new AssociativeArray
4:     method map (integer p_i; string C^{p_i})
5:         P ← ExtractReferences(C^{p_i})
6:         for all references p ∈ P
7:             S^p ← ComputeScore(p)
8:             A^p ← ExtractAuthors(p)
9:             for all authors a ∈ A^p
10:                if pair(a, p) ∉ H
11:                    H.add(pair(a, p), S^p)
12:                else
13:                    H.update(pair(a, p), +S^p)
14:    method close
15:        for all pairs (a, p) ∈ H
16:            emit (pair(a, p), S^p)
```

text files. The other two contain certain meta-data of the documents expressed in SQL and XML formats respectively. The raw text format of the articles requires much and intensive effort towards two directions: a) disambiguation of the authors names and b) references extraction. Although these problems are both interesting and challenging, they are out of the scope of this paper. For this reason, we choose to work with the XML formatted dataset.

For each article of the dataset there are one or more small-sized XML files, each of which represents a different version of the same article. The dataset includes in total 3.9 million XML files, however, in our experiments we use only the latest version; consequently, 1.8 million XML files are used. This large number of small-sized files renders the dataset inappropriate for MapReduce, because the underlying distributed file system is designed for optimal performance when dealing with considerably larger files. For this reason, we performed a conversion of the dataset by packing thousands of these XML files into larger binary files. After this process, our "new" dataset was comprised of 432 files of 64 MB each.

5.2 Data Sizes

In this subsection we perform measurements of the data sizes exchanged between the Mappers and the Reducers of our proposed methods. Initially we provide

Statistic	**Value**
Input Records	1,844,272
Input Size	27.6 GB
Output Records	2,865,282
Output Size	39.9 MB

Table 3. Problem input-output statistics

method independent numbers indicating the data sizes involved in the examined problem. The first two rows of Table 3 concern the Mapper input, whereas the last two are connected to the Reducer output. As mentioned, the input consists of approximately 1.8 million articles which occupy in total roughly 27.6 GB. After the processing of the dataset with MapReduce, the system outputs a set of about 2.8 million $(author, metric)$ pairs the size of which touches 40MB.

Table 4 illustrates various statistics; in the first double column we measure the size of the Mapper output of all four examined methods, expressed in number of records and data size in MB. The latter measurement is essential since it reflects the overall size of the data exchanged among the Mapper and Reducer nodes of the cluster. In the next column we record the counts of the Reducers input groups which represent the number of the unique keys which *arrive* at the Reducers. To acquire these measurements, we executed all four methods by employing only one Worker node; Table 4 derives from the report generated by the framework at the end of the task.

Initially we examine the performance of our methods in terms of sizes of the Mapper outputs. The map phase of methods 1 and 2 transmitted in total 36.7 million records occupying roughly 688 MB. On the contrary, the usage of a Combiner in method 1-C decreased these values by 42% (21.7 million records) and 13% (601 MB) respectively. As we anticipated, method 2-C was not equally efficient despite the usage of a Combiner. Compared to methods 1 and 2 we only achieve a reduction in the size of the outputted data by a margin of 6.5%. This is due to the fact that the Combiner of method 1-C lists $(paper, score)$ pairs per each unique author, whereas method 2-C stores one partial score value for each distinct $(author, paper)$ key; the latter key type is much rarer than the former.

The counts of the Reducer input groups reveal that the number of the unique keys which arrive at the Reducers of methods 1 and 1-C is equal to the number of records that depart from it (see third row of Table 3). This is due to the fact that the output of the entire task (i.e. (a, h^a) pairs) has the same key as the Mapper output of these two methods. On the other hand, the tuples produced by the Mappers of methods 2 and 2-C are keyed by using $(author, paper)$ pairs, consequently, the unique keys which arrive at the Reducers increase by a factor of approximately 4.2.

5.3 Efficiency Measurements

In this subsection we evaluate the performance of the four methods. To exhaustively attest the scalability of our algorithms, we measured their running

Method	**Mapper Output**		**Reducer**
	Records	**Size (MB)**	**Input Groups**
method 1	36,687,999	688.4	2,865,282
method 2	36,687,999	688.4	12,260,311
method 1-C	21,736,395	600.8	2,865,282
method 2-C	34,251,437	643.2	12,260,311

Table 4. Record counts and data sizes for the four examined methods

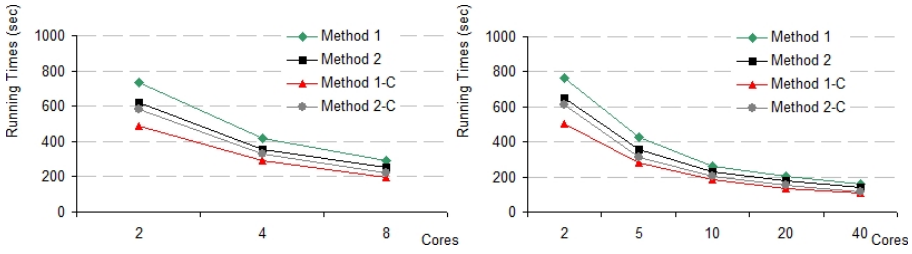

Fig. 1. Running times of the four methods in a small local cluster (Left), and a Web cluster infrastructure (Right).

