

# An Efficient Document Indexing-Based Similarity Search in Large Datasets

Trong Nhan Phan<sup>1</sup>(✉), Markus Jäger<sup>1</sup>, Stefan Nadschläger<sup>1</sup>,  
Josef Küng<sup>1</sup>, and Tran Khanh Dang<sup>2</sup>

<sup>1</sup> Institute for Application Oriented Knowledge Processing,  
Johannes Kepler University Linz, Linz, Austria  
{nphan,mjaeger,snadschlaeger,jkueng}@faw.jku.at

<sup>2</sup> Faculty of Computer Science and Engineering,  
HCMC University of Technology, Ho Chi Minh City, Vietnam  
khanh@cse.hcmut.edu.vn

**Abstract.** In this paper, we principally devote our effort to proposing a novel MapReduce-based approach for efficient similarity search in big data. Specifically, we address the drawbacks of using inverted index in similarity search with MapReduce and then propose a simple yet efficient redundancy-free MapReduce scheme, which not only takes advantages over the baseline inverted index-based procedures but also adapts to various similarity measures and similarity searches. Additionally, we present other strategic methods in order to potentially contribute to eliminating unnecessary data and computations. Last but not least, empirical evaluations are intensively conducted with real massive datasets and Hadoop framework in the cluster of commodity machines to verify the proposed methods, whose promising results show how much beneficial they are when dealing with big data.

**Keywords:** Similarity search · Efficiency · Mapreduce · Large datasets · Clustering · Filtering · Redundancy-free capability · Document indexing

## 1 Introduction

While consecutively playing the important role in the wide scopes of applications such as duplicate detection, plagiarism exposure, recommendation systems, data cleaning, data clustering [9], to name a few, similarity search has also to cope with challenges in the era of big data by its “three Vs” characteristics as follows: (1) Volume demonstrates the large amount of data; (2) Velocity denotes the high speed of data; and (3) Variety represents the various data forms [11, 22]. The issue has gained lots of attention and effort whilst there are many studies which never stop experiencing and looking for favorable solutions [3, 6, 8, 13, 15, 16, 19–21]. Most of them, to the best of our knowledge, only concentrate on scalability by employing divide and conquer strategies on parallel mechanisms, such as MapReduce paradigm [5], to deal with enormous data. Many studies [6, 8, 10, 12–16, 19, 20], on the other hand, additionally utilize a data index structure known as an inverted index or a postings list to allow fast text searches, which is widely-used in the area of information retrieval in general and in similarity

search in particular. Nevertheless, inverted index-based methods encounter three main problems when they are performed in MapReduce paradigm as following: (1) Every key-value pair in the inverted index has to be scanned sequentially because of the full-scan manner of MapReduce as well as the structure of the inverted index; (2) Processing data from the inverted index brings much redundancy to identify candidate pairs among documents due to their duplicate values; and (3) It is not convenient to derive the total length of each document for fast set-based similarity computing, like Jaccard or Dice [18, 19] for example, in order to speed up the similarity computing process. These problems implicitly lead to complicated data processing and affect the overall performance. Motivated from finding out an efficient similarity search under the big data context, we propose a novel MapReduce-based approach, in this paper, not only to support resolving scalability but also to take care of data redundancy and intensive data-driven processing manners which originally exist in MapReduce paradigm. Other than improving the overall performance of similarity search, our goal basically aims at what various kinds of applications might benefit and facilitate from our methods. Hence, our main contributions can be generally summarized as follows:

1. We address the three common problems with which inverted index-based methods usually encounter.
2. We then propose a simple yet efficient redundancy-free MapReduce scheme, which not only overcomes the problems from the baseline inverted index-based procedures but also has its adaptability to diverse similarity measures as well as different similarity searches such as pairwise similarity, range query, and K-Nearest Neighbor (K-NN) query.
3. We consider promising strategies that contribute to eliminating dissimilar candidate pairs and unnecessary computations as well as diminishing data redundancy throughout MapReduce processes in order to improve the effectiveness of similarity search.
4. We intensively conduct empirical experiments with real massive datasets to verify our proposed methods, whose results shows how potentially beneficial the methods are when dealing with big data.

The rest of the paper is organized as follows: Sect. 2 presents state-of-the-art which are pointed out how close and different they are when compared to our research work. Section 3 introduces the general concepts related to similarity search and MapReduce as well as some definitions and notations we use in the paper. Next, the proposed clustering scheme, the redundancy-free capability, and other collaborative strategies are given in Sect. 4. Afterwards, several empirical experiments are measured and evaluated in Sect. 5 before our remarks in Sect. 6.

## 2 Related Work

Efficiently doing similarity search and improving performance are of the main objectives in which much work is interested and calls for much attention. Dittrich et al. [7] do research related to Hadoop efficient processing. Their aim is to improve Hadoop performance in many different ways such as partitioning data layouts and building

indices. In order to achieve the goal, they have to, however, change the Hadoop pipelines and get involved in many low-level components inside Hadoop as well as Hadoop distributed file system. Having a different approach but still towards the same objective, we approach performance improvement from the point of view of high-level layers, i.e., algorithms and schemes, such that we build indices and exploit them for candidate search during the MapReduce jobs run time. Besides, Deng et al. [6] present a three-phase MapReduce-based algorithm for string similarity joins in that the first MapReduce operation is for the filter stage and the last two MapReduce operations are for the verification stage. In the verification stage, their algorithm needs, however, to re-access the original datasets whilst our approach only accesses the datasets once from the beginning stage. Rong et al. [19] also introduce a three-phase MapReduce algorithm for string similarity join. Their objective is to reduce the number of candidate string pairs as well. In order to do that, they apply multiple prefix filtering technique, which is based on different global orderings, to their algorithm. Nevertheless, the algorithm behaves in a full-scan manner while our method performs a clustering technique which helps access the right data. Additionally, there is no mention of resolving the redundancy of string pairs as we do in our approach.

