SN Computer Science (2020) 1:1
https://doi.org/10.1007/s42979-019-0007-y

ORIGINAL RESEARCH

A Lightweight Indexing Approach for Efficient Batch Similarity Processing with MapReduce

Trong Nhan Phan[1] · Tran Khanh Dang[1]

Received: 12 April 2019 / Accepted: 13 June 2019 / Published online: 25 June 2019
© Springer Nature Singapore Pte Ltd 2019

Abstract

Similarity search is a principle operation in different fields of study. However, the cost for that operation is expensive due to several reasons, mainly by redundancy and big data load. There are many approaches that concentrate on how to speed up similarity search, especially with massive datasets, so that we can employ it for plenty of recent applications. In this paper, we study an efficient way for either single or batch similarity processing with MapReduce while minimizing redundant data by building lightweight indexes from the data and query sources. More specifically, we propose a general query processing scheme that not only handles a single query but also deals with sets of query in an incremental manner. In addition, we build the indexes in an ordered fashion, the so-called sorted inverted indexes, so that we can perform our quick pruning strategy that discards unrelated objects. Moreover, we embed metadata inside the indexes to reduce inessential duplicates. Last but not least, we measure our proposed solution by conducting empirical experiments on real datasets. The results verify the efficiency of our method when we do similarity search with query batches, especially when both query sets and datasets are large.

Keywords Similarity search · Batch processing · Lightweight indexing · Metadata · MapReduce · Hadoop

Introduction

Similarity search is the principle operation not only in databases but also in interdisciplinary fields of study such as machine learning, recommendation systems, biology, or data analytics. The main goal of similarity search is to find the object similar to the given pivot, know as the query. The similarity computing, unfortunately, suffers high overheads due to distance calculation pair by pair together with similarity metrics [18]. It is more inevitable especially when data grow bigger and have a higher dimensionality. As a result, the challenge is how to speed up the similarity searching process to enjoy its benefits.

This article is part of the topical collection "Future Data and Security Engineering" guest edited by Tran Khanh Dang.

✉ Trong Nhan Phan
nhanpt@hcmut.edu.vn

Tran Khanh Dang
khanh@hcmut.edu.vn

[1] HCMC University of Technology, VNU-HCM, Ho Chi Minh City, Vietnam

There are several approaches to achieve the goal, for example, by improving indexing [3, 15], hashing [13], filtering [16], or processing in parallel [1]. Among them, the parallel processing approach in distributed environment calls much attention to researchers worldwide [6, 9, 10, 14]. This trend keeps motivating both academia and industry due to the large amount of data. Other approaches may fail to assure scalability when processing massive datasets. As a result, we catch the trend by employing MapReduce, a large-scale processing paradigm [4], when enhancing the performance of similarity search.

Even though MapReduce helps us process an enormous amount of data, it would suffer heavy overheads when processing big unnecessary or irrelevant objects. The scenario becomes even worse when those big objects are involved in the similarity search. In other words, those redundant objects are combined with every other object to evaluate their similarity pair by pair. Moreover, the MapReduce-based process is strictly bound by I/O costs, so processing irrelevant or unnecessary data leads to extra penalty.

Meanwhile, those recent literatures only deal with a single similarity query. Consequently, when given a query batch, each query is processed one by one, which slows down the whole performance of batch processing. In fact, we observe

that queries in the batch may share their search space. As a consequence, it would be better to search the same shared search space for all queries in the batch rather than looking for the search space for one query and then redo it several times for other queries.

In our work, we, therefore, take batch processing into account rather than single query processing. Furthermore, we propose our strategies to make our MapReduce-based solution more effective. Specifically, our main contributions are as follows:

- We present a query batch processing scheme that not only handles sets of query but also does similarity search in an incremental way.
- We introduce a simple but efficient method to support quick pruning in similarity search by sorted inverted indexes.
- We propose an indexing scheme with metadata so that we can diminish duplicate data and then build lightweight indexes.
- We perform our empirical experiments on real datasets. The results verify the efficiency of our proposed solution when it does similarity search with query batches.

It is worth noting that this paper is the extended version of our work [8]. The new content is added as follows:

- We further discuss about our incremental computing with our query batch processing scheme in Sect. "Query Batch Processing Scheme".
- We present our MapReduce-based algorithms and their descriptions in Sect. "MapReduce-based Similarity Search".
- We show our MapReduce-based algorithms with lightweight indexing and their descriptions in Sect. "Lightweight Indexing".
- We analyze how our proposed solution can support pairwise similarity query, range query, as well as k-nearest neighbor query in Sect. "Towards Other Similarity Queries".
- We add more descriptions about similarity search and MapReduce in Sects. "Similarity Search" and "MapReduce".

The rest of paper is organized as follows. Sect. "Related Work" presents our related work. Additionally, Sect. "Preliminaries" introduces our background related to the similarity search and MapReduce paradigm. In Sect. "Our Proposed Solution", we propose our solution for similarity search in general and that with query batches in particular. After that, we conduct our experiments in Sect. "Empirical Experiments and Evaluations" before making our remarks in Sect. "Conclusion and Future Work".

Related Work

Metwally and Faloutsos introduce a method for all-pair similarity joins of multisets and vectors [6]. Their method is composed of two main phases, each one comprised of two MapReduce jobs. While a MapReduce job is costly, the greater the usage of MapReduce jobs is, the higher the costs will be. Additionally, their method does not consider duplicate data items during the execution of MapReduce job, which usually adds extra penalty. Besides, Tang et al. presented their way, the so-called HA index, to speed up Hamming-based distance computing for range queries [14]. In addition, redundancy during the computing is also eliminated. Their whole process consumes three phases with two MapReduce jobs. Nevertheless, the costs for data pre-processing and post-processing are excluded from the MapReduce jobs.

Gao et al. bring up efficient and scalable metric similarity joins with MapReduce [5]. They focus on the load balancing and how to avoid unnecessary object pairs with their filtering methods, including the range-object filtering, the double-pivot filtering, the pivot filtering, and the plane sweeping techniques, so that they can achieve better query performance. In the meantime, Phan et al. proposed an efficient hybrid similarity search with MapReduce [10]. Their basic idea is to first cluster similar objects and second define upper and lower boundaries to shrink the search space before looking at similar pairs. The method is then to deploy in a hybrid MapReduce-based architecture that deals with challenges from big data. In addition, their empirical studies show that their method is efficient in terms of data processing and storage. Though their method works well with batch processing, each query is sequentially processed in a batch with regard to their search scheme, while we have indexing strategy for query batches supporting quick similarity search.

Nguyen et al. built the VP tree algorithm on top of the MapReduce framework to achieve good performance, scalability, and fault tolerance for similarity search over the large datasets in the distributed environment [7]. Moreover, their method can reduce the number of data that need to scanned during the search phase. In contrast, our approach is towards MapReduce-based scheme-driven algorithms that are independent of the underlying MapReduce framework. By doing this, we are able to gain two main advantages as follows: (1) no internal or additional changes from the framework; and (2) mutual support from both the top algorithms and the underlying framework.

Preliminaries

Similarity Search

A corpus, denoted as Ω, consists of a set of document objects D_p, which is formally represented as

$\Omega = \{D_1, D_2, D_3, \dots, D_n\}$. In addition, each document object D_p is composed of a set of words, which is shown as $D_p = \{word_1, word_2, word_3, \dots, word_w\}$. Similarity search is the operation that retrieves all objects in the corpus Ω so that these objects satisfy required constraints. The constraints may be different from query to query. There are various similarity queries with different constraints. In this section, we present some fundamental similarity queries as follows:

- **Pairwise similarity query.** For each document object in the corpus Ω, the similarity search computes how much similarity is present between one document object and every $(n - 1)$ other object in the corpus.
- **Query-by-example or pivot query.** The similarity search looks for similar objects in the corpus Ω when given a query object called a pivot. When given a pivot Q_j, the similarity search computes how much similar the pair (Q_j, D_p) is for every document object D_p in the corpus.
- **Range query.** When given a pre-defined range threshold ϵ, or sometimes it is known as a similarity threshold, the similarity search retrieves all objects in the corpus Ω, whose similarity scores are greater or equal to the threshold ϵ.
- **K-nearest neighbor query.** When given a pre-defined k parameter, the similarity search retrieves all objects in the corpus Ω such as they are the top-k similar objects. In other words, they are k objects that are the most similar to a query object.

