Efficient and Scalable Processing of String Similarity Join

Chuitian Rong, Wei Lu, Xiaoli Wang, Xiaoyong Du,
Yueguo Chen, Anthony K.H. Tung

Abstract—The string similarity join is a basic operation of many applications that need to find all string pairs from a collection given a similarity function and a user specified threshold. Recently, there has been considerable interest in designing new algorithms with the assistant of an inverted index to support efficient string similarity joins. These algorithms typically adopt a two-step filter-and-refine approach in identifying similar string pairs: (1) generating candidate pairs by traversing the inverted index; and (2) verifying the candidate pairs by computing the similarity. However, these algorithms either suffer from poor filtering power (which results in high verification cost), or incur too much computational cost to guarantee the filtering power. In this paper, we propose a multiple prefix filtering method based on different global orderings such that the number of candidate pairs can be reduced significantly. We also propose a parallel extension of the algorithm that is efficient and scalable in a MapReduce framework. We conduct extensive experiments on both centralized and Hadoop systems using both real and synthetic datasets, and the results show that our proposed approach outperforms existing approaches in both efficiency and scalability.

Index Terms—Similarity Join, Multiple Filtering, MapReduce

1 Introduction

As a fundamental data type, strings are widely used in a variety of applications, such as recording product and customer names in marketing, producing publications in academic research, publishing contents in websites. Frequently, different strings may refer to the same real-world entity due to various reasons [17]. In order to find different strings that may refer to the same entity, the string similarity join is proposed as a primitive operation that finds all pairs of strings in a given string collection, based on a string similarity function such as the Jaccard similarity [16] and a user specified threshold. Existing methods for string similarity join fall into two categories [20], [4], [28], [24]. Methods in the first category index strings using a tree structure and perform the join operation from the root to leaves by employing a set of filtering rules. The B$^+$-tree [28] and Trie-tree [24] are two such indexes. Unfortunately, the Trie-tree based indexing technique constrains itself to in-memory join operations and is inefficient for long strings, while the B$^+$-tree based indexing technique requires the whole index to be constructed in advance, and its performance suffers for short strings. Thus, they are not suitable for processing similarity joins over very large string collections. Methods in the other category extract signatures (e.g., n-grams, words) from strings, and index the strings based on their signatures using inverted indexes [20], [4]. A pair of strings that share a certain number of signatures are regarded as a candidate pair. Our solution in this paper falls into this category.

To identify all similar string pairs in a collection, the inverted index based methods follow a filter-and-refine process which first generates candidate string pairs based on their signatures, and then verifies each candidate pair by computing their similarity. As an example, we consider the titles for two publications "Functional Programming and Parallel Graph Rewriting" and "Functional Dependencies from Relational to XML". If we use a single word of each string as a signature, these two strings are considered as a candidate pair as they share one signature "Functional". Unfortunately, the number of qualified candidate string pairs will grow exponentially when the strings contain popular words. To overcome this problem, a prefix based inverted index has been proposed in [3], in which some selected words of each string are extracted as signatures. More specifically, words of each string s are sorted on the basis of a global order. The first T tokens (words) are selected as its prefix, where T only relies on $|s|$ (the number of words in s), the similarity function sim, and the user specified similarity threshold θ. It shows that for any other string t, the necessary condition of $sim(s,t) \geq \theta$ is that the prefix of s and t must have at least one token in common [4], [3], [27]. Take Jaccard similarity function as an example, if $sim(s,t) \geq \theta$, they share at least $C = max(\lceil |s| * \theta \rceil, \lceil |t| * \theta \rceil)$ tokens. That is to say they contain at most $|s| - C$ and $|t| - C$ different tokens to

- *X. Du, C. Rong and Y. Chen are with the Key Laboratory of Data Engineering and Knowledge Engineering, Ministry of Education, China, and School of Information, Renmin University of China, China. E-mail: {duyong,rct682,chenyueguo}@ruc.edu.cn.*
- *A. Tung, W. Lu and X. Wang are with the School of Computing, National University of Singapore, Singapore. E-mail:{luwei1,xiaoli,atung}@comp.nus.edu.sg.*

each other. So, when tokens in two strings are sorted by one global ordering, these two strings must share at least one common token in their first $|s|-\lceil |s|*\theta\rceil+1$ and $|t|-\lceil |t|*\theta\rceil+1$ tokens, respectively. Therefore, the tokens in the prefix of a string are extracted as its signatures. Let us consider the Example 1 and apply a prefix filtering technique to generate its candidate pairs, based on the Jaccard similarity function with $\theta = 0.8$.