Meanwhile, Zadeh and Goel show how to assess MapReduce algorithms. According to their work in [21], the two main complexity measures for MapReduce are the largest bucket reduce because of “the curse of the last reducer” and the shuffle size because of the total file I/O. Thus, it emerges an essential need to reduce candidate sizes as much as possible throughout MapReduce processes. In order to deal with this problem, Kolb et al. [10] focus on how to eliminate redundant similarity comparison between pairs. At REDUCE task, when considering candidate pairs, the reducers only compare those which are disjoint from the list of smaller keys. In contrast to our approach, we do not attach any additional data to intermediate key-value pairs for duplicate-pairs detection at reducers. As an alternative, we keep them identically output in a natural way from mappers and then immediately derive the similarity score between a pair of document. In addition, Metwally and Faloutsos from the work [13] propose a scalable MapReduce-based framework for discovering all-pairs similarity. This method, however, suffers heavy storage and transmission costs due to redundant data in key-value pairs, which is avoided by our method. In another work engaging in diminishing unnecessary data and computations, Phan et al. [15, 16] propose MapReduce-based filtering schemes in an effort of dealing with scalability and improving similarity search with MapReduce. The schemes are shown to generally adapt to the most common similarity search cases such as pairwise similarity, pivot case, range query, and k-Nearest Neighbor query while assuring unqualified candidates are sooner discarded. Meanwhile, Lin in [12] studies three MapReduce algorithms for brute force, large-scale ad-hoc retrieval, and Cartesian product of postings lists. Nevertheless, their concern is typically about scalability aiming at the large amount of data. In the scope of this paper, we not only integrate collaborative strategic refinements to reduce the search space but also address the standard problems of the baseline inverted index-based procedures and data redundancy. Furthermore, our proposed approach easily adapts to different similarity measures thanks to the document indexing-based data structures which interchangeably denote the term as document indexes.

### 3 Preliminaries

#### 3.1 Similarity Search

Consider a universal set  $\Omega = \{D_1, D_2, D_3, \dots, D_n\}$ , which represents a set of  $n$  documents. In the scope of this paper, we employ the concept *k-Shingles* from the work in [18, 20] instead of terms to represent a document, whose idea is that a near duplicate object can be identified by the shingles starting with stop words. Furthermore, *k-shingles* originating from natural language processing are commonly exploited to better represent documents than using terms because of their continuous order while two documents might have the same number of terms but they turn out to appear in different positions which lead to different similarity in terms of meaning. As a consequence, a document from now on is represented by a set of shingles  $D_i = \{SH_1, SH_2, \dots, SH_k\}$ , and the length of a document  $|D_i|$  is known as the total number of shingles belonging to the document.

**Definition 1 (k-Shingles).** Given a document  $D_i$  as a string of characters, *k-shingles* are defined as any sub-string having the length  $k$  found in the document.

**Definition 2 (Similarity Search).** Given a document  $D_i$  and a similarity threshold  $\varepsilon$ , the similarity search looks for all document pairs  $(D_i, D_j)$  in the universal set  $\Omega$ , such that their similarity scores  $SIM(D_i, D_j) \geq \varepsilon$ .

In order to derive the similarity score between a document pair, we utilize the most widely-used similarity measure known as Jaccard coefficient [13, 16, 18, 19, 21] for fast set-based similarity computing. The form of Jaccard is given below:

$$SIM(D_i, D_j) = \frac{D_i \cap D_j}{D_i \cup D_j} \quad (1)$$

The value domain of  $SIM(D_i, D_j)$  is within the range  $[0, 1]$ . If the document  $D_i$  is more similar to the document  $D_j$ , their similarity score is close to 1. Otherwise, their similarity score is close to 0. Last but not least, in the scope of this paper, we use the sign  $[\cdot]$  to demonstrate a list, the sign  $[[\cdot], [\cdot]]$  to specify a list of lists.

#### 3.2 MapReduce Paradigm

Dean and Ghemawat in [5] present MapReduce (MR) as an effective parallel programming paradigm dealing with scalability. The basic idea is to divide a big problem into smaller ones which can be easily done in parallel in a cluster of commodity machines. The main parts of MapReduce constitute a MAP function, which produces intermediate key-value pairs, and a REDUCE function, which deliver results from the key-value pairs. When the MapReduce paradigm is deployed in the cluster, one machine plays the role of master while the others take the responsibility as workers. The master dynamically assigns MapReduce tasks to free workers in the system. Those which are assigned MAP tasks are called mappers whilst those which are assigned REDUCE tasks named as reducers. The principle data flow of a MapReduce phase is

briefly described as follows: (1) Input data whose form is of key-value pairs  $[key_1, value_1]$  from the distributed file system is split into  $m$  Map tasks; (2) Mappers execute MAP function and produce  $r$  local files carrying intermediate key-value pairs  $[key_2, value_2]$ ; (3) The shuffling process is then in charge of grouping these pairs into  $[key_2, [value_2]]$  according to the keys; and (4) Reducers execute REDUCE function to aggregate the key-value pairs and derive the final results which are eventually written back to the distributed file system. To avoid ambiguity, we use the term MapReduce operations when generally mentioning both of them as a whole. Otherwise, they are separately referred as MAP and REDUCE tasks. Additionally, the terms candidate pairs refer to candidate key-value pairs. In other cases, either candidate clusters or candidate document pairs are explicitly pointed out.