To evaluate how similar a pair is, a similarity measure such as Euclidean distance, Cosine similarity, Hamming distance, or Jaccard coefficient is used to quantify their similarity [18]. The similarity score is usually standardized into the interval [0, 1] in that the pair is more similar when its similarity score is close to 1, while it is less similar when its similarity score is near 0. Moreover, a similarity threshold, like 80% similarity may be provided to filter those pairs whose similarity scores are greater or equal to 0.8.

In the meantime, we observe that modeling the content of a document as a set of words does not reflect much how really similar a pair is, because the two same words in different objects do not bring the same meaning. To better improve a part of semantic similarity, a concept of K-shingles [11], known as any sub-string having the length K found in the document, is used instead. As a consequence, each document object D_p is composed of a set of K-shingles, which is shown as $D_p = \{S_1, S_2, S_3, \dots, S_z\}$.

Furthermore, to speed up the process of similarity search, different types of indexing are employed. One of the most well-known indexing supporting similarity search is the inverted index, a popular data structure used in information retrieval systems [2, 10]. In our work, we build another version of inverted index known as the sorted inverted index so that we can skip unnecessary computing for those elements not in either document or query objects when searching candidate pairs, which is discussed later on in Sect. "MapReduce-based Similarity Search".

In general, a similarity search process consists of two main phases as follows:

1. Candidate generation phase. This is the phase where two objects are identified as a candidate pair.
2. Candidate verification phase. This is the phase where the pair is evaluated for its similarity score.

MapReduce

MapReduce is a parallel paradigm for large-scale processing [4]. The philosophy behind is to apply "divide-and-conquer" strategy to data. A large data is split into different smaller data chunks, which are then processed at various machines. The intermediate results generated by each machine are aggregated into the final result. To implement this strategy, a MapReduce job is composed of a Map task and Reduce task in that the former is specified by a Map function, while the latter is determined by a Reduce function. When a MapReduce job is executed on a cluster of commodity machines, those machines assigned Map tasks called mappers, whereas those assigned Reduce tasks are known as reducers. A Map task emits intermediate key-value pairs, while a Reduce task writes the final key-value pairs into the distributed file system. Moreover, there is a shuffle phase between the Map task and the Reduce task, which re-distributes data based on the output keys by the mappers.

Suppose that there are M mappers and R reducers, a single MapReduce job is described as follows:

1. Input data are loaded into the distributed file system and then divided into partitions based on their data size.
2. Mappers read their data partitions, perform the Map function, and emit intermediate results in the form of key-value pair $[k_i, v_j]$. These intermediate key-value pairs are locally stored at mappers.
3. The shuffle process aggregates the intermediate key-value pairs $[k_i, [v_j]]$ into R data partitions, which is based on their key values.
4. Reducers retrieve the intermediate key-value pairs $[k_i, [v_j]]$ from R data partitions and perform the Reduce function. The final output is written back to the distributed file system.

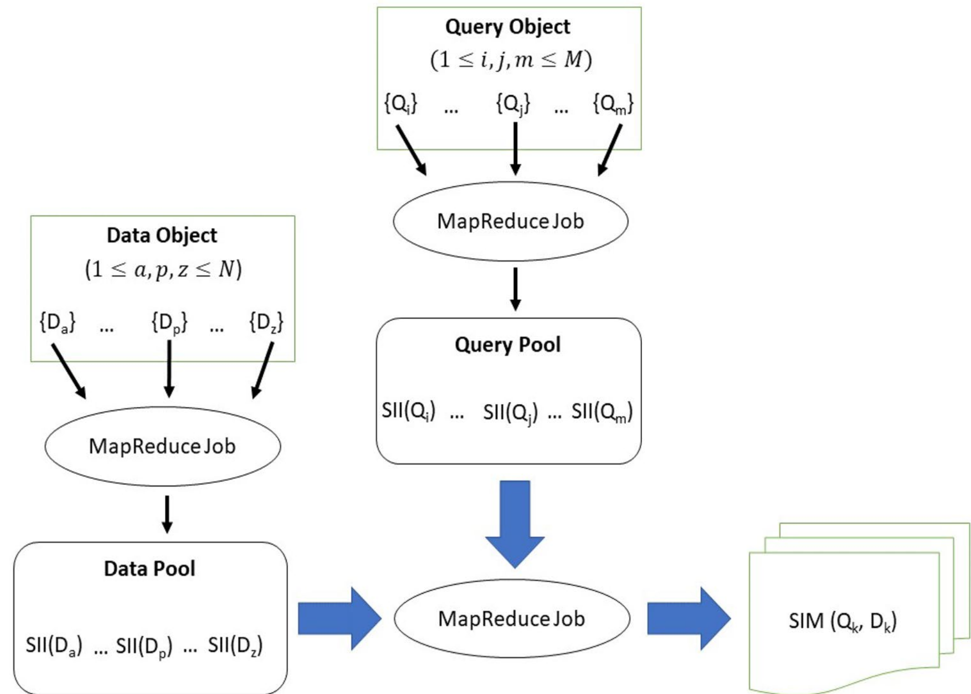


Fig. 1 Query batch processing scheme [8]

One of the most well-known open-source framework that implements the idea of MapReduce is Hadoop,[1] which is designed to perform scale-up computing with massive datasets. In our work, we employ Hadoop to deploy our MapReduce-based algorithm for efficient batch similarity processing.

Our Proposed Solution

Query Batch Processing Scheme

In this paper, we introduce our general similarity search scheme as illustrated in Fig. 1. Either data or query objects are indexed into either data or query pools, respectively. Besides, we employ inverted index as an index data structure. In addition, the indexes are organized in an ordered way to serve our quick pruning strategy later on. In general, our basic idea is to separate the data preparation phase from the similarity search phase. By doing it this way, we can perform incremental computing when data or query objects change. In other words, we do not need to pre-process either existing or unchanged data and query objects, but with those which are new or have been changed. To index data objects, we use one MapReduce job. For instance, a set of data object $\{D_p\}$ is indexed into the data pool in the form of sorted inverted index (SII), known as SII(D_p). Similarly, a set of query object $\{Q_j\}$ is indexed into the query pool in the form of sorted inverted index, known as SII(Q_j). After the indexing phase, both data and queries are ready for similarity search. Later on, another MapReduce job computes the similarity (SIM) among queries against data and produces the final result in the form SIM(Q_j, D_p). In our work, we employ Jaccard coefficient, a well-known metric for fast set-based similarity [6, 10, 12, 17], to derive the similarity score of a pair as described in the Eq. 1 below.

$$SIM(Q_j, D_p) = \frac{\| Q_j \cap D_p \|}{\| Q_j \cup D_p \|} \tag{1}$$

In that, $\| Q_j \cap D_p \|$ is the intersection cardinality between Q_j and D_p, while $\| Q_j \cup D_p \|$ is the union cardinality between Q_j and D_p.

With the proposed scheme, we can perform similarity search in an incremental manner. The reason is that both data and query objects are available in the data and query pools in the form of SII. Hence, if there are new data or query objects, the pools will include them in the form of SII. Other existing data and query objects remain unchanged, because they have already been in the data and query pools, respectively. Then, a MapReduce job for similarity search can be configured to compute similarity scores from a set of SII in the pools as required. Furthermore, the general scheme is applied not only to single query processing but also to query batches. Figure 2 shows three cases for incremental computing with our proposed scheme as follows.

[1] https://hadoop.apache.org/.