Example 1: Table 1 lists 5 titles from the DBLP dataset. We then sort the words of each title (after removing stop words) in alphabetical order, and select the first $(|s|-\lceil |s|*\theta\rceil+1)$ words (prefixes) as the signatures, which are highlighted in bold in Table 2. (s_2, s_3) and (s_2, s_5) constitute the final result since (s_2, s_3) and (s_2, s_5) share the signature "Functional" and "Graph", respectively.

In the above example, the number of candidate pairs is reduced to two with the application of the prefix filtering technique on the basis of alphabetical order. As expected, sorting based on a different global ordering yields different prefixes for each string. That in turn may result in different candidate pairs for the different signatures of the strings.

Example 2: We sort the words of each string in Table 1 based on another order, and highlight its prefixes in bold as shown in Table 3. Obviously, the candidate pair based on this global ordering is (s_1, s_2), as they share a word of "Parallel".

By applying different global orderings to sort the words of strings in a given collection, we can derive multiple sets of candidate pairs. The string pairs whose similarity is above the predefined threshold always qualify as candidate pairs as they must share at least one common prefix token under any global ordering. In this way, string pairs lying in all candidate sets constitute the final candidate pairs while the remaining pairs can be safely pruned off. In the above examples, when applying two different global orderings, the final candidate pair set is empty since there is no overlap in the two candidate sets. Compared to using a single global ordering, the number of candidate pairs can be significantly reduced to lower the verification cost. In contrast to existing methods which attempt to find a cost-efficient filtering method using one global ordering, we aim to answer string similarity joins by using cost-efficient multiple prefix filtering.

However, there are two challenges that need to be addressed: (1) Selection of a set of global orderings such that the number of candidate pairs can be reduced as much as possible. (2) Derivation of the final candidate pairs.

Figure 1 shows an intuitive and simple way to apply multiple global orderings to derive the final candidate pairs. It first sequentially applies each global ordering to find individual candidate pairs, and then compute the overlap among these partial results. This entails building and traversing the inverted index for each single global ordering method, which however incurs high overall overhead. To keep the overhead low, we propose to apply multiple global orderings and filtering in an effective pipelining manner. Unlike the existing approach of building and traversing multiple inverted indexes based on different global orderings, we use only one inverted index to support multiple filterings. Our approach is simpler, but yet more efficient and scalable in both centralized and parallel environments.

In this paper, we make the following contributions:

- We propose a multiple prefix filtering technique that applies the filtering using different global orderings in a pipelining manner. The objective is to reduce the number of candidate string pairs as efficiently and effectively as possible. This will reduce the cost of the more expensive verifications in the refinement step.
- We provide a detailed analysis of our proposed method. We propose a cost model to determine the number of multiple global orderings, and optimizations for the selection and application of global orderings.
- We parallelize our algorithm for a MapReduce framework.
- We conduct extensive experiments in both centralized and Hadoop-like platforms. The experiments demonstrate that our proposed approach outperforms other existing methods by a wide margin in terms of efficiency and scalability.

The rest of the paper is organized as follows: The related works are given in Section 2. Section 3 presents the problem definitions and preliminaries. Section 4 describes our proposed multiple prefix filtering method. Section 5 presents how to extend our proposed method in the *MapReduce* framework. Experimental evaluation is given in Section 6. Section 7 concludes the paper.

2 Related Work

In this section, we shall review some recent works proposed for centralized systems and Hadoop-like systems.

String similarity join is a primitive operation in many applications such as merge-purge [13], record linkage [10], [25], object matching [21], reference reconciliation[7], deduplication [19], [2] and approximate string join [11]. In order to avoid verifying every pair of strings in the dataset and improve performance, string similarity join typically consists of two phases: candidate generation and verification [9], [17]. In the candidate generation phase, the signature assignment process or blocking process is invoked to group the candidates into groups by using either an approximate and exact approach, depending on whether some amount of error could be tolerated

TABLE 1
String of Records

id	values
S_1	Parallel Relational Database Systems
S_2	Functional Programming and Parallel Graph Rewriting
S_3	Proof Pearl From Concrete to Functional Unparsing
S_4	Functional Dependencies from Relational to XML
S_5	Inequational Deduction as Term Graph Rewriting

TABLE 2
Sorted by Alphabetical Ordering

S_1	**Database** Parallel Relational Systems
S_2	**Functional Graph** Parallel Programming Rewriting
S_3	**Concrete Functional** Pearl Proof Unparsing
S_4	**Dependencies** Functional Relational XML
S_5	**Deduction Graph** Inequational Rewriting Term