## 4 The Proposed Methods

### 4.1 The Clustering Scheme

Due to the fact that MapReduce paradigm itself performs a full-scan fashion to the data, it would be slow to directly work with every single shingle as a unit in order to check whether it matches to the set of query shingles. Even though an inverted index, also known as a postings list, is widely-used in information retrieval and well-employed in lots of research work [6, 8, 10, 12–16, 19, 20] to achieve fast text searches, this method has three main drawbacks, in terms of MapReduce paradigm, for application domains in general and for similarity search in particular. Firstly, when a document is popularly represented by a set of terms or shingles, they are then performed in a full-scan manner from the inverted index. Secondly, the inverted index produces redundant data throughout MapReduce operations, which we will later on give our further analysis in Sect. 4.2. Finally, it is not easy to derive the total length of each document without adding any further information, additional processing, or at least another MapReduce operation. Hence, a research concern related to the former matter emerges such that either “*Is there a way not to sequentially scan every data unit but still get data in need?*” or “*Is it possible to only access the right data from the portion of the whole?*”

To cope with these issues, one possible method comes from clustering techniques. The basic idea is that elemental objects are group into different clusters according to their preferred properties. Thus, a cluster becomes the representative of a group, or in other words, it plays the role of a pivot. Since then, pivots partition the search space into sub-spaces in that they navigate data access to the right objects. In our approach, we cluster shingles into different compartments, which is based on their own documents, so that we can decrease the number of unnecessary data-accessing times. From this point of view, we build the data structure where a document contains a set of its shingles in an incremental manner. That is to say, a document from now on is a cluster of its own shingles and plays the role of a pivot. In comparison with the inverted index, this way of clustering shingles brings three big advantages as follows: (1) To deal with the atomic full-scan from MapReduce paradigm, we model data into a two-layer data access in that the objects we firstly check in regard to a given query are clusters instead of shingles. If the query conditions are met, the shingles of the particular clusters are retrieved for further processing; (2) At the same time, we easily derive the total number

of shingles a document has to carry on in order to support length-based filtering as well as similarity computing afterwards; and (3) This method promotes REDUCE-2 task to be transparent. In other words, it makes REDUCE-2 task get rid out of its burden processing as usual while only play the role of a transmitter writing the final result to the distributed file system.

Figure 1 illustrates two candidate-identifying processes for an inverted index and a document index, respectively in the same dataset. When given a query  $D_q$  and a threshold, let us say, with 70 % similarity, the candidates when we apply the inequality 3 in Sect. 4.3 are those whose number of shingles should greater than 3. Consequently, only  $D_1, D_2, D_4$ , and  $D_5$  satisfy this filtering condition and are then combined with the query to be candidate pairs. In the case of the inverted index, the list of checking objects sequentially includes  $[T, N, D_1, D_7, R, H, D_1, D_4, O, A, D_1, D_4, D_7, G]$ . On the contrary, the process in the case of the document index performs the checking only on keys. Once a key is matched, it becomes a candidate. Hence, the list of checking objects for this case is much shorter and sequentially includes  $[D_1, D_2, D_3, D_4, D_5, D_6, D_7]$ . It is worth noting that a number of shingles in the universe set are usually so many than that of documents. For instance, 4000 Gutenberg files in Fig. 6b have 6358196 shingles in total. Therefore, the document index approach significantly reduces the number of checking objects. Last but not least, its checking process becomes independent of the number of shingles in each document.

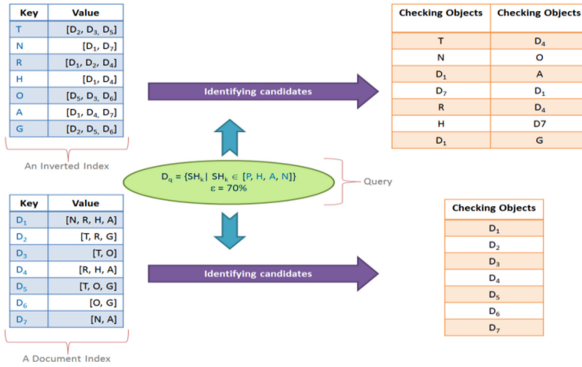


Fig. 1. Candidate-identifying processes

## 4.2 Redundancy-Free Compatibility

When observing data processed in MapReduce operations from the inverted index-based methods, we find out that there are lots of redundant data inner either a mapper or a reducer as well as amongst them. It is totally possible due to the fact that each mapper or reducer only processes a portion of the whole datasets. As a consequence, multiple mappers or reducers may emit duplicate key-value pairs at the same processing phase. On the other hand, because a document contains a set of shingles, or in other words, many different shingles may belong to the same document, each shingle in the inverted index carries the same information. So the research problem here is that “How to avoid redundancy throughout MapReduce operations?” In our research work,

