Cluster Comput (2016) 19:109–126
DOI 10.1007/s10586-015-0506-0

Efficient top-*k* similarity document search utilizing distributed file systems and cosine similarity

Mahmoud Alewiwi[1] · **Cengiz Orencik**[1] · **Erkay Savaş**[1]

Received: 19 February 2015 / Revised: 29 September 2015 / Accepted: 29 October 2015 / Published online: 9 November 2015

Abstract Document similarity has important real life applications such as finding duplicate web sites and identifying plagiarism. While the basic techniques such as *k*-similarity algorithms have been long known, overwhelming amount of data, being collected such as in big data setting, calls for novel algorithms to find highly similar documents in reasonably short amount of time. In particular, pairwise comparison of documents' features, a key operation in calculating document similarity, necessitates prohibitively high storage and computation power. In this paper, we propose a new filtering technique that decreases the number of comparisons between the query set and the search set to find highly similar documents. The proposed filtering technique utilizes *Z*-order prefix, based on the cosine similarity measure, in which only the most important features are used first to find highly similar documents. We propose a three-phase approach, where the phases are *near duplicate detection*, *common important terms* and *join phase*. We utilize the Hadoop distributed file system and the MapReduce parallel programming model to scale our techniques to big data setting. Our experimental results on real data show that the proposed method performs better than the previous work in the literature in terms of the number of joins, and therefore, speed.

Keywords Z-order · Document similarity · MapReduce · Hadoop · Cosine similarity

✉ Cengiz Orencik
cengizo@sabanciuniv.edu

Mahmoud Alewiwi
alewiwi@sabanciuniv.edu

Erkay Savaş
erkays@sabanciuniv.edu

[1] Faculty of Science and Engineering, Sabanci University, Istanbul, Turkey

1 Introduction

Big data, referring to not only the huge amount of data being collected, but also associated opportunities, has big potential for major improvements in many fields from health care to business management. Therefore, there is an ever-increasing demand for efficient and scalable tools that can analyze and process immense amount of, possibly unstructured, data, which keeps increasing in size and complexity. Finding similarities (or duplicates) among multiple data items, or documents, is one of the fundamental operations, which can be extremely challenging due to the nature of big data. In particular, similarity search on a huge data set, where the documents are represented as multi-dimensional feature vectors, necessitates pair-wise comparisons, which requires the computation of a distance metric, and therefore can be very time and resource consuming, if not infeasible.

A commonly used technique, known as filtering, decreases the number of pairwise comparisons by skipping the comparison of two documents if they are not potentially similar; e.g., they do not share any common feature. Also, representation, storage, management and processing of documents play an important role in the performance of similarity search method. A distributed file system and a parallel programming model such as MapReduce [9] are necessary components of a scalable and efficient solution in big data applications.

This work focuses on the general problem of detecting the *k*-most similar documents for each document within a given data set (henceforth *self join*), and between two arbitrary sets of documents (R–S join), namely query set and data set. The problem is formalized as follows.

Definition 1 (*R–S join top-k set similarity*) Let $\mathcal{D}$ be a set of documents $\{d_1, \ldots, d_n\}, d_i \in \mathcal{D}$. Let $\mathcal{Q}$ be a set of query documents $\{q_1, \ldots, q_m\}, q_j \in \mathcal{Q}$. Then R–S top-*k* set similarity

is defined as:

$$\forall q_j \in \mathcal{Q}, top\text{-}k(q_j, \mathcal{D}) = \{d_{j1}, \ldots, d_{jk}\},$$

where d_{ji} is the ith nearest record to q_j in $\mathcal{D}$.

Definition 2 (*Self join top-k set similarity*) Let $\mathcal{D}$ be a set of documents $\{d_1, \ldots, d_n\}$, $d_i \in \mathcal{D}$. Then self join top-k set similarity is defined as:

$$\forall d_j \in \mathcal{D}, top\text{-}k(d_j, \mathcal{D}) = \{d_{j1}, \ldots, d_{jk}\},$$

where $d_{ji} \neq d_j$ is the ith nearest record to d_j in $\mathcal{D}$.

Intuitively, the self join case is the generalization of the R–S join case such that $\mathcal{Q} = \mathcal{D}$.

The trivial solution for the set similarity problem, for two sets of items $\mathcal{Q}$ and $\mathcal{D}$, is to compare each element in $\mathcal{Q}$ with each element in $\mathcal{D}$. This solution has $\mathcal{O}(\delta mn)$ complexity, where m and n are the number of elements in $\mathcal{Q}$ and $\mathcal{D}$ respectively, and δ is the number of dimensions (i.e., features) in $\mathcal{Q}$ and $\mathcal{D}$. Recent trends and research are concerned about computing set similarity-join algorithms in an efficient and high performance manner, hence new set similarity-join algorithms that reduce the number of comparisons are proposed.

Current research mainly adapts *filtering techniques* that filter out pairs that have similarity below a given threshold. Clearly, the adopted filtering technique plays the utmost role in the efficiency as well as the effectiveness of a similarity search algorithm. In this work, we propose a new cosine similarity based filtering technique to improve the performance of the similarity calculation.

In our solution, we suggest a new Z-order based filtering technique in order to eliminate dissimilar documents before performing the costly operation of calculating the cosine similarity. Documents in a data set are represented as points on Z-order space filling curves on multidimensional space of documents' features. A projection based on the most important features of documents is utilized to filter out dissimilar documents that do not share common important features. The proposed method also filters out documents that only share features of low importance, which have very little effect on the similarity between the data objects. The technique provides a remarkable improvement, especially on finding highly similar documents very quickly. And finally, we describe the algorithms to implement the proposed technique as parallel programs that can take advantage of the Hadoop [2] MapReduce parallel programming framework.

The rest of the paper is organized as follows. In Sect. 2, an overview of the related methods in the literature is provided. The Z-order based filtering method and the MapReduce framework are introduced in Sect. 3. The details of the proposed Z-order with lambda iteration prefix (ZOLIP) filtering algorithm are explained in Sect. 4. Section 5 presents the experimental results that show the accuracy and the efficiency of the proposed method. Finally, the conclusions are provided in Sect. 6.

2 Related work

In the literature, the problem of *set-similarity* on a single machine is considered in several works [3,5,7,19]. These works are mainly focused on reducing the complexity of *vector similarity join*. Angiulli et al. [1] used the Z-order space filling curve in order to find the similarity between two high dimensional spatial data sets using Minkowski metrics. This method performs well for finding close pairs in high dimensional data, but it is not suitable for text based similarity detection. For text based similarity, as in the case of document similarity problem, the cosine similarity metric is more suitable than the Minkowski metric.

Connor and Kumar [8] suggested another technique for the similar document detection problem. They used a binary search technique to find the k-NN within a selected Z hypercube. A popular approach in other works is adapting *filtering techniques* that filter out pairs that cannot surpass a given similarity threshold. This technique decreases the number of candidates and therefore the number of similarity join operations by eliminating the documents that do not share a common important term with the query.

There are various filtering techniques used in the literature. A prefix filtering method is suggested by Chaudhuri et al. [7]. The length filtering method is utilized in the works [3] and [5]. Positional and suffix filters are proposed by Xiao et al. [22]. Sarawagi and Kirpal [19] proposed a method called *PPJoin+* that utilizes inverted index and uses a *Pair-Count* algorithm which generates pairs that share certain number of tokens. Arasu et al. [3] proposed a signature based method, in which the features of documents are represented by signatures and the similarity among the documents is calculated using the similarity of the underlying signatures. Zhu et al. [25] suggested a searching technique based on cosine similarity. They proposed an algorithm that utilizes a diagonal traversal strategy to filter out unrelated documents. In this algorithm, the elements in the data set are represented by binary vectors, meaning that only the existence of terms is considered, ignoring their frequencies or importance in the data set.

The MapReduce computing model is also considered for the similarity search problem and this leads to parallel join algorithms for large data sets that are stored on cloud servers. Elsayed et al. [10] suggested a MapReduce Model with a *Full-Filtering* technique. They used a simple filter that finds only the pairs that share common tokens. The proposed method is composed of two phases. While the first phase parses and creates the indexes for the terms in each document,

the second phase finds the similar pairs that share these terms. Vernica et al. [21] used the *PPJoin+* method [19] in order to perform the *self-join* operation. Yang et al. [23] proposed a method that uses prefix and suffix filters with two phases of MapReduce. Inverted index is used in [14] combined with prefix filtering. A modified double pass MapReduce prefix filtering method was proposed by Baraglia et al. [4]. Phan et al. [17] used Bloom filtering for building similarity pairs, in which each pair should intersect at least in one position with the arrays generated by the Bloom filters.

The previous works in the literature of similarity search did not take the importance of the terms into consideration. This affects the semantic similarity between documents (i.e., some documents may have the same terms but in different contexts). In our algorithm, in order to address this issue, we utilize a cosine similarity based filtering technique using the relative importance of terms in documents for finding similar documents.

3 Preliminaries

This section provides the main principles used in the proposed ZOLIP filtering algorithm. First, we give the relevancy scoring method used for calculating the similarity of documents. Next, the principles of the Z-order mapping are described. In the last part, we explain the MapReduce framework, which is used to efficiently implement the proposed method. Table 1 provides a list of the notations used throughout the paper for easy reference.

Table 1 Notations used in the paper

Notation	Meaning
$\mathcal{D}$	Data set
$\mathcal{Q}$	Query set
d_i	ith document in the data set
q_i	ith query document in the query set
d_{i_j}	jth term in the ith document
$d_{i_j}^l$	Most significant l-bits of the term d_{i_j} in document d_i
$\bar{d}_i^l$	Projection of the vector d_i to lth Iteration on the Z-order curve
T	Set of all terms in the data set
δ	Number of terms in the dictionary, i.e., $\|T\|$
t_i	ith term in the dictionary T
tf-idf	Term frequency-inverse document frequency
μ	Maximum tf-idf value in the data set
$\mathcal{S}_c(d_i, d_j)$	Cosine similarity between d_i and d_j
σ	Minimum similarity threshold
λ	Number of iterations on the Z-order curve

3.1 Term relevancy score

Any data object (e.g., a document, an image, a video file, etc.) can be represented as a vector of features, which identifies that data object. In this work, we represent documents by a set of terms (i.e., keywords). More formally, each document d_i in the data set $\mathcal{D}$ contains a certain number of terms from a global dictionary T, where $|T| = \delta$ is the total number of terms in the dictionary. More specifically, each document in the data set is represented as a vector of *term weights*, where a term weight captures the importance of the corresponding term in identifying a document. In our scheme, a component of the term vector for the document d_i is in fact the relevance score of the corresponding term t_j, which simply indicates the importance of the term t_j in distinguishing d_i from all the other documents in $\mathcal{D}$.