TABLE 3
Sorted by Another Global Ordering

S_1	**Parallel** Database Relational Systems
S_2	**Functional Parallel** Graph Rewriting Programming
S_3	**Unparsing Pearl** Proof Concrete Functional
S_4	**Dependencies** Functional XML Relational
S_5	**Inequational Deduction** Graph Rewriting Term

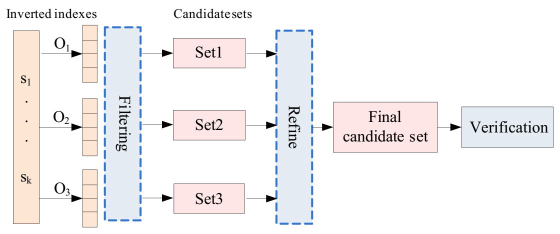


Fig. 1. Similarity Join Processing

or not. Since we aim to provide exact answers, we will focus on the exact approaches. Recent works that provide exact answers are typically built on top of some traditional indexing methods, such as tree based and inverted index based. In [5], the Trie-tree based approach was proposed for edit similarity search, where an in-memory Trie-tree is built to support edit similarity search by incrementally probing it. The edit similarity join method based on the Trie-tree was proposed in [24], in which sub-trie pruning techniques are applied. In [28], a B$^+$-tree based method was proposed to support edit similarity queries. It transforms the strings into digits and indexes them in the B$^+$-tree. However, these algorithms are constrained to in-memory processing, not efficient and scalable for processing large scale dataset.

The methods making use of the inverted index are based on the fact that similar strings share common parts and consequently they transform the similarity constraints into set overlap constraints. Based on the property of set overlap [4], the prefix filtering was proposed to prune false positives [4], [3], [27], [11]. In these methods, the partial result of the candidate generation phase is a superset of the final result. The *AllPairs* method proposed in [3] builds the inverted index for prefix tokens and each string pair in the same inverted list are considered as candidates. This method can reduce the false positives significantly compared to the method that indexes all tokens of each strings [20]. In order to prune false positives more aggressively, the *PPJoin* method applies the position information of the prefix tokens of the string. Based on the *PPJoin*, the *PPJoin+* uses the position information of suffix tokens to prune false positives further [27]. As these methods need to merge the inverted lists during the candidate generation phase, some optimization techniques for the inverted list merging were introduced in [15], [27]. The exact computation method proposed in [1] is based on the pigeon hole principle. It transforms similarity constraints into Hamming distance constraints and transforms each record into a binary vector. The binary vector is divided into partitions and then hashed into signatures, and the strings that produce the same signatures are considered as candidate pairs. However, the signature scheme is time consuming and introduces unnecessary false positives.

A common drawback of the above proposals is that they cannot be easily parallelized to run efficiently on a MapReduce framework. In the MapReduce framework, the global information about the whole dataset cannot be accessed easily, and therefore, the filtering strategies used in a centralized system are not effective in this share-nothing computing environment. In a demonstration paper [23], a framework was briefly introduced, without providing much details. In the recently work [22], two methods for similarity join on MapReduce are proposed. One is *RIDPairsImproved*, in which each prefix token of the string is considered as its signature (key) in the Map procedure, the candidate generation phase. Then, the strings that with the same signatures will be shuffled into one group for further verification. In the verification process, filtering methods are applied to avoid similarity computation for as many false positives as possible. The other method is *RIDPairsPPJoin*, which has the same implementation of Map. In Reduce, *RIDPairsPPJoin* builds inverted index for each group of strings to accelerate processing. However, the filtering method needs to scan each string pair more than one time to compute the similarity upper bound for pruning purpose. This incurs high overhead in a distributed environment.

In this paper, we propose an efficient and MapReduce friendly multiple prefix filtering approach based on different global orderings. In the Map phase, we apply one global ordering to generate signatures for the strings and apply other global orderings to get different prefix token sets which are appended to the string. In the $Reduce$ phase, for each string pair their prefix token sets obtained by the same global ordering are checked and pruned if they are obviously not candidates. As the size of the prefix token set is shorter than the string and the checking process is applied in a pipelining manner, the verification process is therefore very efficient.