we classify the redundancy into two classes as follows: (1) **Outer redundancy** is the case that there are at least two mappers or reducers emit the same key-value pairs; and (2) **Inner redundancy** is the case that duplicate data are emitted by only either one mapper or one reducer. Figure 2 illustrates how redundant data appear in MapReduce when the inverted index is used to search for candidate pairs. Assume that there are two mappers named  $mapper_a$  and  $mapper_b$ , and one reducer in a MapReduce operation computing candidate similarity pairs. The inverted index, which contains references to documents  $D_i$  for each shingle represented by an upper-case letter, is fed to them as the input in MAP task. When given a query  $D_q$ , the two mappers look up the inverted index the documents sharing the same shingles with  $D_q$ . As a result,  $mapper_a$  finds the candidate pairs as  $[D_{q1}, D_{q4}]$  and  $mapper_b$  has its candidate pairs as  $[D_{q1}, D_{q4}, D_{q7}, D_{q1}, D_{q7}]$ . We see that  $mapper_b$  produce duplicate pairs  $D_{q1}$  and  $D_{q7}$ , which leads to the case of inner redundancy. Meanwhile, both  $mapper_a$  and  $mapper_b$  emit the same candidate pairs  $D_{q1}$  and  $D_{q4}$ , which gives us the case of outer redundancy. Both cases emerge very easily and frequently and add extra costs to data transmission and data computing when one works with MapReduce. Recall that other than MAP task and REDUCE task, the shuffle phase implicitly between them also suffers such a burden in the two cases.

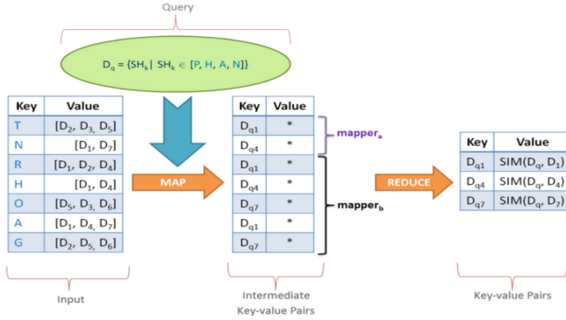


Fig. 2. Data redundancy with the inverted index

Aiming at improving performance, our proposed methods completely avoid the redundancy scenarios when seeking for candidate pairs. To keep away from the case of outer redundancy, one might look for a solution in that once mappers or reducers have emitted the pair  $D_{ij}$ , the other mappers or reducers should not emit the same pair  $D_{ij}$ . Our research work, however, does not need to do that. Actually, our methods look candidate pairs up from the clustering scheme, which is based on clusters instead of shingles themselves. As soon as there is an intersection between a pair, mappers emit it with its similarity score. Since a cluster is unique in the cluster universe, there is no chance for mappers to emit duplicate pairs. Our methods are, therefore, different from the inverted index-based ones in that shingles are not distinctive in the shingle universe due to the fact that two similar documents share the same set of shingles. Meanwhile, the case of inner redundancy impossibly takes place in our methods. The reason is that a document involving in the computing process contains distinct shingles when duplicates are sooner discarded by filtering in Sect. 4.3. In addition, the experimental

result from Fig. 9b in Sect. 5.2 shows that the collision probability of the same document is higher than that of the same shingle. In short, our methods naturally stay away from the redundancy scenarios while without adding any further information when compared to the work in [10]. Moreover, the methods easily adapt to other popular similarity searches such as pairwise similarity, range query, and K-NN query without essentially changing the scheme.

### 4.3 Filtering Strategies

As we know that while I/O operations are very expensive in MapReduce, useless pairs give extra-overheads not only to overall performance but also to data storage. In order to tackle this problem, we actually aim at shrinking the output from mappers. Firstly, we observe that when given a query object, a similarity search process looks for other similar ones based on their signatures, and duplicate signatures do not make sense to the similarity between a pair of objects. Moreover, it is totally redundant if we count duplicate signatures when computing similarity scores. In this paper, we use a set of shingles as the signatures of a document. We discard these duplicate shingles, therefore, from the very beginning of Map tasks, where data are at the first time read by mappers. Once the duplicates are removed, the list of shingles becomes the set of shingles, and the similarity problem turns out to be the overlap set problem [19]. Secondly, when obtaining candidate pairs, it would be useful to refine them in regard to the query object. According to a particular query, range query or K-NN query for example, we utilize the query parameters to sooner prune unnecessary candidate pairs before associating them as true similar pairs and deriving their similarity scores. More concretely, the pruning process is conducted at MAP-2 task. For instance, when given a similarity threshold  $\varepsilon$ ,  $L_i$  is the length of a candidate document, and  $L_q$  is the length of a query document, the candidate pairs are the ones satisfying the below inequality, which is known as length-based filtering from the work in [18]:

$$\frac{L_i}{L_q} \geq \varepsilon \quad (2)$$

In our method, each cluster contains the number of shingles  $NOS$ . The candidate clusters should, therefore, satisfy the below inequality:

$$\|NOS\| \geq \|NOS_{query}\| * \varepsilon \quad (3)$$

On the other side, in the case of K-NN query, we simply exploit the parameter  $k$  to control the emission quantity of each mapper. This can be easily achieved if the keys in the document indexes are ordered by  $NOS$ . Consequently, we can employ this index structure to have key-value pairs in the ascending ordered manner. In other words, we will try to find those pairs having smallest  $NOS$  in order to maximize their similarity scores. Then for K-NN queries, the mappers emit the number of candidate pairs until they reach the top- $k$ .