One of the most commonly used weighting factor in information retrieval is the tf-idf value of a term in a document [16]. This factor quantifies the importance of a term in a document and combines two metrics: (i) the term frequency (tf) which is the number of occurrences of the term in a document (i.e., $tf_{j,i}$ is the number of occurrence of the term t_j in the document d_i) and (ii) the inverse document frequency (idf), which represents the fraction of the documents that contain the term t_j among the whole document set. In other words, idf is a measure of the rarity of the term t_j in document set $\mathcal{D}$. The tf-idf of a term t_j in the document d_i is calculated as

$$\textit{tf-idf}_{j,i} = tf_{j,i} \times idf_j.$$

In practice, since a given term usually occurs only in a limited number of documents, the tf-idf vectors contain many zero elements and thus, tf-idf values are stored in a sparse vector to optimize the memory usage.

Let $\mathcal{S}(d_i, d_j)$ be the similarity function that quantifies the similarity between two documents, d_i and d_j. Let σ be the threshold of minimum required similarity for the pair d_i and d_j to be considered as similar. The similarity join problem is to find the candidate d_j for the document d_i such that $\mathcal{S}(d_i, d_j) \geq \sigma$. There are different choices for suitable similarity functions depending on the application domain. The most commonly used similarity metrics in the literature for objects d_i and d_j are described as follows

- *Jaccard similarity* $\mathcal{S}_j(d_i, d_j) = \dfrac{|d_i \cap d_j|}{|d_i \cup d_j|}$,
- *Cosine similarity* $\mathcal{S}_c(d_i, d_j) = \dfrac{d_i \cdot d_j}{||d_i|| \cdot ||d_j||}$,
- *Hamming distance* $\mathcal{S}_h(d_i, d_j) = |(d_i - d_j) \cup (d_j - d_i)|$,

Generally speaking, two documents may contain many common terms; but this does not necessarily imply that they are sufficiently similar as the importance of terms can have a decisive impact on the similarity. Therefore, the Jacard similarity metric focused on the existence of a term in a document, ignoring its relative importance (e.g., its tf-idf score), is not the best choice for document similarity problem, where similarity relations are more complex. As tf-idf scores can easily be used in similarity calculations, the cosine similarity is a better option in this context. A drawback of the cosine similarity is that it is computationally more expensive than the other metrics. Thus, it is necessary to avoid its computation for documents that are not potentially similar.

In our approach, we adopt the cosine similarity metric to capture more complex similarity relations in document data sets and a filtering method to avoid cosine similarity computation for document pairs that have similarities lower than the required threshold σ. The terminology used in this paper is that a *join* operation is the computation of cosine similarty between the document pairs if they are potentially similar. The simple idea, therefore, is to decrease the number of join operations considering only the important terms (i.e., those that have high tf-idf values) that affect the similarity more than the other terms. More precisely, the cosine similarity can be calculated as

$$\mathcal{S}_c(d_i, d_j) = \frac{d_i \cdot d_j}{||d_i|| \cdot ||d_j||} = \frac{\sum_{t=1}^{\delta} d_{it} \times d_{jt}}{\sqrt{\sum_{t=1}^{\delta} d_{it}^2} \times \sqrt{\sum_{t=1}^{\delta} d_{jt}^2}},$$

where d_{it} and d_{jt} are the weights of the corresponding terms in the documents d_i and d_j. Without loss of generality, we can assume that d_i and d_j contain the same number of terms. In case there are different number of terms in documents, we can always pad the term vector with terms whose tf-idf values are 0. From the above formula, one can understand that the terms with higher tf-idf values contribute to the cosine similarity metric more significantly than the terms with relatively smaller tf-idf values. This observation is the core of our filtering technique. Example 1 demonstrates the calculation of the similarity of documents using only the important terms.

Example 1 As a small example, let $\delta = 10$ and a, b and c be documents represented with tf-idf vectors as follows,

$$a = (0, 8, 5, 0.25, 0.125, 0, 0.25, 0, 0, 0.75)$$
$$b = (0.5, 9, 4, 0, 0, 0.125, 0, 0, 0, 0)$$
$$c = (9, 0.25, 7, 0, 0.75, 1, 0.5, 1, 0, 7).$$

Further let $\bar{a}, \bar{b}, \bar{c}$ be the projected vectors that use only the terms with high tf-idf values, ignoring any value less than 1. Then we obtain the following projected term vectors for the objects a, b, and c, respectively

$$\bar{a} = (0, 8, 5, 0, 0, 0, 0, 0, 0, 0)$$
$$\bar{b} = (0, 9, 4, 0, 0, 0, 0, 0, 0, 0)$$
$$\bar{c} = (9, 0, 7, 0, 0, 1, 0, 1, 0, 7).$$

Notice that, if we calculate the cosine similarity between pairs of the tf-idf vectors, we see that

$$\mathcal{S}_c(\bar{a}, \bar{b}) = 0.9901 \approx \mathcal{S}_c(a, b) = 0.9849, \quad (1)$$
$$\mathcal{S}_c(\bar{a}, \bar{c}) = 0.2758 \approx \mathcal{S}_c(a, c) = 0.3325. \quad (2)$$

Therefore, even though the pair (a, c) has more common terms, we can conclude that the closest pair is (a, b), since the terms with small tf-idf values have a very low effect on the cosine similarity.

In the next section, we formalize the proposed method used for extracting the terms that have an actual effect on the cosine similarity, namely the most important terms in the document.

3.2 *Z*-order mapping

Z-order or Morton order is a space filling curve, whose different iterations can be computed as shown in Fig. 1. As the number of iterations increases, the space can be filled with higher accuracy. One important property of Z-order curve is that, it preserves the locality of data points in the space. Therefore, Z-order is a frequently used approach for mapping multidimensional data into one dimensional space while preserving the support for operations such as comparison and similarity check after the mapping. Here, we formalize our terminology for the Z-order curves.

Definition 3 (*Iteration on Z-order curve*) The lth iteration of the Z-order curve in δ-dimensional space is a set of $2^{\delta l}$ sub-curves, where each sub-curve is composed of points whose coordinates have the same l most significant bits.

Figure 1 illustrates the Z-order sub-curves for different iterations on the original curve.

Definition 4 (*Z-shape of order l*) A Z-shape of order l in δ-dimensional space is any of the Z-order sub-curves in an lth iteration of the Z-order curve.

Intuitively, each sub-curve in an iteration on a Z-order curve is a Z-shape. For instance, the circles labeled as A and B in Fig. 2 enclose the second and first order Z-shapes, respectively.

Definition 5 (*Z-value*) The Z-value of a data point in the multidimensional space is obtained by interleaving the

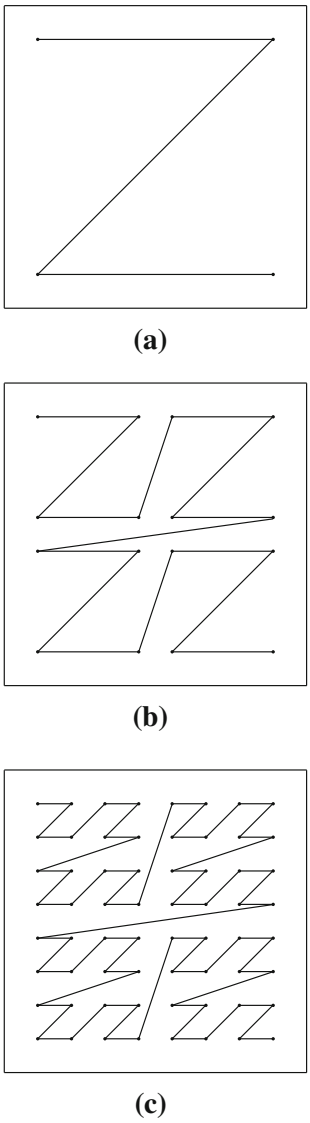

Fig. 1 Z-order space filling. **a** Zeroth iteration on Z-order. **b** First iteration on Z-order. **c** Second iteration on Z-order

bits of binary representation of the data point coordinate values.

Points in the δ-dimensional space are represented with scalars (i.e., Z-value), which preserve their locality in such a way that similarity computation and comparison between points can be performed. For instance, the point $(110, 001)$ in the Z-shape of order 2 in Fig. 2 is mapped to Z-value of 101001. In summary, the Z-value of a point in multidimensional space is simply a scalar that can be used in various applications. In the literature [8,20,24], the Z-value is used to find the k-nearest neighbors in spatial data sets.

A document represented by a vector of tf-idf values can be viewed as a point in the multidimensional space of terms. Then, the Z-order mapping can be used to map a document into one dimensional space by preserving the locality of the

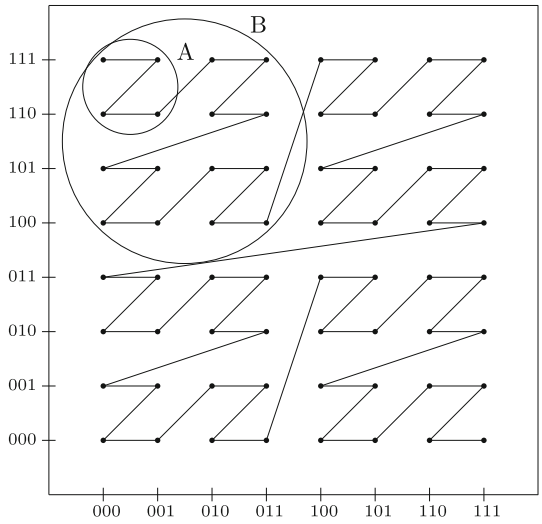

Fig. 2 Data points on Z-order curve

document in the multidimensional space. Consequently, it will be possible to compute the similarity of documents using their Z-values.