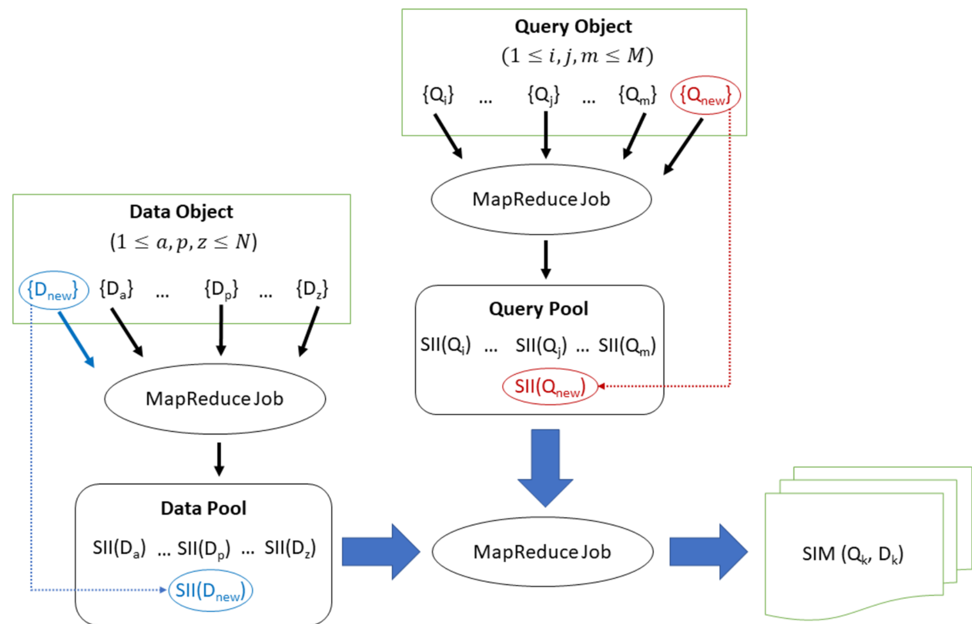


Fig. 2 Incremental processing scheme

- **Case 1. New data objects appear.** Assume that there is a set of new data objects $\{D_{\text{new}}\}$. A MapReduce job processes it to produce SII(D_{new}) in the data pool. It is worth noting that SII(D_{new}) is now ready in the data pool and is independent of other forms of SII from other existing data objects (i.e., SII(D_a) ... SII(D_p) ... SII(D_z)). Consequently, when doing similarity search according to a particular query set (e.g., SII(Q_i) ... SII(Q_j) ... SII(Q_m)), another MapReduce job takes those relevant inputs from the data pool, which already includes new data objects.
- **Case 2. New query objects appear.** Assume that there is a set of new query objects $\{Q_{\text{new}}\}$. A MapReduce job processes it to produce SII(Q_{new}). It is worth noting that SII(Q_{new}) is now available in the query pool and is independent of other forms of SII from other existing query objects (i.e., SII(Q_i) ... SII(Q_j) ... SII(Q_m)). As a consequence, when doing similarity search according to a particular dataset (e.g., SII(D_a) ... SII(D_p) ... SII(D_z)), another MapReduce job takes those relevant queries from the query pool, which already includes new query objects.
- **Case 3. Both new data and query objects appear.** This is the combination of Case 1 and Case 2 above. In other words, both new data and query objects emerge together at the phase of doing a similarity search. Like the other two cases, if we do not take new data and query objects into account, the similarity search would produce its out-of-date result. Therefore, new data and query objects had better be processed to be available in the data and query pools, respectively. When conducting a similarity search, another MapReduce job pulls those related data and query input to compute up-to-date similarity scores.

MapReduce-Based Similarity Search

Our MapReduce-based similarity search following the above scheme consists of two main phases: (1) indexing; and (2) searching. In the former phase, we will index data and query objects into the pool in the form of SII while doing the similarity search with Jaccard measure in the latter phase. Last but not least, we model our documents as bags of 4-shingles rather than sets of words [10, 11].

As illustrated in Fig. 3, at Phase 1: indexing, Map-1 takes original objects as its input. It is worth noting that objects mentioned here include data and query objects. Map-1 then processes the input and emits intermediate key-value pairs of the form [Element, URL] in that Element is a shingle of an object, while URL is the uniform resource locator of that object in the distributed environment. Here we do not use object identification as we want to clearly know where the object is in the distributed system so that we can retrieve that object after the similarity search. Next, Reduce-1 aggregates those intermediate key-value pairs emitted by Map-1 with regard to their key values. Moreover, Reduce-1 sorts the list of key-value pairs in the form of [Element, $[URL]]_{ord}$. At Phase 2: Searching, Map-2 starts considering candidate pairs for their similarity from those object already indexed at Phase 1. If the query object and the data object share the same element, Map-2 generates the candidate pair of the form $[URL_D - URL_Q, ||D \cup Q||]$. After that, Reduce-2 aggregates the same candidate pairs, computes their similarity scores, and outputs the final result of the form $[URL_D - URL_Q, SIM(D, Q)]$.

To better understand our proposed method, we present here our algorithms for each MapReduce job. Figure 4

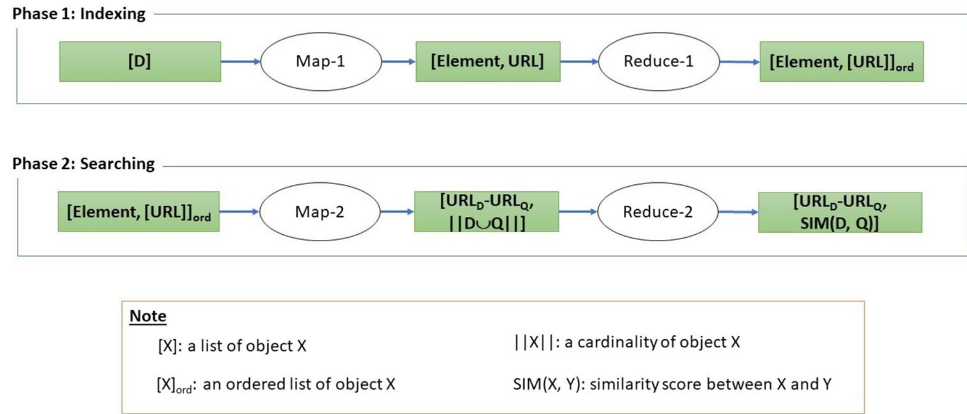


Fig. 3 MapReduce flow chart

Fig. 4 Map-1 algorithm

```
Algorithm 1: MAP-1
Input:
  - A set of documents [Di]
Output:
  - Intermediate key-value pairs [SHk, URLi@NOSi]
1  itemListi ← []
2  shingleListi ← []
3  count ← 0
4  itemListi ← GetShingles(Di)
5  foreach item in itemListi do
6  |  shingleListi.insert(count, item)
7  |_ count ← count + 1
8  shingleListi ← Filter(shingleListi)
9  NOSi ← GetLength(shingleListi)
10 URLi ← GetURL(Di)
11 foreach SHk in shingleListi do
12 |_ Emit(SHk, URLi ∪ NOSi)
```

illustrates Map-1 algorithm, which gets the input of documents D_i and then produces the intermediate key-value pairs of the form $[SH_k, URL_i@NOS_i]$. First, necessary variables are initialized as in steps 1–3. Next, we generate shingles from the input documents D_i as in step 4. For each shingle, we insert it into a $shingleList_i$, regarding the document D_i, as in steps 5–7. Later, we filter duplicate shingles as in step 8, because duplicates do not contribute to the overall similarity scores, which is based on the Jaccard measure. Besides, we get the number of shingles as in step 9 and get the URL_i of the corresponding document D_i as in step 10. Finally, we let mappers emit the intermediate key-value pairs as in steps 11–12.

Figure 5 illustrates Reduce-1 algorithm, which gets the input from Map-1 of the form $[SH_k, URL_i@NOS_i]$ and then builds a sorted inverted index, which has the form as $[SH_k, [URL_i@NOS_i]]_{ord}$. First, reducers read input data as in step 1, while necessary variables are initialized as in steps 2–5. Next, steps 6–17 aggregate those documents that share the same shingle, and we keep their tracks using a two-dimensional matrix. Then, we sort the matrix to create a sorted list of key values (i.e., shingle values) as in step 18. Finally, we let reducers emit the key-value pairs as in steps 19–20.