3 Problem Definition and PRELIMINARIES

3.1 Definitions

A string s is considered as a set of tokens, each of which can be either a word or an n-gram (a substring of s with length n). For example, the tokens of

TABLE 4
Symbols and Definitions

Symbols	Definition
Σ	the alphabet
s	a string with a sequence of characters in Σ
$\mathbb{S}$	collection of strings
$\lvert\cdot\rvert$	the element number of a set
t	a token s
T_s	set of tokens for string s
V_s	T_s transformed to Vector Space Model
T_s^p	the first p tokens of string s
U	a token collection $U = \{\cup_{s \in \mathbb{S}} T_s\}$
θ	pre-assigned threshold
$sim(s_i, s_j)$	similarity between s_i and s_j
$\mathcal{O}$	global ordering
$\mathcal{O}_g$	a group of global orderings
$T_s^p(O)$	T_s^p under $\mathcal{O}$

string of s = "Parallel Relational Database Systems"are {Parallel, Relational, Database, Systems}.

Definition 1: (String Similarity Join) Given a set of strings $\mathbb{S}$, and a join threshold θ, string similarity join finds all string pairs (s_i, s_j) in $\mathbb{S}$, such that $sim(s_i, s_j) \geq \theta$.

Table 4 lists the symbols and their definitions that will be used throughout this paper.

3.2 Similarity Measures

A similarity function measures how similar two strings are and returns a value in [0,1]. Typically, the larger the value is, the more similar two strings are. In this paper, we utilize three widely used similarity functions, namely Dice [18], Jaccard [16], and Cosine [26], whose computation problem can be reduced to set overlap problem [3]. They are based on the fact that similar strings share common components. Clearly, the similarity between two strings is zero if they do not have any token in common. In other words, we only verify string pairs with at least one common token. For the sake of better understanding, the similarity measures and their definitions are summarized in Table 5. Unless otherwise specified, we use Jaccard as the default function, i.e., $sim(s_i, s_j) = sim_{jaccard}(s_i, s_j)$.

3.3 Prefix Filtering

Prefix filtering technique is commonly used in the refinement step to further prune false positives of candidate pairs that share a certain number of common tokens. In [3], [4], the methods sort the tokens of each string based on some global ordering $\mathcal{O}$ (e.g., an alphabetical order or a term frequency order), select a certain number of its first tokens as the prefix, and use the tokens in the prefix[1] as its signatures. They prove that the necessary condition for every two strings s_i and s_j to be a candidate pair is that the prefixes of s_i and s_j must have at least one token in common. The number of tokens in the prefix for each string can be computed and the computation formulae are shown in Table 6. Essentially, they rely on the similarity function, join threshold, and length of the string.

1. When there is no ambiguity, we will simply refer token prefixes as prefixes.

TABLE 5
Similarity Function and Its Definition

Similarity Function	Definition
$sim_{dice}(s_i, s_j)$	$\frac{2\times\lvert T_{s_i} \cap T_{s_j}\rvert}{\lvert T_{s_i}\rvert + \lvert T_{s_j}\rvert}$
$sim_{jaccard}(s_i, s_j)$	$\frac{\lvert T_{s_i} \cap T_{s_j}\rvert}{\lvert T_{s_i} \cup T_{s_j}\rvert}$
$sim_{cosine}(s_i, s_j)$	$\frac{V_{s_i} \cdot V_{s_j}}{\sqrt{\lvert V_{s_i}\rvert \times \lvert V_{s_j}\rvert}}$

TABLE 6
Prefix Length

Similarity Function	Prefix Length
Dice	$\lvert T_{s_1}\rvert - \lceil \lvert T_{s_1}\rvert * \theta \rceil + 1$
Jaccard	$\lvert T_{s_1}\rvert - \lceil \lvert T_{s_1}\rvert * \theta \rceil + 1$
Cosine	$\lvert T_{s_1}\rvert - \lceil \lvert T_{s_1}\rvert * \theta^2 \rceil + 1$

By applying the prefix filtering technique, candidate pairs with no overlap in their corresponding prefixes can be safely pruned. In this manner, the number of candidate pairs can be significantly reduced since popular words or frequent tokens can be set to the end of the global ordering so that they are not probable of lying in the prefixes.

4 Similarity Join Processing

In this section, we first introduce the similarity join method by using a single global ordering scheme, and then extend it for multiple global ordering schemes, finally conduct a detailed complexity analysis.

4.1 Single Global Ordering

In general, the similarity join using the prefix filtering technique based on a single global ordering consists of three phases.

1) Index Construction: In this phase, the prefix tokens of each record are extracted under one global ordering and indexed in an inverted index. Figure 2 shows an inverted index built for the prefix tokens of strings shown in Table 1 using the alphabetical ordering.
2) Candidate Pairs Generation: The strings in the dataset are processed sequentially. For each string $s \in \mathbb{S}$, the prefix tokens T_s^p are derived. For each token $t \in T_s^p$, the corresponding inverted list is scanned to get all the candidates for the current string. Take the string s_5 for example. The prefixes of s_5 which are presented in Table 2 are $T_{s_5}^p$={"Deduction","Graph"}. The inverted lists for "Deduction", "Graph" are therefore scanned, and s_2 is identified as the similarity candidate of s_5. Thus, (s_2, s_5) constitutes a candidate pair. The optimization of scanning the inverted index is well studied, and reported in various work such as [20], [27].