For instance, in the two-dimensional space in Fig. 2 (i.e., $\delta = 2$), the circle A denotes a Z-shape of the second order while the one denoted by B is a Z-shape of the first order. The points in the Z-shape A are $(000,110)$, $(000,111)$, $(001, 110)$, and $(001, 111)$. Their corresponding Z-values, which are (010100), (010101), (010110), and (010111), have the same prefix of (0101). Similarly, the Z-values of the points in B share the prefix (01). From this, we can conclude that the points on a Z-shape of larger order (i.e., which share a longer prefix) are closer. The common prefix in Z-values of the points can be used to calculate the similarity of documents (i.e., closeness in the multidimensional space), where the coordinates of the points are the tf-idf values of the corresponding terms in the documents.

Next example demonstrates a technique that uses the Z-order mapping to obtain the most important terms in a document. Let $\lambda \cdot \delta$ be the number of prefix bits shared in the same Z-shape, where δ is the number of terms in the dictionary T (i.e., the dimension of the document space) and λ is an accuracy parameter chosen appropriately.

Example 2 Recall that in Example 1, we have the following term vectors for three documents, where $\delta = 10$

$$a = (0, 8, 5, 0.25, 0.125, 0, 0.25, 0, 0, 0.75)$$
$$b = (0.5, 9, 4, 0, 0, 0.125, 0, 0, 0, 0)$$
$$c = (9, 0.25, 7, 0, 0.75, 1, 0.5, 1, 0, 7).$$

Initially, each coordinate (i.e., the elements in the tf-idf vectors) is represented by the number of bits that is necessary to represent all tf-idf scores in the data set. Here, the largest tf-idf score is 9, which requires 4 bits to represent the integer

part. Also the fractional parts of the tf-idf scores require at most 3 bits. Thus, a total of 7 bits are needed to accurately represent all tf-idf values in this example. For example, tf-idf values of 9 and 0.75 are represented as 1001.000 and 0000.110, respectively. However, by setting $\lambda = 3$ at the expense of losing precision, we use only the three most significant bits in the binary representation (i.e., the prefix values) of the term vectors as shown in the following table. For example, tf-idf values of 9 and 0.75 are now represented by bit strings of 100 and 000, respectively.

Z-order iteration prefix	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
Doc a										
1^{st}	0	1	0	0	0	0	0	0	0	0
2^{nd}	0	0	1	0	0	0	0	0	0	0
3^{rd}	0	0	0	0	0	0	0	0	0	0
Doc b										
1^{st}	0	1	0	0	0	0	0	0	0	0
2^{nd}	0	0	1	0	0	0	0	0	0	0
3^{rd}	0	0	0	0	0	0	0	0	0	0
Doc c										
1^{st}	1	0	0	0	0	0	0	0	0	0
2^{nd}	0	0	1	0	0	0	0	0	0	1
3^{rd}	0	0	1	0	0	0	0	0	0	1

Notice that documents a and b have the same prefix for all three iterations. This is expected since a and b are very similar as shown in Example 1.

In this work, we first develop an efficient method to find similar documents such that, cosine similarity between two documents is computed only if they are in the same Z-shape of the λth order; i.e., having the same $\lambda\delta$ bits as prefix in their Z values. On the other hand, this method, which eliminates the need of calculating the cosine similarity between many dissimilar document pairs and therefore yields a very efficient implementation, can only be applicable in cases where highly similar documents exist. If the data set does not contain sufficiently many (i.e., k) highly similar documents, it is required to also consider the documents that do not reside in the same Z-shape of λ order.

For similar documents that do not reside in the same Z-shape of the desired order, we propose a slightly different method, in which only documents that contain common important terms (i.e., that have high tf-idf values) will be compared. In other words, if two documents contain at least one important term in common, their cosine similarity is calculated, otherwise the computation is skipped.

Definition 6 (*lth iteration projection*) Let $d_i = (d_{i_1}, \ldots, d_{i_\delta})$ be a document represented in δ-dimensional space of term tf-idf values. Let also $d^l_{i_j}$ denote the most significant l-bits of the values d_{i_j} for $j = 1, \ldots, \delta$. Then we can define the projection of the vector d_i to lth iteration on the Z-order curve as

$$\bar{d}^l_{i_j} = \begin{cases} d_{i_j}, & \text{if } \; d^l_{i_j} > 0 \\ 0, & \text{otherwise,} \end{cases}$$

for $j \in \{1, \ldots, \delta\}$.

The projection takes an already sparse vector d_i and generates a possibly much sparser vector, $\bar{d}_i$, that contains only the tf-idf values of important terms. Then, $\bar{d}_i$ is checked whether it represents the document with a sufficiently high accuracy.

In the proposed method, we start with $1st$ iteration projection $\bar{d}_i{}^1$ of a document, for which we try to find similar documents, and compute the similarity, $\mathcal{S}_c(d_i, \bar{d}^1_i)$. If the similarity is larger than a predefined similarity threshold σ, namely $\mathcal{S}_c(d_i, \bar{d}^1_i) > \sigma$, then we use $1st$ iteration projections of the two documents to decide to compute their cosine similarities.

If $\mathcal{S}_c(d_i, \bar{d}^1_i) < \sigma$, then we use a higher level projection $\bar{d}_i{}^l$ with $l > 1$, where l is the minimum value that satisfies the threshold σ. Then, we compute $\mathcal{S}_c(d_i, d_j)$ of the two documents d_i and d_j only if $\bar{d}^l_i$ and $\bar{d}^l_j$ have at least one common non-zero term. Examples 3 and 4 describe the projection process of document vectors on different Z-order dimensions.

The aim of the projection process is to decrease the dimensions required for traversing the space filling curve and consequently decrease the number of similar document candidates.

Example 3 Let a, b, c, d and e be documents represented with the following tf-idf vectors and there be only 3 dimensions (i.e., $\delta = 3$) namely, x, y and z, for easy visualization purposes. Further let $\sigma = 0.8$ (minimum similarity threshold), and maximum number of iterations $\lambda = 3$.

$$a = (16, 14, 2)$$
$$b = (2, 1, 10)$$
$$c = (16, 16, 10)$$
$$d = (2, 7, 10)$$
$$e = (4, 4, 1).$$

For document a, if we calculate the first and second iteration projections, we obtain the following vectors

$$\bar{a}^1 = (16, 0, 0)$$
$$\bar{a}^2 = (16, 14, 0).$$

In order to achieve the targeted similarity threshold, $\sigma = 0.8$, we calculate the similarity between the original document a and the documents projected on these dimensions, namely $\bar{a}^1$ and $\bar{a}^2$. Using cosine similarity, we obtain the following results:

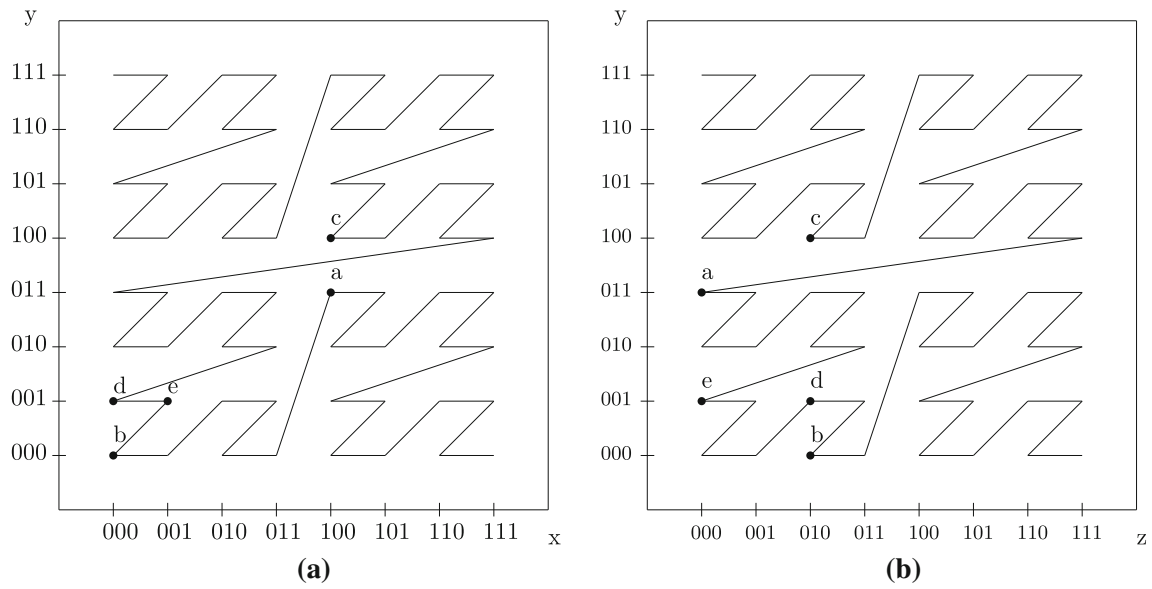

Fig. 3 Projected Z-order space filling on (x, y) and (y, z) dimensions

$$\mathcal{S}_c(a, \bar{a}^1) = \frac{a_i \cdot \bar{a}^1_j}{||a_i|| \cdot ||\bar{a}^1_j||} = 0.75,$$

$$\mathcal{S}_c(a, \bar{a}^2) = \frac{a_i \cdot \bar{a}^2_j}{||a_i|| \cdot ||\bar{a}^2_j||} = 0.99.$$

As $\mathcal{S}_c(a, \bar{a}^2)$ is larger than σ, it is not necessary to check for the third iteration of the projection.

In this example, the only common important terms are on the x and y dimensions, therefore we project all other document vectors only on x and y dimensions. The resulting Z-shape, which is a new hyperplane extracted from the original Z-order, is illustrated in Fig. 3a.

All candidate documents should appear in this new generated Z-order, except for the points that have zero value in their Z-order prefixes for the coordinates used in the projection (e.g., x, y dimensions for this example). Intuitively, such points are either biased toward the other dimensions, which are ignored or the corresponding documents contain none of the important terms. In any case, the cosine similarity of these points with the queried point would be very low.

As the projected vector of document b has the value of 0 for the two important terms (i.e., x and y dimensions), it will be filtered out in the search. On the other hand, documents c, d and e are considered as candidates. While document c has high values for both important terms corresponding to x and y dimensions, document e contains relatively small values for both of these terms. And finally, document d contains the important term corresponding to y dimension, but only with a small value.