#### 4.4 Examples on the Fly

Our proposed methods are packaged into two MapReduce phases. The first phase is to ahead of time prepare the data whilst the second one is to on-demand process the queries. Each phase consists of Map and Reduce tasks. In order to get insight of the proposed methods, we introduce a step-by-step example in a nutshell. Assuming that there are two different data sources where the set of documents  $\{Doc_1, Doc_2, Doc_3\}$  belongs to the first one whereas the set of documents  $\{Doc_4, Doc_5, Doc_6\}$  belongs to the other. As showed in Fig. 3, each document owns a set of shingles where a shingle is represented by an upper-case letter. Through MAP-1 task, mappers emit their intermediate key-value pairs of the form  $[URL_i, SH_k]$ . It is worth noticing that common shingles which are very popular or high-frequency shingles across the whole datasets should be, besides duplicate shingles, filtered in this phase. Common shingles can be obtained by datasets statistics or experiences. In addition, pre-defined symbols, blank space between two letters, should also be removed so that clear shingles can be easily acquired. For instance, assuming that the letter “N” is the common shingle, it should be then discarded at mappers. After MAPREDUCE-1 operation, local data are readily prepared in the form of document indexes. When given  $Doc_q$  as a query document, the same MAP-1 task and REDUCE-1 task is executed to analyze the query. These processes are put on display in Fig. 4. At MAP-2 task, only qualified candidate pairs are emitted. On the running example, the query whose  $NOS$  is equal to 5 maintains a list of its shingles  $[K, O, D, E, R]$ .

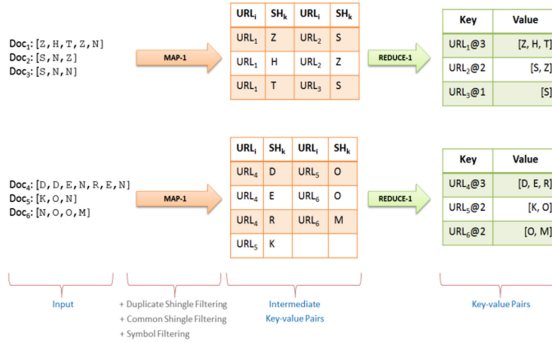


Fig. 3. MAPREDUCE-1 operation

Before finding out the intersection between the query and a document object, the length-based filtering is firstly double-checked against  $NOS$  values. More concretely, assuming that the similarity threshold  $\varepsilon$  is equal to 60 % as in Fig. 5, the candidate pairs are those satisfying the candidate pruning, i.e., the inequality 3 in Sect. 4.3. In other words, the selected pairs have to have their  $NOS$  equal or greater than 3. Consequently, none of candidates in the first local data source is taken because  $URL_2$  and  $URL_3$  do not satisfy the pruning condition whilst  $URL_1$  does not have any shingles in common with the query. At the same time, only  $URL_4$  in the second local data source is chosen for further examining. Even though the shingles of  $URL_5$  and  $URL_6$  in the second local data source are in the set of the query clusters, they are sooner discarded due to the

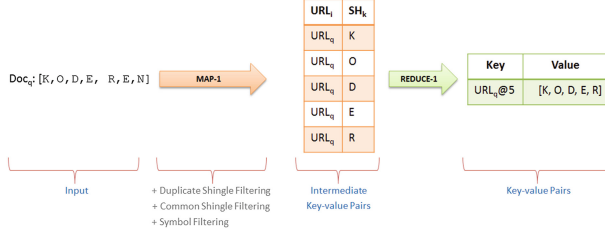
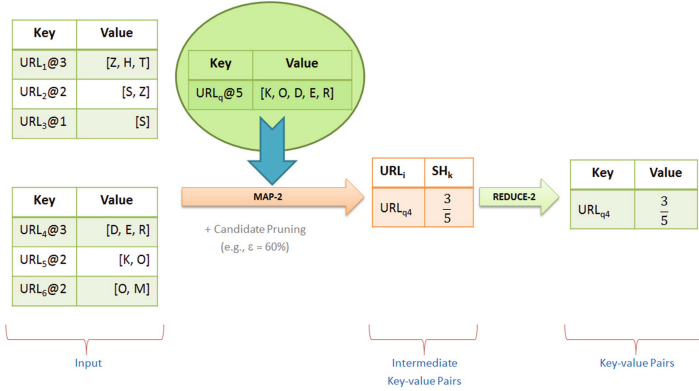
Fig. 4. MAPREDUCE-1 operation with Doc<sub>q</sub>

Fig. 5. MAPREDUCE-2 operation

candidate pruning. Once the candidate pairs are identified, mappers then perform similarity computing. Finally, the reducers from REDUCE-2 task output the final result. For the instance in Fig. 5, we have  $Doc_4$  which is at least 60 % similar to  $Doc_q$  with the similarity score as 3/5. The overview of MapReduce operations and their related information are showed in Table 1. We use a special character, e.g., @, to simply illustrate the separate sub-values.