times by using two platforms. The first one includes a small-sized lab network, whereas the second one is a larger Web cluster infrastructure. Each experiment was repeatedly performed by employing different numbers of processing cores each time. The results are depicted in Figure 1.

Our first observation is that in both platforms, all of our methods scale well for fewer than 20 cores; the doubling of the cluster size almost leads to halved running times. For more cores the gains are slightly limited, due to the increased network latencies. Notice that the running times between the two clusters are not comparable, since these clusters are equipped with different hardware and they adopt different architectures. In all occasions, method 1-C outperformed the other approaches by a margin ranging between 32% and 35%. Apparently, the existence of the Combiner results in decreased exchange of data among the nodes of the clusters. Although method 2-C also employs a Combiner, it did not perform equally well; compared to method 1-C it was outperformed by about 18%. We have previously explained that the $(author, paper)$ keys of method 2-C are more numerous than the simple author keys of method 1-C, consequently, the benefits of using a Combiner are limited.

Regarding the plain methods 1 and 2, we notice that the latter completed the assigned task slightly faster. Although the amount of data exchanged among the nodes of the system is equal in both methods, method 2 achieves better performance due to the more robust implementation of its Reducer. More specifically, the $(author, paper)$ keys emitted by the Mapper of method 2 are brought to the Reducer in sorted author and paper order (secondary sort), thus saving us the cost of searching for the input papers.

6 Conclusions

In this paper we studied the issue of computing several scientometrics in large-scale academic search engines with MapReduce. The scientometrics are scalar values used to evaluate the research work of a scientist. The large volumes of data employed by the modern academic search engines in combination with their popularity, has rendered the examined problem both interesting and challenging. We introduced four methods to compute three of these metrics, h-index, and

two variations, the contemporary and trend h-indexes. However, these methods can be applied to compute a wider variety scientometrics with no additional effort. We proposed optimizations with the aim of decreasing the size of the data exchanged among the nodes of the system, and we conducted experiments with the CiteSeerX dataset, a large repository comprised of about 1.8 million research articles. Our experiments demonstrated the usefulness of method 1-C, a strategy which achieves both effective and efficient execution.

References

1. CiteSeerX Data. In *http://csxstatic.ist.psu.edu/about/data*.
2. A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.
3. D. Borthakur. The Hadoop distributed file system: Architecture and design. 2007.
4. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.
5. L. Egghe. Theory and Practise of the g-index. *Scientometrics*, 69(1):131–152, 2006.
6. T. Elsayed, J. Lin, and D. Oard. Pairwise document similarity in large collections with MapReduce. In *Proceedings of 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies*, pages 265–268, 2008.
7. S. Ghemawat and J. Dean. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating System Design and Implementation (OSDI04), San Francisco, California, USA*, pages 137–150, 2004.
8. S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43, 2003.
9. J. Hirsch. An Index to Quantify an Individual's Scientific Research Output. *Proceedings of the National Academy of Sciences*, 102(46):16569, 2005.
10. D. Katsaros, L. Akritidis, and P. Bozanis. The f index: Quantifying the Impact of Coterminal Citations on Scientists' Ranking. *Journal of the American Society for Information Science and Technology*, 60(5):1051–1056, 2009.
11. J. Lin. Scalable language processing algorithms for the masses: A case study in computing word co-occurrence matrices with MapReduce. In *Proceedings of the Conference on Empirical Methods in Language Processing*, pages 419–428, 2008.
12. J. Lin and C. Dyer. Data-intensive Text Processing with MapReduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.
13. R. McCreadie, C. Macdonald, and I. Ounis. On single-pass indexing with MapReduce. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 742–743, 2009.
14. A. Sidiropoulos, D. Katsaros, and Y. Manolopoulos. Generalized Hirsch h-index for Disclosing Latent Facts in Citation Networks. *Scientometrics*, 72(2):253–280.
15. A. Sidiropoulos and Y. Manolopoulos. A Citation-Based System to Assist Prize Awarding. *ACM SIGMOD Record*, 34(4):60, 2005.
16. H. Yang, A. Dasdan, R. Hsiao, and D. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, 2007.
17. G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD international conference on Management of data*, pages 135–146, 2010.