If the query contained the document c, then all three coordinates x, y and z would correspond to important terms. However, since $\bar{c}^1$ is $(16, 16, 0)$ and $\mathcal{S}_c(c, \bar{c}^1) > \sigma$, only the Z-shape with x and y dimensions would be considered for searching the most similar documents, omitting the third dimension z. Consequently, the document b would again be filtered out even though b and c share a common important term in the z dimension. Note that, if the search were done for document d, then the Z-shape with y and z dimensions, as illustrated in Fig. 3b would be considered, since the important terms that guarantee the σ threshold reside on those dimensions.

As a different type of example, suppose that we have document query $f = (1, 1, 0)$. This document has $(0, 0, 0)$ value for the projections $\bar{f}^1$, $\bar{f}^2$ and $\bar{f}^3$; hence it will be projected on none of the dimensions. As it does not contain any important term, it will not be considered as a query term and filtered out from the search result.

Example 4 Let the threshold and the precision parameters be set as $\sigma = 0.8$ and $\lambda = 3$, respectively. Also let the data set have the following three documents with $\delta = 13$,

$$a = (0, 27, 17, 0, 5, 9, 0, 11, 6, 11, 0, 13, 14)$$
$$b = (0, 27, 21, 0, 0, 0, 15, 0, 5, 0, 0, 6, 10)$$
$$c = (0, 0, 0, 29, 0, 0, 16, 0, 0, 0, 4, 0, 5).$$

Using the first and second iteration projections, we obtain the following vectors for the document a,

$$\bar{a}^1 = (0, 27, 17, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$$
$$\bar{a}^2 = (0, 27, 17, 0, 0, 9, 0, 11, 0, 11, 0, 13, 14)$$

The corresponding cosine similarities are computed as

$$\mathcal{S}_c(a, \bar{a}^1) = \frac{a \cdot \bar{a}^1}{||a|| \cdot ||\bar{a}^1||} = 0.7590,$$

$$\mathcal{S}_c(a, \bar{a}^2) = \frac{a \cdot \bar{a}^2}{||a|| \cdot ||\bar{a}^2||} = 0.9826,$$

where $a \cdot \bar{a}^j = \sum_{i=1}^{13} a[i] \cdot \bar{a}^j[i]$, $||a|| = \sqrt{\sum_{i=1}^{13} a[i]^2}$, and $j \in (1, 2)$. As can be observed, the second iteration projection $\bar{a}^2$ satisfies the given threshold. Therefore, documents whose second iteration projections do not share any term with $\bar{a}^2$ (i.e., that have a zero tf-idf score for the corresponding terms that appear in $\bar{a}^2$), will be filtered out and their cosine similarities will not be computed. Note that, $\bar{a}^2$ has seven nonzero terms as opposed to nine nonzero terms in the original vector a, which potentially eliminates unnecessary similarity comparisons.

Here, $\mathcal{S}_c(b, a)$ is computed since $\bar{b}^2$ and $\bar{a}^2$ have common terms. However, $\mathcal{S}_c(c, a)$ is not computed since $\bar{a}^2$ and $\bar{c}^2$ do not share any common term. Although a and c have a common term (i.e., the last term), it is omitted due to its low tf-idf value in c. Indeed, the document b is much closer to a as the following cosine similarities of the original documents indicate

$$\mathcal{S}_c(a, b) = 0.8045,$$
$$\mathcal{S}_c(a, c) = 0.0494.$$

The threshold σ between a document and its projected version is data set dependent, and should be determined experimentally. The methods that are briefly introduced in this section, will be formalized in the subsequent sections.

3.3 Hadoop and MapReduce framework

In this work, we utilize the MapReduce programming framework, which employs a parallel execution and coordination model that can be used to manage large-scale computations over massive data sets [6,18]. One of the well known open source frameworks that supports the MapReduce model, is the Apache Hadoop framework [2], which is designed to run on clusters. The files are stored in a special file system called the distributed file system (DFS).

Data replication is one of the key factors that improves the effectiveness of the Hadoop framework, which survives node failures while utilizing a huge number of cluster nodes for data intensive computations. We implemented our algorithms using the MapReduce model in the Hadoop framework such that the details of the network communication, process management, interprocess communication, efficient massive data movement and fault tolerance are all transparent to the user. Typically, a developer needs to provide only configuration parameters and several high-level routines. The Hadoop framework is used by major actors including Google, Yahoo and IBM, largely for applications involving search engines and large-scale data processing (e.g., big data applications).

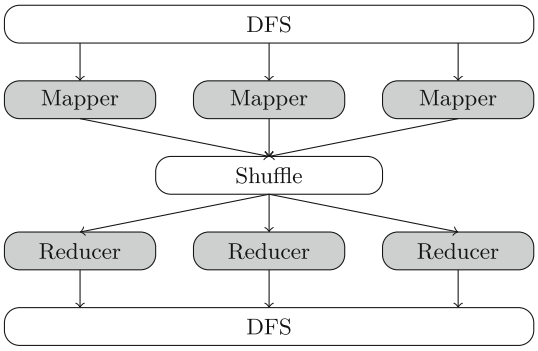

Fig. 4 MapReduce Execution Flow

The MapReduce model is based on two functions, *map* and *reduce*. The map function is responsible for assigning a list of data items, represented as key-value pairs, to cluster nodes. The map function receives key-value pairs, and sends the result as intermediate data to the reducer. The reducer function gets the mapped data and applies the processing operation. It receives the intermediate data as a key and a list of values as (key,[values]). The signature of these two functions are

$$\text{map}:(k_1, v_1) \rightarrow [(k_2, v_2)]$$
$$\text{reduce}:(k_2, [v_2]) \rightarrow [(k_3, v_3)].$$

The shuffling process, between the map and reduce functions in Fig. 4, is responsible for designating all keys with the same value to the same computation node.

4 The proposed filtering method

In this section we explain the proposed techniques of the ZOLIP filtering algorithm. The algorithm has three phases. In the first phase, the ZOLIP algorithm considers the document vectors that lie on the same Z-shape of λ order in the Z-order curve. In other words, this phase is designed to find highly similar (i.e., near duplicate) documents in the data set. If this phase cannot find k documents, the second phase considers the documents that are on different Z-shapes. In the second phase, the algorithm finds the most important terms in the queried document such that any document that does not contain any of those important terms, cannot be in the top k similar list of the queried document. Then the similarities with the documents sharing important terms are calculated. Finally, the last phase performs the actual join operations such that the outputs of the first and the second phases are combined and sorted to find the top k similar list of the queried document.

4.1 Phase 1: near-duplicate detection (NDD)

Initially, the first phase reads each document record that is composed of a key/value pair. While the key represents a unique document id, the value holds a vector which contains the tf-idf values of the terms of the corresponding document. Instead of the classical vectors, sparse vector representation is used for improving the performance and easing memory requirements as most of the vector elements are zero.

Sparse vector is a data structure that stores only the none-zero entries together with corresponding index values. We assume that the number of dimensions (δ), the number of iterations on the curve (λ) and the maximum tf-idf value (μ) in the whole data set are publicly known and passed as a configuration parameter to the map function in the proposed method, which is described in Algorithm 1. Note that, the operations over sparse vectors are optimized even though the algorithm descriptions are given in classical forms. For instance, the for loop in Steps 4–6 in Algorithm 1 is only executed for non-zero tf-idf values.

The map function in Algorithm 1 receives the tf-idf scores of each document in the data set $\mathcal{D}$, its id, $docId$, the maximum tf-idf value μ, the dimension δ, and the precision λ for tf-idf values.

The "bitLength" function (Step 2) returns the bit length of the binary representation for the maximum tf-idf value (μ). Then, the method goes over the coordinate values and normalizes them so that they are represented with the number of bits used for the maximum tf-idf value (Step 5). The "Normalize" function gets the bit length of each coordinate tf-idf value and finds the difference (dif) between the number of bits of the maximum tf-idf and the tf-idf of the current coordinate, and pads (dif)-many zero bits to the left of the actual coordinate value to make all the tf-idf values represented with the same number of bits.

Then, the map function extracts the bits of the normalized tf-idf value of each term starting from the most significant bit using the function "getBit" (Step 9), which returns the lth most significant bit. This operation will be repeated for each of the λ bits of the tf-idf scores. At each iteration, the mapper interleaves the extracted bit and concatenates it to a $ZOLIP$ key (Step 10), which is a sparse vector that represents the none-zero bits in the Z-value.

The key/value pair sent to the reducer is composed of the $ZOLIP$ key and the list of document identifiers and their tf-idf scores as in $(ZOLIPkey, \langle docId : d\rangle)$. All the documents that have the same $ZOLIP$ key are grouped by the shuffle process and sent to the reducer instances.

The reducers receive the list of documents that have the same ZOLIP key and calculates the cosine similarity between these records to find the top-k similar documents for each document. The function "findTKSD" (which stands for "find top-k similar documents") calculates the similarity between documents and sorts the documents depending on their similarity. Then, it selects the top-k similar documents and saves them in a sorted list "$ksim_i$".

Algorithm 1 Near-Dupplicate Detection(NDD)

```
1: procedure MAP(docId, d, μ, δ, λ)
2:     γ ← bitLength(μ)
3:     ZOLIP ← null
4:     for j = 1 to δ do
5:         d[j] ← Normalize(d[j], γ)
6:     end for
7:     for l = 1 to λ do
8:         for j = 1 to δ do
9:             bit ← getBit(d[j], l)
10:            ZOLIP ← ZOLIP||bit
11:        end for
12:    end for
13:    sendToReducer(ZOLIP, ⟨docId : d⟩)
14: end procedure

    ▷ listDOC contains the documents with the same ZOLIP key
15: procedure REDUCE(ZOLIP, {⟨docId : d⟩} )
16:    for all d_i ∈ listDOC do      ▷ for each document in listDOC
17:        ksim_i ← null
18:        for all doc_j ∈ listDOC and i ≠ j do
19:            ksim_i ← findTKSD()    ▷ compute similarity and sort
20:        end for

21:    end for
22:    return ∀i (⟨docId_i : d⟩ , ⟨{⟨ksim_i, S_c⟩}, (yes/no)⟩)
23: end procedure
```

Figure 5 illustrates an example, in which the mapper creates the ZOLIP keys Z_1, for documents "a" and "b" and Z_2 for document "c". Then, it passes the key value pairs to the reducer. The reducer receives the documents with the same ZOLIP key, and compares these documents to find the top-k similar documents.