## 5 Empirical Experiments

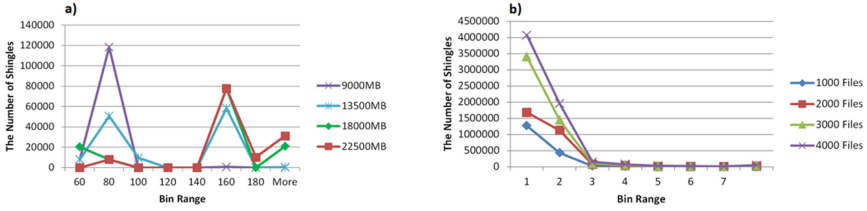
### 5.1 Environment Settings

To setup our experiments, we use DBLP [4] as real datasets where documents containing a number of publications are searched for their similarity. On the other side, we use other real datasets from Gutenberg Project [17], the first provider of free electronic books, to experience a large number of text files.

**With DBLP Datasets.** The datasets are synthetically partitioned into four packages whose sizes are exponentially increased to 9000 MB ( $8 \times DBLP$ ), 13500 MB ( $12 \times DBLP$ ), 18000 MB ( $16 \times DBLP$ ), and 22500 MB ( $20 \times DBLP$ ), respectively. Since recommended in the work [18], the size of a shingle, i.e., the  $K$  parameters for

**Table 1.** The overview of MapReduce operations

MapReduce	Task	Input	Output
	MAP-1	$[D_i]$	$[URL_i, SH_k]$
	REDUCE-1	$[URL_i, SH_k]$	$[URL_i@NOS_i, [SH_k]]$
	MAP-2	$[URL_i@NOS_i, [SH_k]]$	$[D_iD_j, SIM_{ij}]$
	REDUCE-2	$[D_iD_j, SIM_{ij}]$	$[D_iD_j, SIM_{ij}]$
Acronym	<ul style="list-style-type: none"> <li>- <math>D_i, D_j</math>: a document object</li> <li>- <math>SH_k</math>: a k-shingle</li> <li>- <math>URL_i</math>: an uniform resource locator of <math>D_i</math></li> <li>- <math>NOS_i</math>: the total number of shingles of <math>D_i</math></li> <li>- <math>SIM_{ij}</math>: the similarity score between <math>D_i</math> and <math>D_j</math></li> <li>- A special symbol such as "@" is used to separate the values</li> </ul>		

**Fig. 6.** Shingles Histograms; (a) DBLP datasets; (b) Gutenberg datasets

large documents, are chosen as 9 in DBLP datasets and as 4 in Gutenberg datasets. Figure 6a illustrates the number of shingle frequencies among the datasets. It gives a clear vision about the shingle frequency histogram with the bin range representing the interval of shingle frequency in that the majority of shingles falls into the range  $[60, 100]$ ,  $[140, 180]$ , and above the range 180 while most of the shingle frequencies are from the two former ranges.

**With Gutenberg Datasets.** The datasets are divided into four packages separately including *1000 files*, *2000 files*, *3000 files*, and *4000 files*. These files which are randomly selected from the Gutenberg repository have their sizes ranging from 15 KB to 100 KB. In the meantime, Fig. 6b indicates the number of shingle frequencies among the Gutenberg datasets. It gives a clear vision that the majority of shingles falls under the range 4, and most of the shingle frequencies are from the range 1 to 3.

In addition, we employ the stable version 1.2.1 of Hadoop [2] as a fundamental implementation of MapReduce and deploy the Hadoop framework on the cluster of commodity machines named Alex, which has 48 nodes and 8 CPU cores and either 96 or 48 GB RAM for each node [1]. The configured capacity is set to 5 GB per node, which leads to the total 240 GB for the 48-node cluster. Besides, the number of reducers for a reduce task is set to 168. Moreover, the possible heap size of the cluster is about 629 MB, and each HDFS file has 64 MB Block Size. Last but not least, even though some parameters can be tuned or optimized to make the best fit to a particular cluster like Alex, we leave other configurations in their default mode as much as

possible due to the fact that we really want to measure the general performance with such initial settings in that whatever a cluster of commodity machines might initially have. It is worth noticing that the power of Alex is not exclusively employed for our experiments. In other words, these nodes share their computing resources to other coordinating parallel tasks in the cluster. Hence, we conduct an experiment ten times to obtain average values and their corresponding deviations. Additionally, each benchmark meets the fresh-running condition, where old benchmarks are removed before new ones start running. Furthermore, the same types of experiments are consecutively executed in order for them to have the closest running environment as much as possible. Last but not least, the benchmarks are designed to closely fit and reflex the processing capacity of the cluster.

## 5.2 Evaluation

In this section, we conduct our experiments with the real datasets and streaming computation models helping us pass data between MapReduce operations via the standard input and output. Figure 7 presents the performance of MapReduce operations. Figure 7a demonstrates the processing time of MR-1, MR-2, MR-Query, and the total, which turn by turn corresponds to the four DBLP dataset packages. The left vertical axis measures the average processing time of MR-1, MR-2, and MR-Query while the right vertical axis measures the average processing time of all MR-1, MR-2, and MR-Query as the whole. In general, the total costs slightly grow though the dataset sizes are doubled for each test. As we see that the cost of MR-Query is steady throughout the data packages. Besides, the cost of MR-2 does not significantly grow when the dataset size increases from 9000 MB to 22500 MB. The reason comes from the collaborative filtering where it effectively refines the candidate pairs from MR-1 and leaves the rest but small for MR-2. On the contrary, the cost of MR-1 keeps linearly rising when the dataset size keeps increasing. There are two main reasons for this. Firstly, MR-1 has to deal with enormous original data input from the very first stage, which heavily adds the processing cost. And secondly, although some filters are additionally taken, not many shingles are thrown out in comparison with the rest because of the assurance of exact similarity search. Consequently, the majority of shingles are kept for further procedures. In the meantime, Fig. 7b shows the performance of MapReduce operations on Gutenberg benchmarks. The results we get have the same trend like that on DBLP datasets. The costs of MR-Query and MR-2 seem to be stable and not much affected by the large number of files. On the contrary, the cost of MR-1 is linearly high when the number of files is increased from 1000 to 4000. It is worth noting that the cost for reducers is usually higher than that for mappers because mappers take their responsibility to tokenize the data while reducers pull intermediate key-value pairs, process them to achieve the goal, and write them into the distributed file system. Moreover, the number of mappers is usually driven by the number of distributed file system blocks in the input files whilst the number of reducers is chosen by either experiences or evaluations to a particular cluster of commodity machines. As a consequence, if the number of reducers is not suitably set, it affects the total cost in