Figure 6 illustrates Map-2 algorithm, which gets the input including a set of sorted inverted index $SII(D_i)$ of document objects and a set of sorted inverted index $SII(Q_j)$ of query objects and then produces the candidate pairs of the form $[URL_D - URL_Q, NOS_{D \cup Q}]$. First, mappers read data and query inputs as in steps 1–2. Besides, we also get the query number information from the query set as in step 3. For each document and query, we examine whether there is any intersection between D_i and Q_j as in steps 4–11. If yes, we get necessary information such as URL_i in step 8, URL_j in

Fig. 5 Reduce-1 algorithm

```
Algorithm 2: REDUCE-1
Input:
 -  Intermedidate key-value pairs [SHk, URLi@NOSi]
Output:
 -  A sorted inverted index [SHk, [URLi@NOSi]]ord
1  data ← Read(Input)
2  prev ← null
3  docString ← ""
4  matrix ← [][]
5  index ← 0
6  foreach shingle, doc in Sorted(data) do
7     if (prev == shingle) then
8        docString ← docString ∪ doc
9     else
10       if (prev <> null) then
11          matrix[index].Append(prev)
12          matrix[index].Append(docString)
13          index ← index + 1
14       prev ← shingle
15       docString ← doc
16 matrix[index].Append(prev)
17 matrix[index].Append(docString)
18 sortedList ← Sort(matrix)
19 foreach x in range(0, len(sortedList) do
20    Emit(sortedList[x][0], sortedList[x][1])
```

Fig. 6 Map-2 algorithm

```
Algorithm 3: MAP-2
Input:
 -  A set of sorted inverted index SII(Di) of document objects
 -  A set of sorted inverted index SII(Qj) of query objects
Output:
 -  Candidate pairs [URLD-URLQ, NOSD∪Q]
1  data ← Read(Input)
2  query ← Read(Input)
3  queryNum ← GetQueryNumber(query)
4  foreach line in data do
5     foreach item in range(0, queryNum) do
6        if ((Di in line) ∩ (Qj in query) <> ∅) then
7        foreach doc in line do
8           URLi ← GetURL(doc)
9           URLj ← GetURL(query)
10          NOSD∪Q ← GetNOS(Qj in query) + GetNOS(Di in doc)
11          Emit(URLi-URLj, NOSD∪Q)
```

Fig. 7 Reduce-2 algorithm

```
Algorithm 4: REDUCE-2
Input:
 - Candidate pairs [URL_D-URL_Q, NOS_D∪Q]
Output:
 - Similar pairs [URL_D-URL_Q, SIM(D, Q)]
1  data ← Read(Input)
2  prev ← null
3  total ← -1
4  count ← 0
5  sim ← 0.0
6  foreach pair, num in Sorted(data) do
7     if (prev == pair) then
8        count ← count + 1
9     else
10       if (prev <> null) then
11          sim ← count / (total - count)
12          Emit(prev, sim)
13       prev ← pair
14       count ← 1
15       total ← num
16 sim ← count / (total - count)
17 Emit(prev, sim)
```

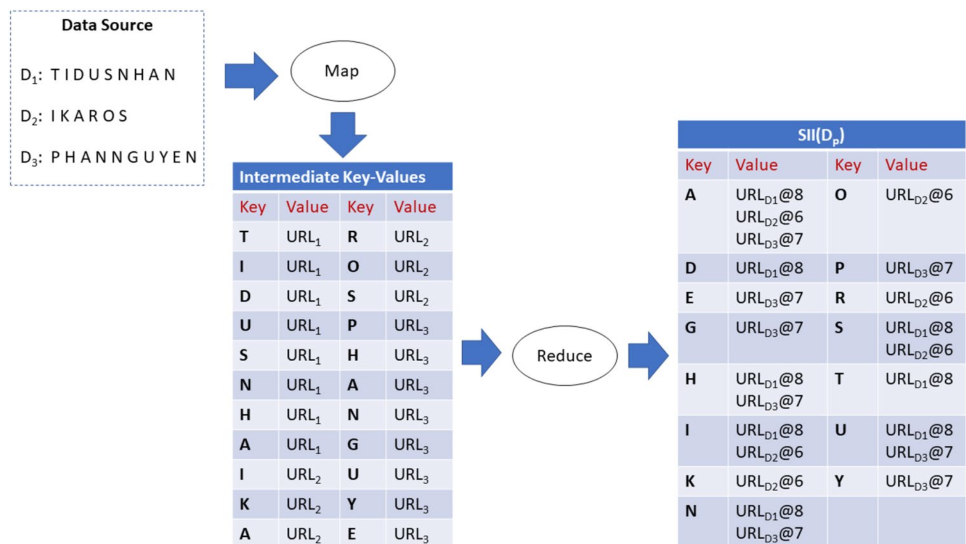


Fig. 8 Example of data indexing [8]

step 9, the total number of shingles between D_i and Q_j as in step 10, and let the mappers emit the candidate pairs as in step 11.

Figure 7 illustrates Reduce-2 algorithm, which gets the input from Map-2 of the form of candidate pairs $[URL_D - URL_Q, NOS_{D \cup Q}]$ and then produces similar pairs of the form $[URL_D - URL_Q, SIM(D, Q)]$. First, reducers read input data as in step 1, while necessary variables are initialized as in steps 2–5. Next, steps 6–17 aggregate the same candidate pairs, count the number of shingles shared by the two pair, and compute the similarity score between each pair.

Furthermore, Fig. 8 shows an example of data indexing by a MapReduce job. Assume that we have three data documents $D_p = [D_1, D_2, D_3]$ with their corresponding

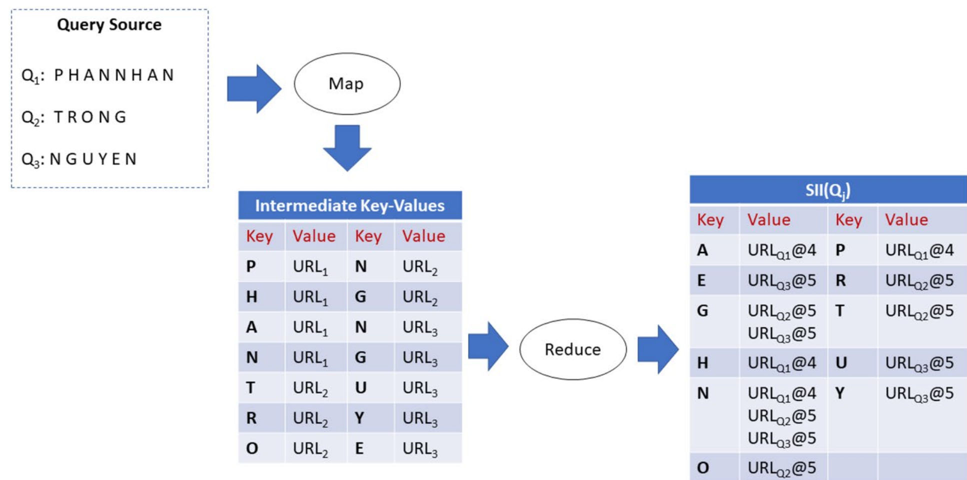


Fig. 9 Example of query indexing [8]

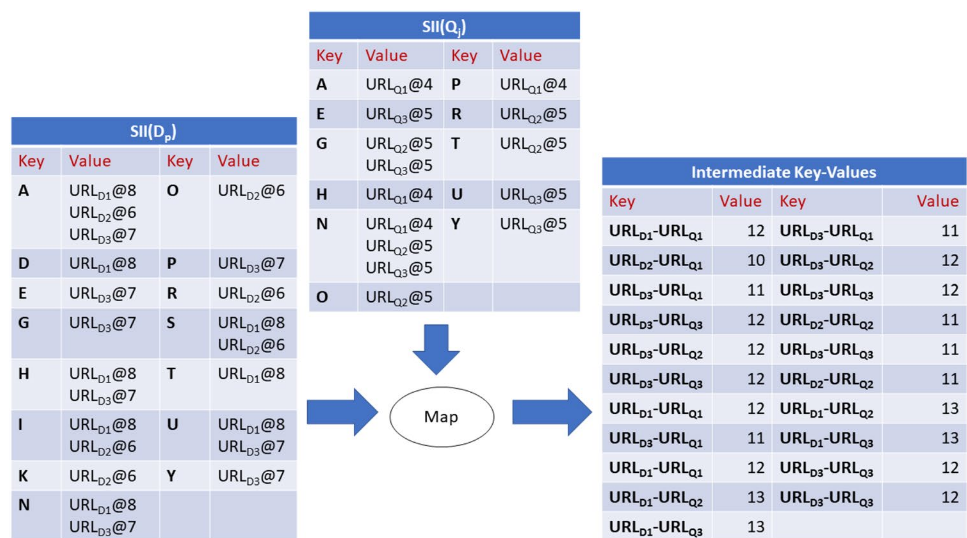
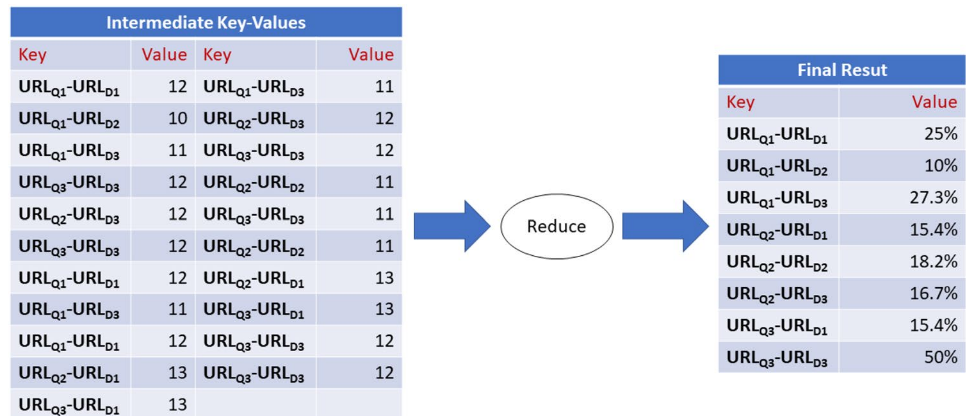