The reduce function returns the records in the format $(\langle docId_i : d\rangle , \langle\{\langle ksim_i, \mathcal{S}_c\rangle\}, (yes/no)\rangle)$. The first element is the key of the record (i.e., the identifier of the document that we intent to find its k-similar documents). The second element, which is the value of the record, contains two parts. The first part is a list that contains the top k-similar document identifiers with their corresponding similarity values as ($\mathcal{S}_c$). The second part (i.e., (yes/no)) is an indicator that shows whether the top-k similar documents are found or not. If k similar documents are found in this step, no further search is applied for this document. Otherwise, further search in a different Z-shape is required.

Algorithm 1 represents the pseudo code based on the MapReduce programming model. The algorithm can be used both for self-join and R–S join operations. Algorithm 1 is described for self-join operation, in which we find k similar documents for every document in the data set. In R–S join operation, documents should be labeled as query documents and data set documents. Consequently, in Step 16 of Algo-

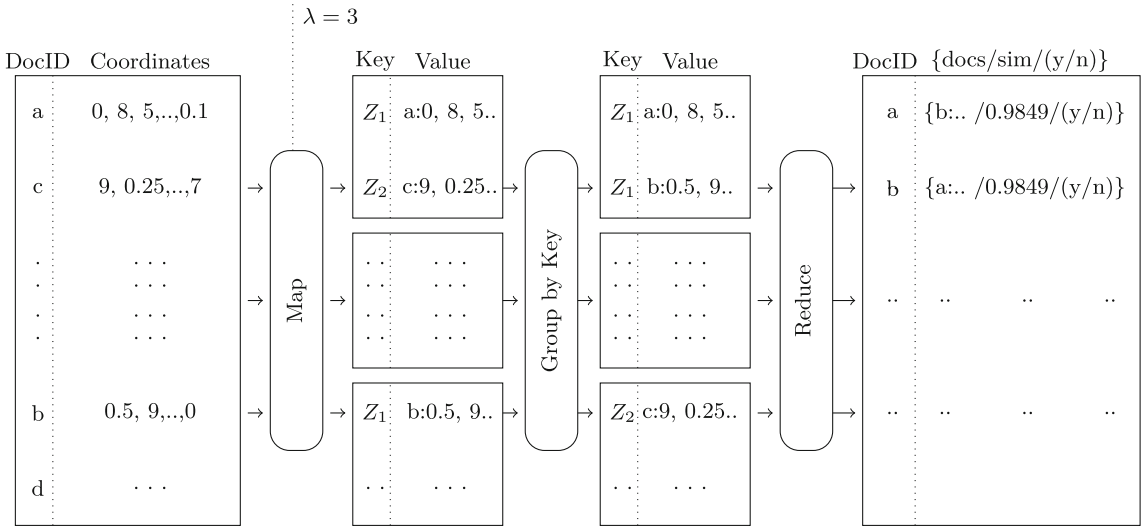

Fig. 5 An Example Execution of ZOLIP Phase 1

rithm 1 only documents in the query set should be considered while Step 18 should use the documents in the data set.

The NDD phase is expected to find k similar documents for a relatively large number of documents without further computations needed for them. While this reduces the overall complexity of the top-k similarity search, further computation is needed for the documents, for which k similar documents are not found. To this end, we propose a method based on common important terms to complete the computation for the remaining part of the database.

4.2 Phase 2: common important terms (CIT)

Some documents, although similar, may reside in different Z-shapes. In order to calculate the similarity between two such documents, we stipulate that they share at least one important (i.e., high tf-idf valued) term. We, therefore, propose utilizing only the most important terms to eliminate similarity computation for the distant documents. Note that the probability of two documents, that do not share any important term, being in the top-k similar documents of each other is very low.

The common important terms (CIT) phase contains two mapper and one reducer functions. The first mapper function reads the output of the NDD phase to check the (yes/no) value for a document. If k similar documents are not found for the document, the second phase, CIT, utilizes the lth iteration projection in Z order curves to determine the important terms in the document that will be used to filter out dissimilar documents.

The first mapper (the procedure MAP1() in Algorithm 2) computes the projection $\bar{d}_i^1$ of document d_i, which contains tf-idf values of the most important terms in the document. Then, the similarity between $\bar{d}_i^1$ and the original vector d_i will be calculated. If the similarity threshold is satisfied, namely $\mathcal{S}_c(d_i, \bar{d}_i^1) \geq \sigma$, the terms in $\bar{d}_i^1$ are sent to the reducer. Otherwise, a higher degree projection $\bar{d}_i^l$ is calculated to satisfy the threshold.

Here, MAP1() creates a key-value pair for each important term, in which the term is used as a key, and the vector d_i and the document id are considered as the value of the pair. In other words, the original record for the document is replicated as many times as the number of important terms in the projection $\bar{d}_i^l$. In order to distinguish the documents in the query set from the documents in the data set (the latter will be processed by the second mapper MAP2() function), a prefix "Q:" is added in the value part of the pair. The resulting record is of the format, $(term, \langle Q : docId : d_i \rangle)$.

Using the setting given in Example 4, Fig. 6 illustrates an example execution of the CIT algorithm steps for finding similar documents that have common important terms. Note that, in the mapper stage, the term indices that are exposed to the reducer correspond only to the terms with high importance (i.e., the terms that appear in $\bar{a}^2$).

The second mapper, (the procedure MAP2() in Algorithm 2), performs the same operations as the first mapper, but this time on the data set documents, with the only difference that, the prefix "D:" is added instead of "Q:". The resulting record format is of the form $(term, \langle D : docID : d_i \rangle)$. As in the case of the document in the query set, MAP2() creates as many new records as the number of important terms for the corresponding document.

The reducer in this phase receives the key value pairs from the first and the second mappers. Then, it partitions the records into two sets, $Qlist$ and $Dlist$, depending on whether the document is in the query or in the data set. The first mapper returns the important terms in the query documents combined with their coordinates, while the second mapper returns the important terms in data set documents

Algorithm 2 Common Important Terms(CIT)

```
1: procedure MAP1(docId, ⟨{⟨ksimilar, S_c⟩}, (yes/no)⟩ , λ, μ)
2:                                  ▷ this mapper reads the output of phase 1
3:   if (yes/no) = no then
4:     d_i^0 ← createEmptyVec()
5:     sim ← 0
6:     for l = 1 to λ AND sim < σ do
7:                                  ▷ λ is the number of bits in tf-idf scores
8:       d_i^l ← Project(d_i, l)
9:       sim ← S_c(d_i, d_i^l)
10:    end for
11:    for j = 1 to δ do
12:      if d_ij^l ≠ 0 then
13:        term ← d_ij^l
14:        sendToReducer(term, ⟨Q : docId, d_i⟩)
15:      end if
16:    end for
17:  end if
18: end procedure

19: procedure MAP2(docId, d_i))   ▷ It reads the data set documents
    and functions exactly as the first mapper.
20:   sendToReducer(term, ⟨D : docId, d_i⟩)
21: end procedure

22: procedure REDUCE(term,{⟨(Q/D) : docId, d_i⟩})
                ▷ Documents will be partitioned into query and data sets
23:   QList ← null
24:   DList ← null
25:   for all d_i do
26:     if d_i is a query document then
27:       addQList(d_i)
28:     else
29:       addDList(d_i)
30:     end if
31:   end for
32:   for all d_i ∈ Qlist do
33:     candidates_i ← null
34:     for all d_j ∈ Dlist do
35:       candidates_i ← docIdD_j
36:     end for
37:     for d_j ∈ candidates_i do
38:       return (⟨docId_i : docId_j⟩, S_c(d_i, d_j))
39:     end for
40:   end for
41: end procedure
```

combined with their coordinates. The reducer creates two partitions. The partition can easily be done by checking the prefixes in the value part, Q and D. Consequently, for every document in the query and data set, the reducer stores them in $Qlist$ and $Dlist$, respectively.

The last stage of the reducer calculates the similarity between $d_i \in Qlist$ and $d_j \in Dlist$. Then, the identifiers of d_i and d_j are combined to form a key for the pair in which the value is $\mathcal{S}_c(d_i,d_j)$. Consequently, the final output of this phase is of the form, $(\langle docId_i : docId_j\rangle, \mathcal{S}_c(d_i,d_j))$, where d_i and d_j represent the documents in the query set and the data set, respectively. Figure 6 visualizes all the steps on the setting from Example 4.

4.3 Phase 3: join phase (JP)

In the join phase, in order to find the actual top-k similar documents, the join operation is applied on the results from the first and the second phases. This phase contains two mappers. The first mapper receives the output from the first phase and sends one key-value pair to the reducer for each document in the list $ksim_i$, in which $docId$ is the key and the tf-idf score vector of the document in $ksim_i$ is the value. Thus, the output of the first mapper is of the form, $(docIdQ,\langle docIdD : \mathcal{S}_c\rangle)$.

The second mapper also works in a similar way. It receives the pair, where the key is the similar document identifiers $\langle docIdD, docIdQ\rangle$ and the value is the corresponding similarity. The mapper parses the $docIdQ$ and $docIdD$. After this step, it exposes the $docIdQ$ as a key and $\mathcal{S}_c$ prefixed with $docIdD$ as a value. The output of this mapper is also of the form, $(docIdQ, \langle docIdD : \mathcal{S}_c\rangle)$.

Algorithm 3 Join Phase

```
procedure MAP1(docId, ⟨⟨ksim, S_c⟩ , (yes, no)⟩)
                                    ▷ this mapper will reads phase 1 output
  docIdQ ← docId
  for all docIdD ∈ ksim do
    sendToReducer(docIdQ, ⟨docIdD : S_c⟩)
  end for
end procedure
procedure MAP2(⟨docIdQ : docIdD⟩ , S_c))
                                    ▷ this mapper will reads phase 2 output
  sendToReducer(docIdQ, ⟨docIdD : S_c⟩)
end procedure
procedure REDUCE(docIdQ, {⟨docIdD, S_c⟩})
  topK ← Sort(docIdD, S_c)
  expose(docIdQ,topKSimilar)
end procedure
```

The last step in the algorithm is to get all the candidate documents and select the top-k similar documents among them. The reducer receives the $docIdQ$ key from the previous mappers and a list of documents that contain the same key (i.e term) as the value. Here, the list is sorted and the first k documents with the highest similarity values are considered as the top-k similar documents. The final result of this reducer is of the form, $(docIdQ, \{\langle docIdD : \mathcal{S}_c\rangle\})$. Figure 7 visualizes the Join phase of the outputs of Examples 2 and 4, where the details are given in Algorithm 3.