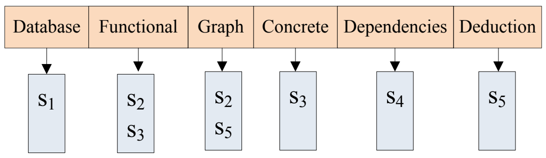


Fig. 2. Inverted Index for Prefix Tokens

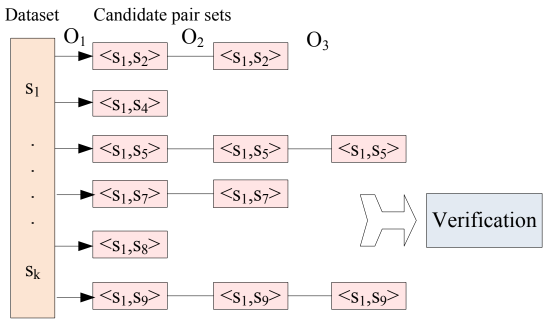


Fig. 3. Pipelining Operations in Multiple Prefix Filtering

3) Verification: We compute the similarity between two strings in each candidate pair one by one and output them when the similarity is no less than θ.

4.2 Multiple Global Orderings

By applying the single global ordering method, the candidate pairs can be significantly reduced comparing with the naive approach, in which every two strings that share common tokens are considered as a candidate pair. However, for large scale datasets, the number of candidate pairs is still very large and needs to be substantially reduced. Therefore, in this section, we propose a more effective alternative by making use of multiple global orderings to reduce the number of candidate pairs. A straightforward solution of implementing such an approach is to repeat the approach of using single global ordering for each ordering, and derive the overlap among these sets of candidate pairs. However, the cost of this process for candidate pair generation is very high. To solve such a problem, we propose a new join method called *MGJoin*.

Figure 3 illustrates the workflow of the *MGJoin* similarity join processing. In the figure, we have used the string s_1 and three global orderings as our example for illustration. At the first phase, we generate a set of global orderings $\mathcal{O}_g = \{\mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3\}$(the details of this generation will be explained in the latter section). We do not generate each set of candidate pairs for each $\mathcal{O}_i \in \mathcal{O}_g$ separately, instead, we select $\mathcal{O}_1$ as the basis to build inverted index for prefix tokens. Further, during the candidate pair generation phase, for each string, as we generate its candidates based on its prefixes, we do the pipeline processing to prune the false positive of its candidates in advance. For string s_1, its candidate set size $|C_{num}(s_1)|$ is six when applying $\mathcal{O}_1$. Based on $C_{num}(s_1)$, $\mathcal{O}_2$ and $\mathcal{O}_3$ are applied in pipelining. We can found that $|C_{num}(s_1)|$ become smaller in each step. More specifically, given a string and its similarity candidate, we check if they constitute a candidate pair by directly comparing their prefixes (without accessing the inverted index) that have been produced by another global ordering. We will continuously check the candidate pair by comparing their prefixes that have been produced by any global ordering until they are pruned off or all their prefixes are checked.

Algorithm 1: $MGJoin(\mathbb{S}, \theta, K)$

```
input : $\mathbb{S}$: the data set, each string is tokenized and
        sort by one global ordering $\mathcal{O}$
        $\theta$: the similarity threshold
        $K$: the number of prefix filtering to apply
output: All pairs of strings $< x, y >$, if $sim(x, y) \geq \theta$
1  $Results \leftarrow \emptyset$ ;
2  foreach $s_x$ in $\mathbb{S}$ do
3      Candidates.clear();
4      $T^p_{s_x} \leftarrow$ get prefix tokens of $s_x$;
5      $T^{mp}_{s_x} \leftarrow$ get multiple prefix tokens of $s_x$;
6      foreach $token \in T^p_{s_x}$ do
7          //scan inverted list of $token$ $I_{token}$
8          foreach $s_y \in I_{token}$ do
9              if $|s_y| > |s_x| \times \theta$ then
10                 res=$MultiPrefixFiltering(T^{mp}_{s_x}, T^{mp}_{s_y}, K)$;
11                 if $res == true$ then
12                     $Candidates[y] += 1$;
13     $Verification(s_x, Candidates)$;
14 return $Results$;
```