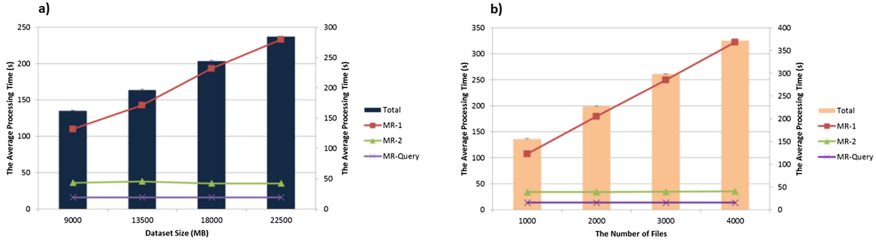


Fig. 7. Performance; (a) DBLP datasets; (b) Gutenberg datasets

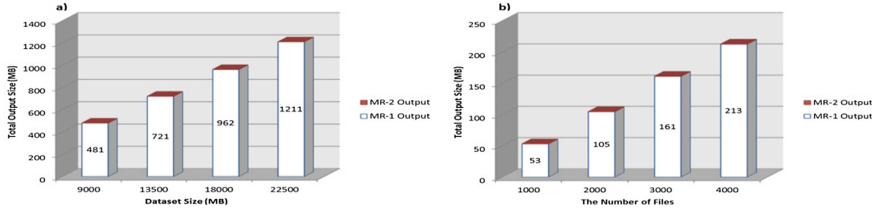


Fig. 8. Data output; (a) DBLP datasets; (b) Gutenberg datasets

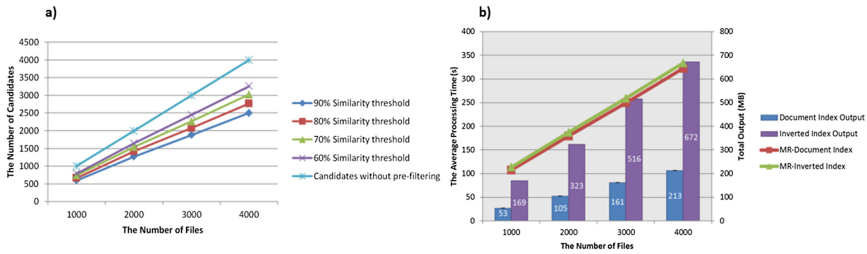


Fig. 9. Measurement; (a) Range query case; (b) The relevance between a document index and an inverted index

the end. We put, therefore, main-point processing on mappers at MAP-2 task and at the same time make reducers at REDUCE-2 task be transparent when doing similarity search, which brings an advantage to the overall performance.

On the other hand, Fig. 8 shows how much data saved from the computing processes. In an overall, the total output of MR-1 and MR-2 on DBLP datasets is much less than the input size, which accounts for 5.35 % rate of the input on the average. The total output of MR-1 and MR-2, nevertheless, accounts for 79.22 % rate of the input on the average. When the next data package in DBLP datasets is doubled, MR1-Output of this package is as nearly 1.36 times larger on the average as that of the previous package. Because of preserving as much data as possible from the datasets, the size of MR1-Output is non-trivial while that of MR2-Output is negligible, for it is around 62 KB to 161 KB. On the other hand, when the number of files is increased in Gutenberg datasets, MR1-Output of this package is as nearly 1.97 times larger on the average as that of the previous package while MR2-Output keeps its small size around

67 KB to 272 KB. Thus, the size of the entire output is totally decided by that of MR1-Output. Again, the power of filtering is completely verified at MAP-2 task. It is worth noticing that if the first MapReduce phase can be alternatively put in offline mode, the cost of the second phase is, therefore, promising in online mode. Meanwhile, Fig. 9a shows the range query case where the inequality 3 in Sect. 4.1 is applied on Gutenberg datasets. In overview, the number of candidates without filtering approximately equals to the number of input files while the number of filtered candidates is more and more when the similarity threshold increases from 60 % to 90 %. More specifically, about 37.83 % on the average unnecessary candidates are discarded in case of 90 % similarity, about 31.16 % of that number are ignored in case of 80 % similarity, about 24.39 % of that number are removed in case of 70 % similarity, and about 19.01 % of that number are filtered in case of 60 % similarity. Another experiment whose results are displayed on Fig. 9b indicates the relevance between the document index approach and the inverted index approach. Normally, the average processing time is not much different between them. The total MapReduce outputs between the two approaches significantly have, however, a big gap. The experimental result shows that building the inverted index produces approximately 3 times as many MapReduce outputs as building the document index. Hence, the document index saves more data for further processing than the inverted index.