4.4 R–S join

The R–S join is used to find k most similar documents for a batch of query documents, which are not a part of the data set. The algorithms given above are described mostly for self join operation that finds top k similar documents for every document from the same data set. In order to adapt the proposed algorithms for R–S join operations, we need to label documents with "Q:" and "D:". Also, several simple

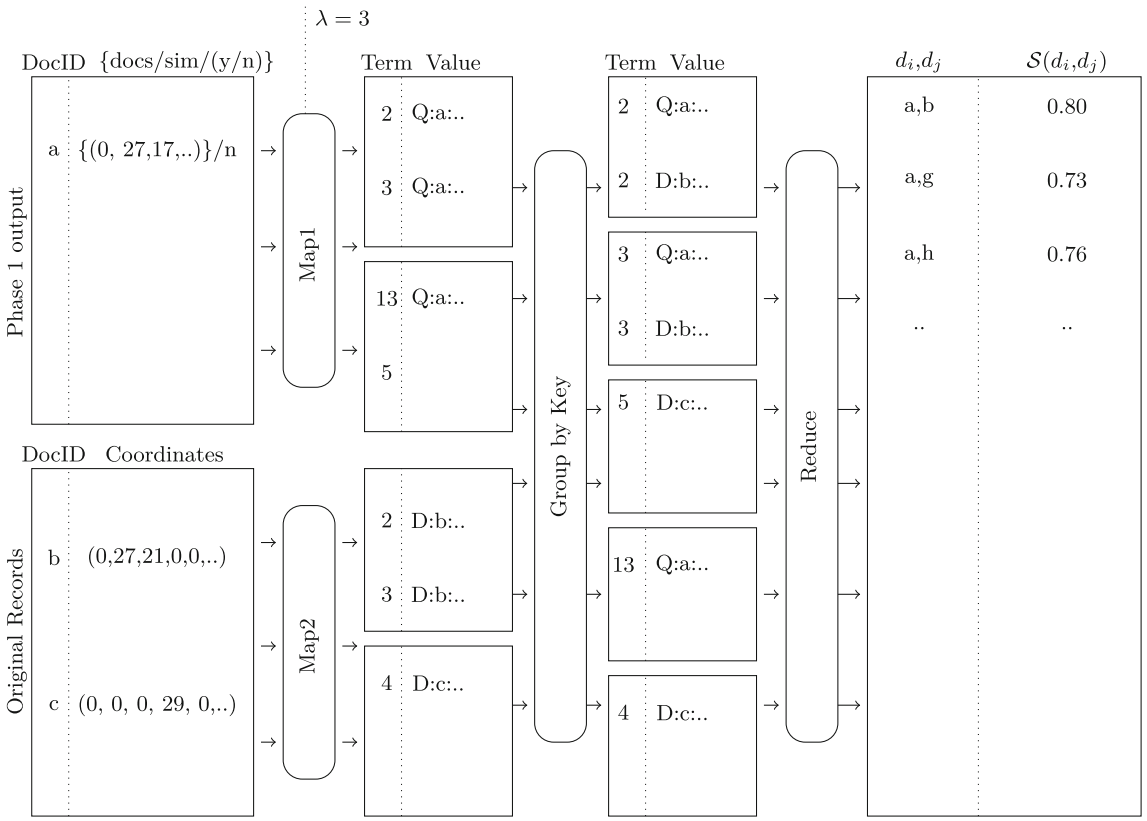

Fig. 6 An Example Execution of ZOLIP Phase 2

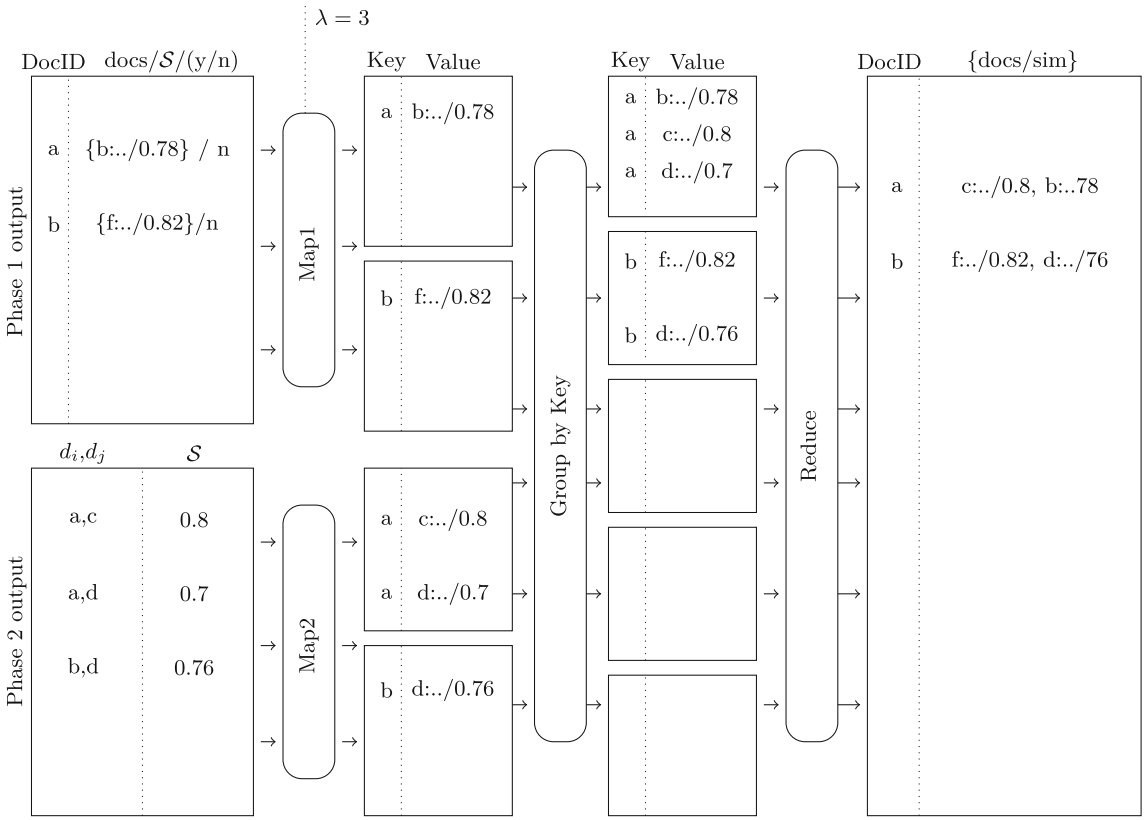

Fig. 7 An Example Execution of ZOLIP Phase 3

modifications are necessary to the algorithm steps. As the modifications are only minor and straightforward, we do not include separate algorithms for R–S join operations.

The time complexity of the R–S join depends on the number of documents in the query set and an R–S join usually requires much less time than the self join since the number of the queried documents is expected to be much lower than the number of documents in the data set. We analyze the effect of the query size by testing the algorithm on queries with various number of documents in the next section.

5 Experiments

In this section, we first describe the system model and the data sets used in our experiments. Then, we compare the proposed method with the method by Vernica et al. [21]. The proposed method is tested with different parameter settings to show its efficiency and scalability in the big data context. Finally, we discuss the accuracy of our method and the effect of increasing λ parameter on its accuracy and efficiency.

5.1 Setup and data description

We used the Cloudera CDH4 installation with a 3-node cluster: Two nodes with Xeon processor E5-1650 3.5 GHz of 12 cores, 15.6 GB of RAM and 1 TB hard disk; and one node with Core i7 3.07 GHz of 8 cores, 15.7 GB of RAM and 0.5 TB hard disk. On each node, Ubuntu 12.04 LTS, 64 - bit operating system is installed and Java 1.6 JVM is used. We used the default Cloudera DFS block size (i.e., 128 MB), so the data is partitioned into 128 MB blocks and a mapper is assigned to each block. As the system is assumed to have no a priori knowledge about the size of the queries, no additional partition is applied before assigning data blocks to mappers. Note that, the data and the query sets are stored in different data blocks in the DFS.

The outputs of the mappers are sent to the reducer directly without any additional combiner function. Note that, query elements labeled with "Q" are compared with data elements labeled with "D". As the types of the outputs of query and data mappers are different, aggregate calculations using combiner operation after mapping are not possible.

In our experiments, we used primarily the Enron data set [11] which is a data set of real e-mail files. The data set consists of 510,000 e-mail files, which is equivalent to a total of 2,686,224 KB data. The size of the files are ranged from 4 KB to 2 MB. First, the data is cleaned from the mail headers and stop words. The terms are stemmed to their roots using the *snowball stemmer* included in the Lucene [15], which is an open source library for information retrieval. We, then, calculated the *tf-idf* values of each term in each file. Finally, we created the sparse vectors of *tf-idf* values, which represent the documents. For the Enron data set, the resulted sparse vectors have 30,000 dimensions (i.e., terms) with an average density of 160/30,000. The density is defined as the ratio of the number of terms with non-zero scores over the total number of terms in the data set [23].

Additionally, to observe the performance of the algorithm on a different data set, we used the Reuters data set [13], which consists of 20,000 news records of 28 MB, with a dictionary size of 136 terms.

5.2 Performance analysis

The performance of our algorithm is compared with the performance of the algorithm proposed by Vernica et al. [21]. Figure 8 shows the absolute running times of our implementations of the two algorithms for various query set sizes on the Enron data set for top 10 results (i.e., $k = 10$). As the size of the Enron data set is very large, which is overwhelming for a small three-node Hadoop cluster used in the experiments, the Java heap memory suffers from the huge amount of joins for the algorithm in [21]. Hence, the timing results for the self join case for Vernica's method is omitted in the paper. A possible solution for this problem is to use a larger cluster with more nodes.

We conducted different experiments to observe the effects of different parameter settings on the performance of the proposed method. The performance of the method for different values of λ is illustrated in Fig. 9. Figure 9 demonstrates that for small values of λ the effect of query size is very low. However, for $\lambda \geq 8$ search time drastically increases as query size increases, which is due to the significant increase in the number of join operations (i.e., similarity calculation between a pair of documents). A trivial remedy to decrease the computation time is to utilize a stronger cluster with more nodes.