Algorithm 2: $MultiPrefixFiltering(T^{mp}_{s_x}, T^{mp}_{s_y}, K)$

```
input : $T^{mp}_{s_x}$: multiple prefix tokens for $s_x$
        $T^{mp}_{s_y}$: multiple prefix tokens for $s_y$
        $K$: the number of prefix filtering applied
output: true or false
1  for $i \leftarrow 0$ to $K$ do
2      if $T^{mp}_{s_x}[i] \cap T^{mp}_{s_y}[i] == \emptyset$ then
3          return false;
4  return true;
```

Algorithm 1 provides the outline of *MGJoin*. Each string in the dataset is tokenized into a token set and sorted based on one global ordering. The other prefix tokens are derived during the canonicalization process. The input strings are sorted based on their lengths in order to avoid unnecessary operations. The sorted strings are processed sequentially in two phases, candidate generation phase and the verification phase. For each string, its prefix token set $T^p_{s_x}$ and other prefix token sets $T^{mp}_{s_x}$ will be obtained (lines 4-5). In this algorithm, the ascending order of term frequency is applied to produce the prefix tokens $T^p_{s_x}$. For each prefix token in $T^p_{s_x}$, its inverted index I_{token} will be scanned using length filtering method and then multiple prefix filtering (lines 8-9). The string pairs that satisfy all the filtering conditions are considered as candidate pairs. In order to avoid duplicate verifications for the same two strings, each inverted index is scanned sequentially and the candidates are counted for their number of occurrences (line 11). After which, the pairwise verification is invoked.

The multiple prefix filtering algorithm is outlined in Algorithm 2. Its input is each string's multiple prefix token sets that have been produced by applying different global orderings on the token set of the string. The K is the number of prefix filters applied. If two strings don't share any prefix token under some global ordering, they cannot be similar (line 2). If two strings are similar, they must have at least one common prefix token under any global ordering.

4.3 Analysis of Global Orderings

MGJoin algorithm applies multiple global orderings in the pipelining manner for improving the efficiency of string similarity join. According to the property of the prefix filtering technique [3], [4], the correctness of *MGJoin* algorithm will be always guaranteed no matter which global orderings are selected. In this section, we give a detailed analysis on the global orderings. For ease of exposition, let $\mathcal{O}_g$ be a set of global orderings; C_{num} be the number of candidate pairs by applying the global orderings in $\mathcal{O}_g$. The analysis is based on the assumption: all global orderings are selected randomly and each of them has the same pruning power.

Theorem 1: Given a group of orderings $\mathcal{O}_g$, the final number of candidate pairs C_{num} is independent on the execution order of the global orderings in $\mathcal{O}_g$.

Proof: Let $\mathcal{O}_g = \{\mathcal{O}_1, \mathcal{O}_2, ..., \mathcal{O}_k\}$. If (s_i, s_j) is a candidate pair certified by applying $\mathcal{O}_g$. When applying $\mathcal{O}_m$ $(1<m \leq k)$, their prefix token sets are $T^p_{s_i}(\mathcal{O}_m)$ and $T^p_{s_j}(\mathcal{O}_m)$, respectively. In pipelining checking process, if there exits $T^p_{s_i}(\mathcal{O}_m) \cap T^p_{s_j}(\mathcal{O}_m) = \emptyset$, the candidate pair (s_i, s_j) will be pruned off. Therefore, C_{num} is decided by the effect of given global orderings in $\mathcal{O}_g$, but independent on the execution order of $\mathcal{O}_m \in \mathcal{O}_g$. □

Lemma 1: $\forall s_i, s_j \in \mathbb{S}$. If $sim(s_i, s_j) < \theta$, the probability of the pair (s_i, s_j) to be pruned off increases as more global orderings are applied.

Proof: For the pair (s_i, s_j), the probability of this pair to be pruned off is defined as $p(0<p<1)$. p can be computed by the occurrence probability of $T^p_{s_i} \cap T^p_{s_j} = \emptyset$ when applying all kinds of global orderings on U. If $sim(s_i, s_j) < \theta$, the occurrence probability of $T^p_{s_i} \cap T^p_{s_j} = \emptyset$ becomes high when more global orderings are applied. Therefore, if $sim(s_i, s_j) < \theta$, the pair (s_i, s_j) is more likely to be pruned off as more global orderings are applied. □

Theorem 2: The candidate pair number C_{num} decreases as more global ordering applied.