## 6 Summary

In this paper, we propose a novel MapReduce-based approach for efficient similarity search. Apart from dealing with scalability, we also consider the drawbacks of the inverted index in terms of similarity search. In addition, we promote a simple yet efficient redundancy-free MapReduce scheme, which shows its advantages when compared to inverted index-based procedures. Furthermore, we present strategic methods to cope with unnecessary data and computations. Last but not least, the results from intensive empirical evaluations with massive real datasets promote the efficiency of our methods. For our future work, we identify a distributed MapReduce-based architecture to which our approach conforms in order to cope with the “three Vs” of big data. Additionally, we further evaluate our proposed methods with other state-of-the-arts as well as more empirical experiments in other popular cases of similarity search and similarity measures.

**Acknowledgements.** Our sincere thanks to Faruk Kujundžić, Scientific Computing, Information Management team, Johannes Kepler University Linz, for his kind support in the Alex Cluster.

## References

1. Alex cluster. Available on the following website link. <http://www.jku.at/content/e213/e174/e167/e186534>. Accessed 4 Feb 2014
2. Apache Hadoop. Wiki at <http://hadoop.apache.org/docs/r1.2.1/>. Accessed 8 Mar 2014

3. Bayardo, R., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: Proceedings of the 16th International Conference on World Wide Web, pp. 131–140 (2007)
4. DBLP data set. <http://dblp.uni-trier.de/xml/>. Accessed 8 Mar 2014
5. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation, USENIX Association, pp. 137–150 (2004)
6. Deng, D., Li, G., Hao, S., Wang, J., Feng J.: MassJoin: a MapReduce-based algorithm for string similarity joins. In: Proceedings of the 30th IEEE International Conference on Data Engineering, pp. 340–351 (2014)
7. Dittrich, J., Richter, S., Schuh, S.: Efficient or Hadoop: why not both? *Datenbank-Spektrum* **13**(1), 17–22 (2013)
8. Elsayed, T., Lin, J., Oard, D.W.: Pairwise document similarity in large collections with MapReduce. In: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies, Companion Volume, pp. 265–268 (2008)
9. Han, J., Kamber, M., Pei, J.: Data mining: concepts and techniques, 3rd edn. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers. ISBN: 978-0123814791 (2011)
10. Kolb, L., Thor, A., Rahm, E.: Don't match twice: redundancy-free similarity computation with MapReduce. In: Proceedings of the 2nd International Workshop on Data Analytics in the Cloud (2013)
11. Letouzé, E.: Big data for development: challenges & opportunities. In: Tatevossian, A.R., Kirkpatrick, R., (eds.) UN Global Pulse, pp. 1–47 (2012)
12. Lin, J.: Brute force and indexed approaches to pairwise document similarity comparisons with MapReduce. In: Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 155–162 (2009)
13. Metwally, A., Faloutsos, C.: V-SMART-join: a scalable MapReduce framework for all-pair similarity joins of multisets and vectors. *PVLDB* **5**(8), 704–715 (2012)
14. Mika, P.: Distributed indexing for semantic search. In: Proceedings of the 3rd International Semantic Search Workshop, pp. 1–4 (2010)
15. Phan, T.N., Küng, J., Dang, T.K.: An efficient similarity search in large data collections with MapReduce. In: Dang, T.K., Wagner, R., Neuhold, E., Takizawa, M., Küng, J., Thoai, N. (eds.) FDSE 2014. LNCS, vol. 8860, pp. 44–57. Springer, Heidelberg (2014)
16. Phan, T.N., Küng, J., Dang, T.K.: An elastic approximate similarity search in very large datasets with MapReduce. In: Hameurlain, A., Dang, T.K., Morvan, F. (eds.) Globe 2014. LNCS, vol. 8648, pp. 49–60. Springer, Heidelberg (2014)
17. Project Gutenberg. <http://www.gutenberg.org/>. Accessed 8 Mar 2014
18. Rajaraman, A., Ullman J.D.: Finding similar items. In: Mining of Massive Datasets, 1st edn, pp. 71–127 (Chap. 3). Cambridge University Press, Cambridge (2011)
19. Rong, C., Lu, W., Wang, X., Du, X., Chen, Y., Tung, A.K.H.: Efficient and scalable processing of string similarity join. *IEEE TKDE* **25**(10), 2217–2230 (2013)
20. Theobald, M., Siddharth, J., Paepcke, A.: Spotsigs: robust and efficient near duplicate detection in large web collections. In: Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 563–570 (2008)

21. Zadeh, R.B., Goel, A.: Dimension independent similarity computation. *J. Mach. Learn. Res.* **14**(1), 1605–1626 (2013)
22. Zikopoulos, P.C., Eaton, C., DeRoos, D., Deutsch, T., Lapis, G.: *Understanding big data: analytics for enterprise class Hadoop and streaming data*. McGraw-Hill Osborne Media, New York. ISBN: 978-0071790536 (2012)



Future Data and Security Engineering  
Second International Conference, FDSE 2015, Ho Chi  
Minh City, Vietnam, November 23-25, 2015,  
Proceedings  
Dang, T.K.; Wagner, R.; Küng, J.; Thoai, N.; Takizawa, M.;  
Neuhold, E. (Eds.)  
2015, XIII, 323 p. 99 illus. in color., Softcover  
ISBN: 978-3-319-26134-8