Figure 10 shows the running times of each phase, namely NDD, CIT and JP, for different query set sizes. We observed

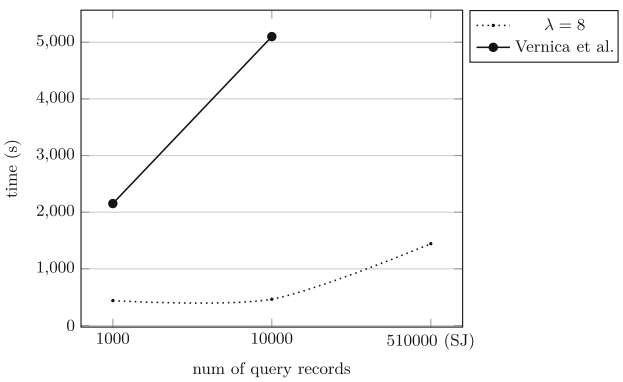

Fig. 8 Performance Comparison between the Proposed Algorithm (ZOLIP) and the Method by Vernica et al. [21] for $k = 10$

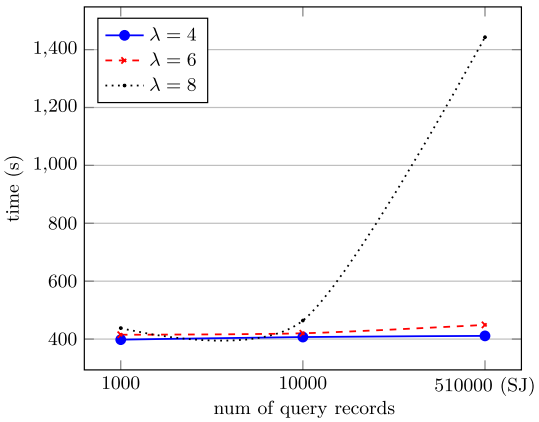

Fig. 9 Effect of Increase in λ on Efficiency for $k = 10$

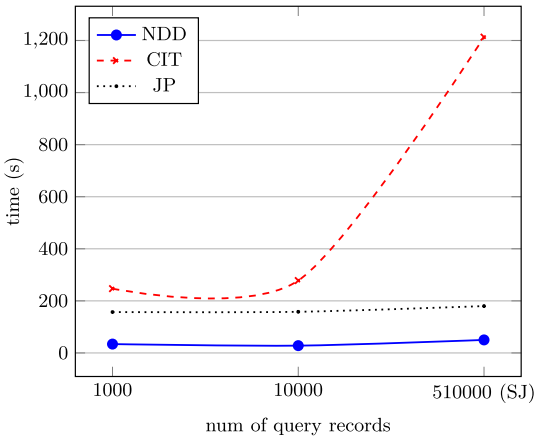

Fig. 10 Running Time of Each Phase for $\lambda = 8$

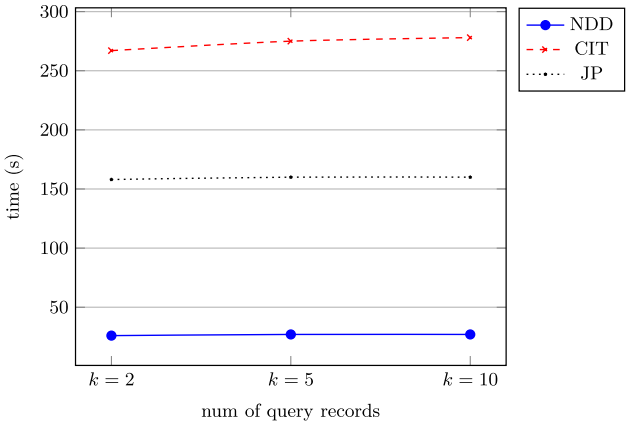

Fig. 11 Running Time of Each Phase for different k where, Query Size is 10,000 and $\lambda = 8$

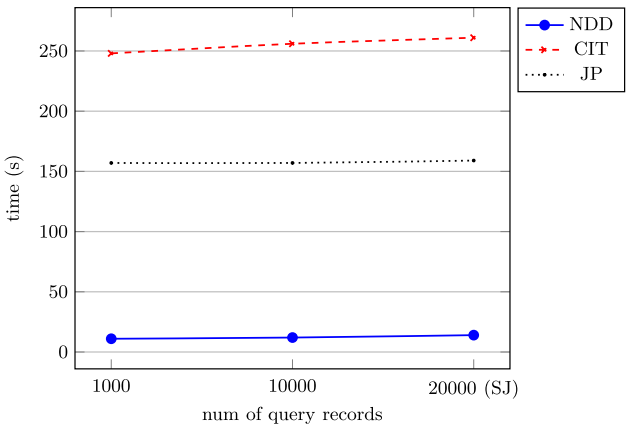

Fig. 12 Running Times for the Reuters data set for $\lambda = 8$

that, the running time of the CIT phase increases with the number of queried documents which is expected as this phase is the main part of the algorithm where all the documents that share a common important term is compared. The NDD phase is constant with the number of queried documents. Although the complexity of this phase is linear with respect to the dictionary size (δ), it performs constant as we use a static data set. We also analyze the effect of the increase in k on the efficiency of the method and illustrate the results in Fig. 11. As the figure illustrates, k does not have a major effect on the similarity search time. The reason for that, independent of k, given a query document our algorithm calculates the similarity scores with all other documents that share an important term with the query document and then selects the top k results. However, if the first phase (NDD) succeeds in finding k or more results then the second phase is not executed; hence the algorithm may perform better for small values of k.

Generally, from our experiments we observed that the similarity of the documents that are returned from the first phase (i.e., near duplicate detection phase) with the query documents (i.e., $\mathcal{S}_c(q, d_i)$) are over 80 %. However, this similarity depends on the nature of the data set used as well as the parameter setting of λ.

The Enron data set is composed of e-mails, which tend to contain duplicate parts, e.g., e-mails written using a common template, which may be advantageous for the first phase (NDD) of our algorithm. Therefore, we also analyzed the performance of our algorithm using the Reuters Corpus Volume 1 (RCV1) data set, which is a corpus of news wire stories that is made available by Reuters, Ltd. [13]. This data set has about 20,000 data elements which is much smaller than the Enron data set. The running times of each phase are demonstrated for the Reuters data set in Fig. 12.

From Fig. 12, it can be observed that although the Reuters data set is much smaller than the Enron data set, the search times are almost equivalent (*cf.* the running time for the Enron data set in Fig. 10). This is due to the fact that, there are only a few near duplicate documents in the Reuters data set. While in the Enron data set, 12 % (120 out of 1000 queries) of the search operations can be addressed using only the NDD

phase using $k = 2$, this value reduces to 1.1 % (11 out of 1000 queries) in the Reuters data set.

5.3 Accuracy analysis

In this section, we analyze the accuracy of our algorithm and the trade-off between performance and accuracy. We define the accuracy of our method using the so-called *ground truth*, where the k-most similar document output of the proposed scheme is compared with the actual k-most documents calculated using the full tf-idf scores in the data set. The ratio of the cardinality of the intersection of both sets to k is defined as the accuracy rate of the algorithm. The corresponding accuracy rate of Vernica's [21] algorithm is 1.0 as they used the *PPJoin* and *PPJoinn+* [22], which is an exact similarity searching algorithm.

The parameter λ determines the accuracy rate of the proposed method. Let e_{max} be the number of bits required to represent the maximum tf-idf value in the data set and in the query set. In the proposed method during filtering, each tf-idf value is represented by only the most significant λ bits. Therefore, the tf-idf values that are less than $2^{e_{min}}$, where $e_{min} = e_{max} - \lambda$, are considered as zero in the ZOLIP filtering algorithm.

Generally, we have two types of errors that lead to loss of accuracy. In the first type, all the terms of a document have tf-idf values less than $2^{e_{min}}$. Since the terms are all filtered out, no similar document can be found for the documents with very low tf-idf values. In the second type, as the terms can only be partially represented after the filtering, there can be inaccuracies in the returned k-similar matches.

Let $\mathcal{N}_{ge}$ be the number of documents in the query set that have at least one non-zero tf-idf value in the λth iteration of Z-order (i.e., having at least one term with tf-idf value greater than $2^{e_{min}} - 1$). As those documents have at least one important term that is represented after the filtering, they are free of the errors of the first type. Therefore, we can formulate the rate of the documents, for which no similar documents can be found (i.e., type 1 error) as:

$$Err_{Type_1} = \frac{|Q| - \mathcal{N}_{ge}}{|Q|}, \tag{3}$$

where $|Q|$ is the number of documents in the query. Note that, as λ increases, more documents are represented in the λth iteration on Z-order, which increases the accuracy.

In the second type of error, the results are not exactly the same as the actual k-similar documents in the ground truth. There are again two possible causes for those errors. In the first case, the first phase (NDD) outputs more than k documents. The matches found are, indeed, highly similar to the queried document as they are detected in the *Near Duplicate Detection* phase. However, the ordering can be different than the actual one in the ground truth and hence, we will miss some of the actual documents among the topmost k documents. Note that, such an error does not occur if NDD phase outputs less than or equal to k results.

The second case are related to the type-1 errors. There may be some documents in the data set that should actually be in the k similar list of a query, but missed in our method due to their low tf-idf scores. As the proposed method is a filtering technique, the final result can only be an approximation to the actual result given in the ground truth.

We tested the accuracy of the method using queries of 10,000 documents compared with the Enron data set of 510,000 documents. Table 2 shows the average number of missed queries (i.e., type-1 error) for the cases of $\lambda = 4, 6$ and 8. Note that, the error rate significantly decreases as λ increases.

Table 3 shows the average accuracy of the method for the queries that returns a result, which are the types of queries, for which type-1 error does not occur.

We noticed that, none of the documents for which no similar documents is found, has a term that contains a tf-idf value that appears in the λth iteration Z-order. Therefore, the results confirm the formula in Eq. (3) for average type-1 error rate.

The same accuracy analysis are also performed for the Reuters data set for $\lambda = 4, 6$ and 8 with k=10, and 10,000 documents, where the results are provided in Table 4.

The average type-1 error rate can be decreased by increasing the value of $\mathcal{N}_{ge}$, which can be achieved by increasing λ. However, the increase in λ, deteriorates the performance of the method as documents that do not contain important terms would also be compared with the queried documents.