Proof: $\forall s_i, s_j \in \mathbb{S}$. If $sim(s_i, s_j) \geq \theta$, the pair (s_i, s_j) always share at least one common prefix token under any global ordering and being reserved as candidate pair. So, we have $C_{num} \geq N_{res}$. If $sim(s_i, s_j) < \theta$, the pair (s_i, s_j) will be pruned off with the probability p, according to Lemma 1. Let the number of all possible string pairs in $\mathbb{S}$ is N. After applying k global ordering, $C_{num} = N_{res} + (N - N_{res}) \times p^k$. So, as more global orderings are applied, the C_{num} decreases and will be more closer to N_{res}. □

According to Theorem 2, in order to reduce the number of candidate pairs as many as possible, we are desired to employ a large number of global orderings. Although the number of candidate pairs can be minimized by employing a maximum number of global orderings, the overhead of candidate generation applying multiple global orderings also increases. The need arises to carefully consider the trade-off between the benefit of reducing the candidate pair number and the overhead of applying more global orderings.

As the similarity join follows a two-step filter-and-refine framework, the final total time cost is decided by not only the verification cost in the refine phase but also the filtering cost in the candidate generation phase. Let $T_{overall}$ be the overall execution time of similarity join, T_c be the time to generate the candidate pairs by applying the global orderings in $\mathcal{O}_g$ including inverted index construction, inverted list merging and multiple prefix filtering, T_v be the verification time for candidate pairs. As the group of global ordering $\mathcal{O}_g$ is determined before similarity join, the time on global ordering selection is not taken into consideration.

Let N be the number of all pairs in dataset $\mathbb{S}$, and N_{res} be the number of results. The number of initial false positives is defined as F, $F = N - N_{res}$. Let the time cost on processing one candidate pair in the candidate generation phase and the verification phase are t_c and t_v, respectively. Given a group of global ordering $\mathcal{O}_g$, let $T_{overall}(k)$ be the overall time cost when applying k global orderings. Then, we have $T_{overall}(k) = T_c(k) + T_v(k)$.

$$T_{overall}(k) = t_c \sum_{i=1}^{k} (N_{res} + F \times p^{k-1}) + t_v \times (N_{res} + F \times p^k)$$

Firstly, as the k increases the false positives are pruned off vastly. As a result, $T_{overall}(k)$ decreases drastically(see Figure 6). But, as more global orderings are applied $N_{res} + F \times p^k$ will be close to N_{res} gradually, the decrease trend of false positives become slower. That is to say, T_v does not change strikingly when k increases. But, the time cost T_c increases with k. So, $T_{overall}(k)$ will increase. Based on above analysis, the cost model about the optimal number selection of global orderings can be given as below.

Theorem 3: The precondition to apply one more global ordering is:

$$T_{overall}(k) - T_{overall}(k+1) > 0$$

Algorithm 3: $TokenPermutation(U, N)$

```
input :
        U: the distinct token list of dataset
        N: the times to exchange the position of token
output:
        newTokenList: the new token list generated by
    applying permutation
1  $size \leftarrow Sizeof(U)$;
2  for $i = 0 \rightarrow N$ do
3      $position_1 = random()\%size$;
4      $position_2 = random()\%size$;
5      Permutation($U, position_1, position_2$);
```

4.4 Selection of Global Orderings

Now we consider how to select global orderings to enhance the efficiency of the *MGJoin*. Due to the pipelining execution process of the multiple prefix filters, the efficiency may be affected by (1) the kind of global orderings: what kind of global ordering we apply; (2) the execution order of global orderings: how to arrange a group of global orderings in pipelining execution. Generally, the objective of selecting a global ordering is to reduce the number of candidate pairs as many as possible. For a given dataset, the number of global orderings is the number of all possible sorted token lists generated from token universe. In this section, we proposed three strategies for global ordering selection. All these three strategies are based on permutations in the token universe.

4.4.1 Random Selection

Given the token universe U of $\mathbb{S}$, the sequence of tokens in U is their first occurrence in the dataset. We can derive one new global ordering by randomly permutating the positions of tokens in U, as shown in Algorithm 3. Since each prefix filter is based on a different global ordering, we can derive the other global orderings using the above method. Then, one group of global orderings can be derived by random selection and defined as $\mathcal{O}_{Random}$.