Fig. 10 Example of Map task [8]

shingle-based contents. The Map task is to emit intermediate key-value pairs in the form of $[SH_p, URL_p]$ in that SH_p is a shingle of a document D_p and URL_p is the path location of D_p in the distributed environment. It is worth noting that duplicate shingles from the same document are discarded, because they do not contribute to the similarity scores with Jaccard measure. We, thus, filter them at this Map task to avoid additional overheads after that. The Reduce task then produces $\text{SII}(\text{D}_p)$ in the form of $[SH_u, [URL_v@S_v]]$. It is worth noting that we need to keep the size S of those documents so that we can derive their similarity scores later on. Likewise, Fig. 9 implies an example of query indexing by a MapReduce job. Assume that we have three query documents $\text{Q}_j = [Q_1, Q_2, Q_3]$ with their corresponding shingle-based contents. The Map task is to emit intermediate key-value pairs in the form of $[SH_j, URL_j]$, whereas the Reduce task produces $\text{SII}(\text{Q}_j)$ in the form of $[SH_u, [URL_v@S_v]]$.

The similarity search phase is done by one MapReduce job. Figure 10 illustrates the Map task with $\text{SII}(\text{D}_p)$ and $\text{SII}(\text{Q}_j)$. It compares key by key and emits the pair whenever they share the same shingles in the form of $[URL_j - URLp, (S_j + S_p)]$. To sooner discard unnecessary pairs, we apply our quick pruning strategy in the comparison to speed up the searching process. Due to the fact that we already organize $\text{SII}(\text{D}_p)$ and $\text{SII}(\text{Q}_j)$ in an ordered way, we can achieve the two following advantages for our quick pruning during the comparison:

1. We can stop the comparison halfway whenever $\text{SH}_p > \text{SH}_j$.
2. We can discard those shingles from the comparing set $\text{SII}(\text{D}_p)$ whenever $\text{SH}_p < \text{SH}_j$. By doing it this way, we can reduce the size of the comparing set $\text{SII}(\text{D}_p)$ during the comparison.



Intermediate Key-Values

Key	Value	Key	Value
URL_{Q1}-URL_{D1}	12	URL_{Q1}-URL_{D3}	11
URL_{Q1}-URL_{D2}	10	URL_{Q2}-URL_{D3}	12
URL_{Q1}-URL_{D3}	11	URL_{Q3}-URL_{D3}	12
URL_{Q3}-URL_{D3}	12	URL_{Q2}-URL_{D2}	11
URL_{Q2}-URL_{D3}	12	URL_{Q3}-URL_{D3}	11
URL_{Q3}-URL_{D3}	12	URL_{Q2}-URL_{D2}	11
URL_{Q1}-URL_{D1}	12	URL_{Q2}-URL_{D1}	13
URL_{Q1}-URL_{D3}	11	URL_{Q3}-URL_{D1}	13
URL_{Q1}-URL_{D1}	12	URL_{Q3}-URL_{D3}	12
URL_{Q2}-URL_{D1}	13	URL_{Q3}-URL_{D3}	12
URL_{Q3}-URL_{D1}	13		

Final Resut

Key	Value
URL_{Q1}-URL_{D1}	25%
URL_{Q1}-URL_{D2}	10%
URL_{Q1}-URL_{D3}	27.3%
URL_{Q2}-URL_{D1}	15.4%
URL_{Q2}-URL_{D2}	18.2%
URL_{Q2}-URL_{D3}	16.7%
URL_{Q3}-URL_{D1}	15.4%
URL_{Q3}-URL_{D3}	50%

Fig. 11 Example of Reduce task [8]

Finally, the Reduce task aggregates the pairs emitted by the Map task and computes their similarity scores. As shown in Fig. 11, we will have eight pairs with their corresponding similarity scores.

Lightweight Indexing

By observing, we see that there are duplicate values in the indexes. Whenever a pair shares the same shingle, the value of the form $URL_v@S_v$ emerges. In the distributed environment, a URL may be long due to its location path. Therefore, those repeated values make the size of indexes bigger. As a result, they add to the heavy cost of MapReduce-based processing, since it is strictly bound by I/O cost. To minimize the redundancy and speed up the MapReduce-based searching process, we propose a metadata-based approach as follows.

1. We build a list L of document URLs as metadata in the form of $\{URL_v@S_v\}$. The metadata is put as header of each file produced by Reduce task.
2. For each value in the inverted index, we replace it with the index of L with regard to the corresponding document URL.

To implement our idea, we change the behaviors of Reduce-1 and Map-2 in that Reduce-1 builds lightweight indexes with metadata, while Map-2 processes the similarity search with those metadata generated by Reduce-1. Figure 12 illustrates Reduce-1 algorithm with metadata, which gets the input from Map-1 of the form $[SH_k, URL_i@NOS_i]$ and then builds a sorted inverted index as $[SH_k, [URL_i@NOS_i]]_{ord}$ with embedded metadata. First, reducers read input data as in step 1, while necessary variables are initialized as in steps 2–8. More specifically, the variable metaDataList, as in step 6, keeps the list of URLs, the variable m_index, as in step 7, keeps the last index of the metaDataList, while the variable current_m_index, as in step 8, keeps the current index of a document D_i. Steps 9–32 almost show the same way as in the original Reduce-1, which means those documents that share the same shingle are aggregated, and their tracks are kept using a two-dimensional matrix. However, the difference is that we check whether the examining document object has already been in the metaDataList or not as in step 11. If not, we generate the next index of the metaDataList and combine it with that object as in steps 12–15. Otherwise, we combine that object with its current index in the metaDataList as in steps 16–17. Moreover, we let reducers output metadata in metaDataList before the sorted inverted index list, as in steps 34–35.