Table 2 Average of missed queries of ZOLIP filtering algorithm with $k = 2$

λ	4	6	8
Missed queries	4309	2466	895
Average of missed queries	43 %	25 %	9 %

Table 3 Accuracy of the top k documents for ZOLIP filtering algorithm with $k = 2$

λ	4	6	8
# Queries with inaccurate top k	11	144	327
Accuracy	99 %	98 %	96 %

Table 4 Accuracy of the top-k documents for ZOLIP filtering algorithm with $k = 2$

λ	4 (%)	6 (%)	8 (%)
Average of missed queries	13	12.5	11
Accuracy	99	98	96

Table 5 Accuracy of ZOLIP filtering algorithm with different values of k when $\lambda = 8$

k	2	5	10
Wrong k-similar	327	384	433
Accuracy	96 %	95.6 %	95.2 %

Table 6 Relative error on the sum (RES) for different values of k when $\lambda = 8$

k	2	5	10
RES	0.0262	0.0216	0.0178

In order to decrease type-1 error rate for the query documents that do not have any representation in the λth iteration Z-order, we boost the term with the highest tf-idf value so that document can now be represented in the λth iteration Z-order. A similar approach is adopted also for the second case of type-2 errors by boosting the tf-idf values of documents that cannot be represented.

Let B be the boosting factor and let $t_{i,max}$ be the maximum tf-idf value of document i. Then, B is defined as $B = 2^{e_{min}}/t_{i,max}$. Note that, only the term with the maximum tf-idf value is boosted and all the other values remain the same. We tested the effectiveness of the proposed boosting method using $\lambda = 8$. While the original method has 895 false matches over 10,000 queries, the boosted method has 891 false matches which provides only a slight improvement in the number of missed queries. However, the search time is almost doubled. The reason for this degradation in performance is that, the documents that do not initially appear in the λ levels, (i.e., all the tf-idf values are very small) appear in the boosted version, which significantly increases the required number of comparisons.

Generally, in any data set, it is difficult to find the similarity for the documents that do not contain keywords with sufficiently high tf-idf values by using the cosine similarity techniques. For such a document, the Jaccard similarity metric may perform better than the cosine similarity based approaches. The proposed ZOLIP method provides efficient and very accurate results especially in data sets that contain longer documents with several important terms.

We also consider the effect of the increase in k on the accuracy of the method. We applied the ZOLIP filtering algorithm with $\lambda = 8$ and we set $k = 10$ to check if it reduces the accuracy of our filtering algorithm. The increase in k does not affect the accuracy in any significant manner, as shown in Table 5.

We also measured the accuracy of the method using relative error on the sum of distances (RES) [12] metric. Let $S_{q,k}$, be the top-k similar result from a query q. And let also $G_{q,k}$ be the ground truth of the top-k similar documents, then RES is defined as

$$RES(q, S_{q,k}) = \frac{\sum_{x \in S_{q,k}} \mathcal{S}_c(q, x)}{\sum_{y \in G_{q,k}} \mathcal{S}_c(q, y)} - 1.$$

RES is calculated for the Enron data set with 10,000 queries, using $\lambda = 8$, and $k = 2, 5, 10$. Table 6 shows the relative error values after removing the queries with type-1 error from the calculations.

6 Conclusion

In this paper, we propose an efficient filtering method based on the Z-order prefix that can be used in document similarity algorithms with cosine similarity metric. The algorithm provides effective and efficient results. In particular, it is very fast in returning highly similar documents. We demonstrate that our method performs much better than the algorithm in [21] for the cases, where the density of the vectors is relatively high. This is especially true when the variance in the tf-idf values are high. This is a general observation in data sets, in which majority of the documents can be represented accurately with only several terms of relatively high tf-idf values.

On the other hand, our algorithm have some limitations for documents, all of whose terms have very low tf-idf values. The documents with terms of very low tf-idf values, however, imply that they cannot be accurately represented by a particular subset of terms. Therefore, a filtering based on important terms do not produce accurate results for those kinds of documents. Instead, we propose to use a hybrid filtering method based on the Jaccard similarity metric for documents with no important terms while our filtering technique can be used for the documents with at least several important terms.

The proposed method provides results with very high accuracy if the accuracy parameter λ is set to a large value. However, for relatively small values of λ, the search time significantly decreases while still satisfying reasonable accuracy. There is a tradeoff between accuracy and efficiency, hence the accuracy parameter λ should be set according to the requirements of the application and the data set. For instance, in finding highly similar documents in applications such as duplicate web site and plagiarism detection, a relatively small values of λ can produce results with high accuracy.

Acknowledgments This project is supported by TUBITAK under Grant Number 113E537.

References

1. Angiulli, F., Pizzuti, C.: An approximate algorithm for top-k closest pairs join query in large high dimensional data. Data Knowl. Eng. **53**(3), 263–281 (2005)
2. Apache Hadoop. http://hadoop.apache.org
3. Arasu, A., Ganti, V., Kaushik, R.: Efficient exact set-similarity joins. In: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB '06, pp. 918–929. VLDB Endowment (2006)
4. Baraglia, R., De Francisci Morales, G., Lucchese, C.: Document similarity self-join with MapReduce. In: 2010 IEEE 10th International Conference on Data Mining (ICDM), pp. 731–736 (2010). doi:10.1109/ICDM.2010.70
5. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: Proceedings of the 16th International Conference on World Wide Web, WWW '07, pp. 131–140. ACM, New York (2007). doi:10.1145/1242572.1242591
6. Brown, R.A.: Hadoop at home: large-scale computing at a small college. SIGCSE Bull. **41**(1), 106–110 (2009). doi:10.1145/1539024.1508904
7. Chaudhuri, S., Ganti, V., Kaushik, R.: A primitive operator for similarity joins in data cleaning. In: Proceedings of the 22nd International Conference on Data Engineering, ICDE '06, p. 5. IEEE Computer Society, Washington, DC (2006). doi:10.1109/ICDE.2006.9
8. Connor, M., Kumar, P.: Fast construction of k-nearest neighbor graphs for point clouds. IEEE Trans. Vis. Comput. Graph. **16**(4), 599–608 (2010). doi:10.1109/TVCG.2010.9
9. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008). doi:10.1145/1327452.1327492
10. 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: Short Papers. HLT-Short '08, pp. 265–268. Association for Computational Linguistics, Stroudsburg (2008)
11. Enron Dataset. http://www.cs.cmu.edu/~./enron/
12. Falchi, F., Perego, R., Lucchese, C., Rabitti, F., Orlando, S.: A metric cache for similarity search. In: LSDS-IR (2008)
13. Lewis, D.D., Yang, Y., Rose, T.G., Li, F.: RCV1: a new benchmark collection for text categorization research. J. Mach. Learn. Res. **5**, 361–397 (2004). http://dl.acm.org/citation.cfm?id=1005332.1005345
14. Li, R., Ju, L., Peng, Z., Yu, Z., Wang, C.: Batch text similarity search with mapreduce. In: Du, X., Fan, W., Peng, Z., Sharaf, M.A. (eds.) APWeb. Lecture Notes in Computer Science, vol. 6612, pp. 412–423. Springer, Heidelberg (2011)
15. Lucene. http://lucene.apache.org/
16. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, New York (2008)
17. Phan, T.C., d'Orazio, L., Rigaux, P.: Toward intersection filter-based optimization for joins in mapreduce. In: Cloud-I'13, p. 2 (2013)
18. Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets. Cambridge University Press, Cambridge (2012)
19. Sarawagi, S., Kirpal, A.: Efficient set joins on similarity predicates. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD '04, pp. 743–754. ACM, New York (2004). doi:10.1145/1007568.1007652
20. Tao, Y., Yi, K., Sheng, C., Kalnis, P.: Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. ACM Trans. Database Syst. **35**(3), 20:1–20:46 (2010). doi:10.1145/1806907.1806912
21. Vernica, R., Carey, M.J., Li, C.: Efficient parallel set-similarity joins using mapreduce. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, pp. 495–506. ACM, New York (2010). doi:10.1145/1807167.1807222
22. Xiao, C., Wang, W., Lin, X., Yu, J.X.: Efficient similarity joins for near duplicate detection. In: Proceedings of the 17th International Conference on World Wide Web, WWW '08, pp. 131–140. ACM, New York (2008). doi:10.1145/1367497.1367516
23. Yang, B., Myung, J., Lee, S.G., Lee, D.: A mapreduce-based filtering algorithm for vector similarity join. In: Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication, ICUIMC '13, pp. 71:1–71:5. ACM, New York (2013). doi:10.1145/2448556.2448627
24. Zhang, C., Li, F., Jestes, J.: Efficient parallel knn joins for large data in mapreduce. In: Proceedings of the 15th International Conference on Extending Database Technology, EDBT '12, pp. 38–49. ACM, New York (2012). doi:10.1145/2247596.2247602
25. Zhu, S., Wu, J., Xiong, H., Xia, G.: Scaling up top-k cosine similarity search. Data Knowl. Eng. **70**(1), 60–83 (2011)

Mahmoud Alewiwi received his M.S. degree in computer science from University Putra Malaysia in 2006 and worked as an instructor in Hebron University, Palestine between 2006 and 2010. Currently, he is a Ph.D. student in computer science at Sabanci University, Istanbul, Turkey. His main research interests are BigData mining, Distributed Systems, and Natural Language Processing.

Dr. Cengiz Orencik received his B.S., M.S. and Ph.D. degrees in computer science from Sabanci University, Istanbul, Turkey, in 2006, 2008 and 2014, respectively. Since 2015, he is an Assistant Professor at Beykent University, Istanbul, Turkey. His main research interests are data security, privacy preserving data mining, BigData and biometric security.

Dr. Erkay Savaş is a professor at Sabanci University, Istanbul, Turkey. He received the BS (1990) and MS (1994) degrees in electrical engineering from the Electronics and Communications Engineering Department at Istanbul Technical University. He completed the Ph.D. degree in the Department of Electrical and Computer Engineering (ECE) at Oregon State University in June 2000. He had worked for various companies and research institutions before he joined Sabanci University as an assistant professor in 2002. He is the director of the Cryptography and Information Security Group (CISec) of Sabanci University. His research interests include cryptography, data and communication security, privacy in biometrics, trusted computing, security and privacy in data mining applications, embedded systems security, and distributed systems.