Due to changes from data structure after Reduce-1, we also need to modify the Map-2 algorithm to match the new way of data processing. Figure 13 illustrates Map-2 algorithm with metadata, which still gets the input including a set of sorted inverted index SII(D_i) of document objects and a set of sorted inverted index SII(Q_j) of query objects and then produces the candidate pairs of the form $[URL_D - URL_Q, NOS_{D \cup Q}]$. First, mappers read data and query inputs as in steps 1–2. Next, we retrieve shingle list from query and store it in queryShingleList as in step 3 as well as metadata list from query and store it in queryMetaDataList. Besides, we use the variable shingle as in step 5 to store key element and value element. Moreover, we also use the variable metaDataFlag as in step 6 to separate the metadata processing as in steps 8–10 from the data processing as in steps 11–24. If metaDataFlag is true, we get the metadata information and store it into metaDataList. Otherwise, we start processing the data and let mappers emit candidate pairs. When comparing shingles from data and query, if the key value of data shingle is greater than the key value of query shingle and the length of queryShingleList is different from 0, we pop that query shingle out of the queryShingleList as in steps 12–13 due to the fact that key values in both data and query indexing are sorted in advance,

Fig. 12 Reduce-1 algorithm with metadata

```
Algorithm 5: REDUCE-1 with Metadata
Input:
 - Intermedidate key-value pairs [SHk, URLi@NOSi]
Output:
 - A sorted inverted index [SHk, [URLi@NOSi]]ord with embedded
   metadata
1  data ← Read(Input)
2  prev ← null
3  docString ← ""
4  matrix ← [][]
5  index ← 0
6  metaDataList ← []
7  mIndex ← 0
8  current m index ← -1
9  foreach shingle, doc in Sorted(data) do
10 |  if (prev == shingle) then
11 |     current m index ← getValueIndex(metaDataList, doc)
12 |     if (current m index == -1) then
13 |        metaDataList.insert(mIndex, doc)
14 |        docString ← docString ∪ mIndex
15 |        mIndex ← mIndex + 1
16 |     else
17 |        docString ← docString ∪ current m index
18 |  else
19 |     if (prev <> null) then
20 |        matrix[index].Append(prev)
21 |        matrix[index].Append(docString)
22 |        index ← index + 1
23 |     prev ← shingle
24 |     current m index ← getValueIndex(metaDataList, doc)
25 |     if (current m index == -1) then
26 |        metaDataList.insert(mIndex, doc)
27 |        docString ← mIndex
28 |        mIndex ← mIndex + 1
29 |     else
30 └─       docString ← current m index
31 matrix[index].Append(prev)
32 matrix[index].Append(docString)
33 sortedList ← Sort(matrix)
34 Emit('@MetaData@', metaDataList)
35 Emit('Query-Group-Ordered-List', sortedList)
```

whereas we only need to find those that are shared by both data and query at this phase. Otherwise, we retrieve query list and document list as in steps 14–16. Next, for each query in the query list and for each data in the data list, we get their necessary information such as the metadata index of the examining query as in step 18, URL of query as in step 19, the metadata index of the examining data as in step 21, URL of data as in step 22, and the total number of shingles between the document object D and the query object Q as in step 23. Finally, we let mappers emit the candidate pairs as in step 24.

Figure 14 shows an example of metadata of dataset, while Fig. 15 gives an example of metadata of query set. For instance, we build the metadata from the dataset as $\{URL_{D1}@8, URL_{D2}@6, URL_{D3}@7\}$. Consequently, the pair [A, $[URL_{D1}@8, URL_{D2}@6, URL_{D3}@7]$] is replaced by [A, [0, 1, 2]]. Meanwhile, we build the metadata from the query set as $\{URL_{Q1}@4, URL_{Q2}@5, URL_{Q3}@5\}$.

Fig. 13 Map-2 algorithm with metadata

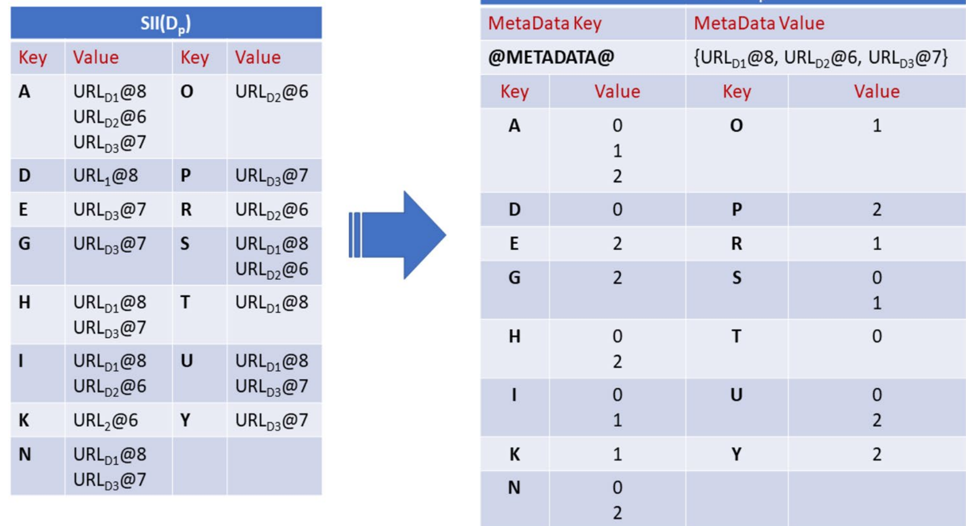
```
Algorithm 6: MAP-2 with metadata
Input:
 - A set of sorted inverted index SII(Di) of document objects
   with embedded metadata
 - A set of sorted inverted index SII(Qj) of query objects
   with embedded metadata
Output:
 - Candidate pairs [URLD-URLQ, NOSD∪Q]
1  data ← Read(Input)
2  query ← Read(Input)
3  queryShingleList ← GetQueryShingleList(query)
4  queryMetaDataList ← GetQueryMetaDataList(query)
5  shingle ← [,]
6  metaDataFlag ← true
7  foreach line in data do
8    if (metaDataFlag) then
9        metaDataList ← GetMetaData(line)
10       metaDataFlag ← false
11   else
12       while ((GetKeyValue(line) > shingle[0]
                 and (len(queryShingleList) <> 0)) do
13         shingle ← queryShingleList.pop()
14       if (GetKeyValue(line) == shingle[0])
              and (len(queryShingleList) <> 0)) then
15         queries ← GetQueryList(shingle[1])
16         docs ← GetDocList(line)
17         foreach query in queries do
18           queryMetaDataIndex ← int(query)
19           URLQ ← GetURL(queryMetaDataList
                              [queryMetaDataIndex])
20           foreach doc in docs do
21             docMetaDataIndex ← int(doc)
22             URLD ← GetURL(metaDataList
                              [docMetaDataIndex])
23             NOSD∪Q ← GetNOS(Qj in URLQuery)
                         + GetNOS(Di in URLDoc)
24             Emit(URLQ-URLD, NOSD∪Q)
```

Consequently, the pair [N, [URL_{Q1}@4, URL_{Q2}@5, URL_{Q3}@5]] is replaced by [N, [0, 1, 2]]. When the data input is large, the indexes with our metadata is much more lightweight than those without that. Our experiments in Sect. "Empirical Experiments and Evaluations" show how much lightweight they are and the efficiency they devote in speeding up the searching process.

Towards Other Similarity Queries

Our presentation so far is for query by example, which looks for similar objects when given either a query or a set of queries. However, our proposed approach would be adapted to other fundamental kinds of similarity queries as follows.

- **Pairwise similarity query.** In this context, we would like to compute the similarity between one object and the others in the dataset. In other words, if we have *N* objects in the dataset, we need to calculate how similar one object is with (*N*-1) other objects. To do that, we interfere the algorithm 3: MAP-2, shown in Fig. 6, or the algorithm 6: MAP-2 with metadata, shown in Fig. 13. In this case, we replace the set of query objects with the same set of data objects. Moreover, we need to add the condition in step 6 of the algorithm 3 and in step 12 of the algorithm 6, so that the examining pair must not be the same.
- **Range query.** When given a threshold ϵ, we would like to retrieve only those objects whose similarity scores are



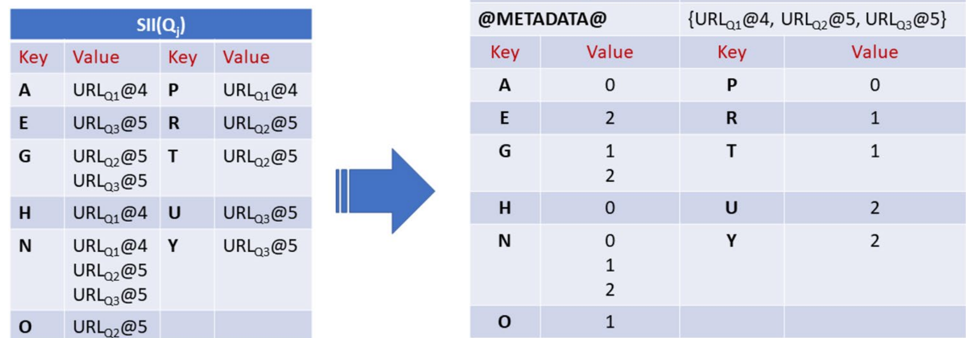
SII(D_p)

Key	Value	Key	Value
A	$URL_{D1}@8$ $URL_{D2}@6$ $URL_{D3}@7$	O	$URL_{D2}@6$
D	$URL_{1}@8$	P	$URL_{D3}@7$
E	$URL_{D3}@7$	R	$URL_{D2}@6$
G	$URL_{D3}@7$	S	$URL_{D1}@8$ $URL_{D2}@6$
H	$URL_{D1}@8$ $URL_{D3}@7$	T	$URL_{D1}@8$
I	$URL_{D1}@8$ $URL_{D2}@6$	U	$URL_{D1}@8$ $URL_{D3}@7$
K	$URL_{2}@6$	Y	$URL_{D3}@7$
N	$URL_{D1}@8$ $URL_{D3}@7$		

SIIWMD(D_p)

MetaData Key		MetaData Value	
@METADATA@		{$URL_{D1}@8$, $URL_{D2}@6$, $URL_{D3}@7$}	
Key	Value	Key	Value
A	0 1 2	O	1
D	0	P	2
E	2	R	1
G	2	S	0 1
H	0 2	T	0
I	0 1	U	0 2
K	1	Y	2
N	0 2		

Fig. 14 Metadata of dataset [8]



SII(Q_j)

Key	Value	Key	Value
A	$URL_{Q1}@4$	P	$URL_{Q1}@4$
E	$URL_{Q3}@5$	R	$URL_{Q2}@5$
G	$URL_{Q2}@5$ $URL_{Q3}@5$	T	$URL_{Q2}@5$
H	$URL_{Q1}@4$	U	$URL_{Q3}@5$
N	$URL_{Q1}@4$ $URL_{Q2}@5$ $URL_{Q3}@5$	Y	$URL_{Q3}@5$
O	$URL_{Q2}@5$		

SIIWMD(Q_j)

MetaData Key		MetaData Value	
@METADATA@		{$URL_{Q1}@4$, $URL_{Q2}@5$, $URL_{Q3}@5$}	
Key	Value	Key	Value
A	0	P	0
E	2	R	1
G	1 2	T	1
H	0	U	2
N	0 1 2	Y	2
O	1		

Fig. 15 Metadata of query set [8]

greater or equal to that ϵ. To do that, we interfere the algorithm 4: REDUCE-2, shown in Fig. 7. After either step 11 or step 16, we check whether the similarity score is greater or equal to the given ϵ or not. If yes, we let reducers output that pair. Furthermore, if there are different similarity thresholds for different queries, we need to check the similarity score against the corresponding threshold.

- **K-nearest neighbor query.** When given a k parameter, we would like to retrieve only k objects that are the most similar as a query object. To do that, we need to aggregate similarity scores after either step 11 or step 16 of the algorithm 4: REDUCE-2, shown in Fig. 7 and perform ranking based on those similarity scores. After that, we choose to output k pairs that have highest similarity scores with regard to their query objects. In case we need to process query batches, we may have different k parameters for different queries. In this scenario, we rank the similarity scores and group them by their queries. It is worth noting that our modified method here may return a super set of the real top-k result.

Empirical Experiments and Evaluations

Environmental Setting

We deploy Hadoop-based two-node cluster on a PC, in which each node has 2.00 GB RAM and 50 GB HDD. The PC has Intel® Core™ i5-4460, 3.20GHz CPU, 8.00 GB RAM, 500 GB HDD, and 64-bit operating system. Additionally, the Hadoop version is 2.7.3[2] run with its default settings. Nevertheless, we set the number of reducers to 4, which is based on four CPU cores.

[2] https://hadoop.apache.org/docs/r2.7.3/.

Table 1 Data organization

Type	Package no.	No. of files	Size range (KB)
Data	D5	5	40–102
	D100	100	15–59
	D300	300	1–32
	D500	500	1–32
	D1K	1000	1–59
Query	Q1	1	59
	Q10	10	59–59
	Q100	100	59–59
	Q1K	1000	59–59

Dataset

We employ datasets retrieved from Gutenberg Project,[3] an online data storage with over 56,000 free e-books, for our experiments. The datasets are randomly chosen from the storage and organized into different data as well as query packages, which is illustrated in Table 1. The data type serves as the data input for similarity search, while the query type is the pivot input to look for similar documents from the data type. For the data type, we organize five different data packages as D5, D100, D300, D500, and D1K with 5, 100, 300, 500, and 1000 files, respectively. As it was randomly chosen, the file size ranges from 1 to 102 KB. In the meantime, the query type is organized into four different query packages as Q1, Q10, Q100, and Q1K with 1, 10, 100, and 1000 files, respectively. In addition, the query size range is 59 KB.

Measurement

We compare the two different methods as follows:

- Sorted inverted index (SII). This method follows our proposed similarity search scheme to build sorted inverted indexes for both data sources and query batches. In addition, the method performs query processing with our pruning strategy.
- Sorted inverted index With metadata (SIIWMD). This method follows our proposed similarity search scheme to build sorted inverted indexes with metadata organization. Additionally, the method performs query processing with our pruning strategy.

[3] http://www.gutenberg.org/.

Evaluation

In our first experiment, we measure the performance of the two comparing methods, known as SIIWMD and SII, for indexing query batches. Figure 16a shows the query indexing time when the sorted inverted indexes are built for query batches. In general, the processing time of the two methods is not much different with Q10 and Q100. In fact, SII tends to have less query indexing time than SIIWMD when the number of queries sharply increases due to the fact that it does not suffer overheads for metadata organization. For example, the gap is around 4.55% with Q1K. Meanwhile, Fig. 16b indicates the query indexing size between the two comparing methods. Generally, SIIWMD generates sorted inverted indexes much lighter than SII. When the query batch size increases from Q10, Q100 to Q1K, SIIWMD saves nearly 12 times more data output than SII on the average. As a result, SIIWMD generates much more lightweight indexes than SII.

To experience the indexing building with larger dataset, we do the same experiment for data packages. As illustrated in Fig. 16c, we observe that the indexing time between SIIWMD and SII is not much different when the dataset is small as with D100, D300, and D500. Nevertheless, when the dataset size is large as with D1K, the gap is around 10.37%. In the meantime, the result from Fig. 16d keeps enforcing the fact that SIIWMD saves much more data output when building indexes than SII. On the average, SIIWMD saves nearly 6 times more data output than SII.

In terms of query processing, we do our next experiments with different query and data packages. Figure 17a shows the MapReduce performance with D5. With the small number of queries such as Q10, the query processing time of SIIWMD is nearly 33% faster than that of SII. Nevertheless, when the query size rapidly increases, the performance gap between them is much bigger. More specifically, the performance gap sharply rises as 85.7% with Q100, whereas it is 90.28% with Q1K. Consequently, SIIWMD processes query batches 6 times much faster than SII does on the average. In parallel, Fig. 17b displays the MapReduce performance with Q100 when the dataset changes. Generally, SIIWMD gives much faster query processing time than SII does. When the dataset size is small as with D100, SII processes Q100 nearly 2 times slower than SIIWMD does. In addition, the query processing time of SII sharply rises when the dataset size becomes larger. More concretely, the performance gap sharply rises as 79% with D300, whereas it is 82.5% with D500. Furthermore, the query processing time of SIIWMD slightly increases while that of SII dramatically rises when the dataset size rapidly grows. As a consequence, SIIWMD processes query batches 4 times much faster than SII does on the average.

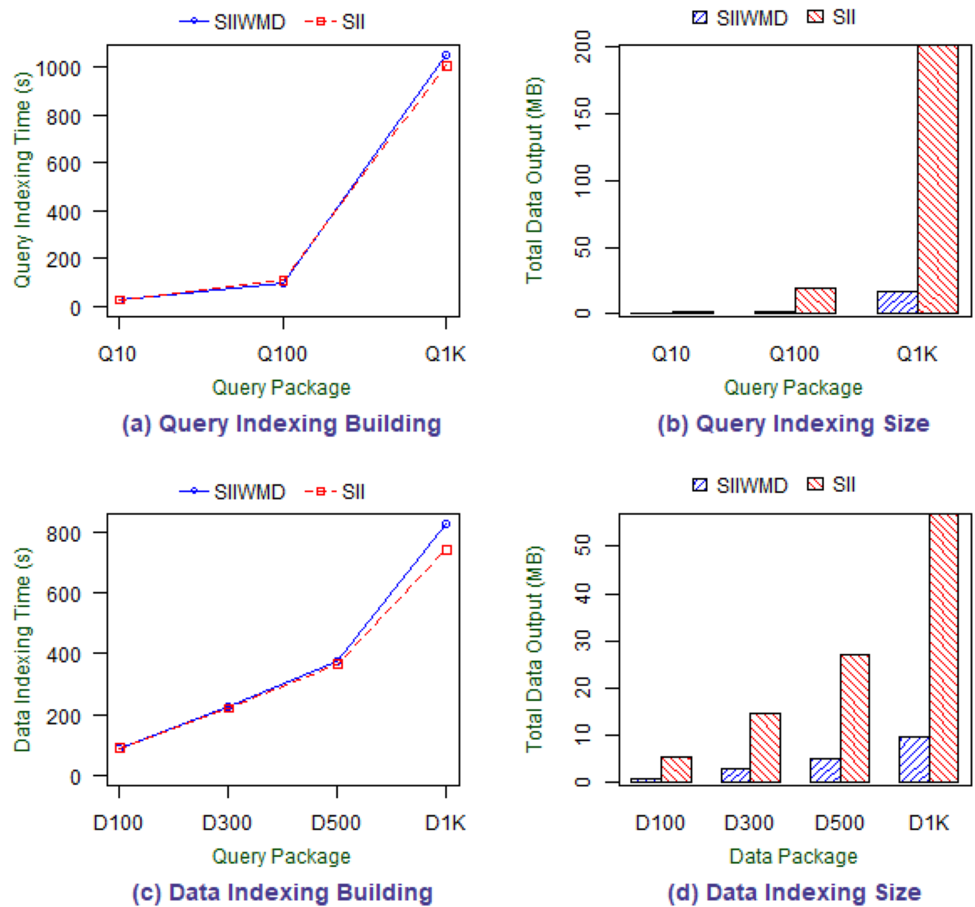


Fig. 16 Query and data indexing [8]

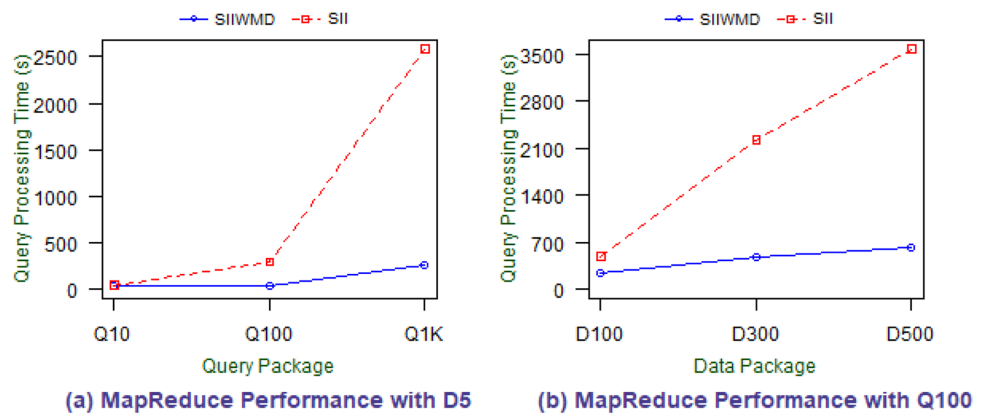


Fig. 17 Query processing [8]

Conclusion and Future Work

In this paper, we propose a general MapReduce-based similarity search scheme that not only effectively works for a single query processing but also efficiently deals with batch processing in an incremental fashion. Additionally, we build ordered inverted indexes for both datasets and query sets so that we can benefit our quick pruning strategy, which discards inessential accesses during the similarity search. Moreover, we embed metadata inside the index structures, so that we can generate much more lightweight indexes, which consequently helps reduce I/O costs as well as extra overheads caused by redundant data. By getting all to work as one, we improved the performance of our MapReduce-based similarity search while keeping the indexing size small, especially when the dataset becomes larger. In the end, the results from our empirical experiments verify the efficiency of our proposed solution. More concretely, our indexes with metadata are much lighter than baseline inverted indexes,

while the building time is not that much. Furthermore, our method saves much more processing time than the baseline method.

In our future work, we are going to apply our proposed method to the variants of similarity queries. Moreover, we plan to experience a larger dataset size as well as the cluster size for large-scale similarity processing. Last but not least, we consider the load-balancing problem whose solution further improves the overall performance of MapReduce-based similarity search.

Acknowledgements This research is funded by Department of Science and Technology, HCMC, with the project titled “Xay dung framework chuyen doi du lieu cho he thong tich hop du lieu-Develop a data conversion framework for data integration systems”.

References

1. Alabduljalil MA, Tang X, Yang T. Optimizing parallel algorithms for all pairs similarity search. In: Proceedings of the 6th ACM International Conference on web search and data mining, 2013. pp. 203–212.
2. Baraglia R, De Francisci Morales G. Lucchese C. Document similarity self-join with Mapreduce. In: 2010 IEEE International Conference on data mining, 2010. pp. 731–736.
3. Dang TK, Küng J, Wagner R. The SH-tree: a super hybrid index structure for multidimensional data. In: Proceedings of the 12th International Conference on database and expert systems applications, 2001. pp. 340–349.
4. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. J Commun ACM. 2008;51(1):107–13.
5. Gao Y, Yang K, Chen L, Zheng B, Chen G, Chen C. Metric similarity joins using MapReduce. IEEE Trans Knowl Data Eng. 2017;29(3):656–69.
6. Metwally A, Faloutsos C. V-SMART-join: a scalable MapReduce framework for all-pair similarity joins of multisets and vectors. Proc VLDB Endow. 2012;5(8):704–15.
7. Nguyen DT-T, Yong CH, Pham XQ, Loan TTK, Huh EN. An index scheme for similarity search on cloud computing using mapreduce over docker container. In: Proceedings of the 10th International Conference on ubiquitous information management and communication, 2016;60:1–6.
8. Phan TN, Dang TK. An efficient batch similarity processing with MapReduce. In: Proceedings of the 5th International Conference on future data and security engineering, 2018. pp. 158–171.
9. Phan TN, Küng J, Dang TK. An adaptive similarity search in massive datasets. In: Transactions on large-scale data- and knowledge-centered systems XXIII, Lecture Notes in computer science. 2016;9480:45–74.
10. Phan TN, Küng J, Dang TK. eHSim: an efficient hybrid similarity search with MapReduce. In: Proceedings of the 30th IEEE International Conference on advanced information networking and applications. 2016. p. 422–429, IEEE computer society.
11. Rajaraman A, Ullman JD. Chapter 3: Finding similar items. In: Mining of massive datasets. pp. 71–127 Cambridge University Press; 2011.
12. Rong C, Lu W, Wang X, Du X, Chen Y, Tung AKH. Efficient and Scalable processing of string similarity join. IEEE Trans Knowl Data Eng. 2013;25(10):2217–30.
13. Satuluri V, Parthasarathy S. Bayesian locality sensitive hashing for fast similarity search. Proc VLDB Endow. 2012;5(5):430–41.
14. Tang M, Yu Y, Aref WG, Malluhi QM, Ouzzani M. Efficient processing of Hamming-distance-based similarity-search queries over MapReduce. In: Proceedings of 18th International Conference on extending database technology, 2015. pp. 361–372.
15. Wang J, Li G, Deng D, Zhang Y, Feng J. Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search. In: 31st IEEE International Conference on data engineering, 2015. pp. 519–530
16. Xiao C, Wang W, Lin X, Yu JX, Wang G. Efficient similarity joins for near-duplicate detection. ACM Trans Database Syst. 2011;6(3):15:1–41.
17. Zadeh RB, Goel A. Dimension independent similarity computation. J Mach Learn Res. 2013;14(1):1605–26.
18. Zezula P, Amato G, Dohnal V, Batko M. Similarity search—the metric space approach. In: Series: advances in database systems, vol. 32, XVIII, 2006. 220 p., ISBN: 0-387-29146-6.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.