$$\mathcal{O}_{Random} = \{\mathcal{O}_1, \mathcal{O}_2, ..., \mathcal{O}_k\}$$

4.4.2 Random Selection with Reverse Ordering

Let CP_i be the number of candidate pairs after employing the i^{th} global ordering $\mathcal{O}_i$ separately. As we utilize the pipelining method to apply multiple global orderings, only the candidate pairs that exist in CP_i would be checked if they share common prefix tokens when applying the next global ordering $\mathcal{O}_{i+1}$. Only the candidate pairs that occur in $CP_i \cap CP_{i+1}$ will be reserved as candidates. If the candidate pair set CP_i and CP_{i+1} are vastly different, the false positives would be pruned off easily. As described above, we expect to select two global orderings such that $|CP_i - CP_{i+1}| + |CP_{i+1} - CP_i|$ is maximized. If $\mathcal{O}_i$ and $\mathcal{O}_{i+1}$ are vastly different, the prefix tokens in one string can be different. Based on this assumption, we utilize the *Footrule Distance* [12], [14] as our basic principle for global ordering selection.

Definition 2: (Footrule Distance) Let $\mathcal{O}_i$ and $\mathcal{O}_j$ be two rankings of U. The footrule distance between the two rankings is defined as:

$$F(\mathcal{O}_i, \mathcal{O}_j) = \sum_{m=1}^{|U|} |\mathcal{O}_i(t_m) - \mathcal{O}_j(t_m)|$$

Footrule Distance depicts the sum of the ranking position gap for each token $\in U$ over two global orderings, $\mathcal{O}_i(t_m)$ is the rank of token t_m under $\mathcal{O}_i$. In particular, for two global orderings $\mathcal{O}_i$ and $\mathcal{O}_j$, $F(\mathcal{O}_i, \mathcal{O}_j)$ obtains its maximum value when $\mathcal{O}_i$ and $\mathcal{O}_j$ are ordered in a reverse way [12], [14].

Based on this, given a token set of a string, if we apply a global ordering $\mathcal{O}_1$ first, and then apply its reverse ordering $\mathcal{O}_1^{-1}$ (in Definition 3) as $\mathcal{O}_2$, the prefix token set selected for this string will be changed most significantly. In this way, the candidate pairs that are separately produced by these two global orderings are significantly different. In theory, we should select the subsequent global orderings to make each selection of prefix tokens can vary most significantly from previous selections. For example, we select the subsequent ordering $\mathcal{O}_3$ to make that $F(\mathcal{O}_1, \mathcal{O}_3)$ and $F(\mathcal{O}_2, \mathcal{O}_3)$ can obtain their maximum values. However, according to [8], the time complexity to compute the optimal global orderings is in *NP-Hard* even when only deciding the selection to $\mathcal{O}_4$.

Definition 3: (Reverse Ordering) Given a global ordering $\mathcal{O}$, its reverse ordering $\mathcal{O}^{-1}$ is defined as: $\forall t \in U$, if t is ranked at the i^{th} position, then t is ranked as the $(|U| - i + 1)^{th}$ position.

Based on above analysis, we propose an approximate strategy for global ordering selection by randomly selecting a global ordering and its reverse as a pair, which assure the $Footrule\ Distance$ of each pair of global orderings get its maximum value. In summary, if we expect to generate k global orderings, we first generate $\lceil k/2 \rceil$ global orderings as in $Random$ method, and the remainder are their reverse orderings and applied consecutively with their corresponding orders. With this, the group of global orderings is defined as:

$$\mathcal{O}_{Random\ with\ Reverse} = \{\mathcal{O}_1, \mathcal{O}_1^{-1}, \mathcal{O}_2, \mathcal{O}_2^{-1}, ...\}$$

4.4.3 TF as the First Global Ordering

In pipelining execution of multiple prefix filters, lower overhead will take in latter phases if more false positives are pruned off in earlier phases. Due to the property of token frequency ordering, in which tokens with low frequency are often selected as the prefix of each string, the candidate pairs can approximately be minimized [3], [4]. Therefore, we apply the token frequency ordering as the first global ordering followed by its reverse ordering. Therefore, if we expect to generate k global orderings, the first two global orderings are token frequency ordering and its reverse ordering. Additional $\lceil k/2-1 \rceil$ global orderings will be randomly generated, and the remainder are their reverse orderings. One group of global orderings derived by this strategy is defined as $\mathcal{O}_{TF\ as\ the\ 1^{st}}$.

$$\mathcal{O}_{TF\ as\ the\ 1^{st}} = \{\mathcal{O}_{tf}, \mathcal{O}_{tf}^{-1}, \mathcal{O}_2, \mathcal{O}_2^{-1}, ...\}$$

4.4.4 Discussion on other methods

The above three methods for global ordering selection are independent on the dataset. Once the token universe U is obtained, other global orderings can be derived by applying random permutations on U without scanning the dataset.

In this paper, we also consider other methods based on statistical information, such as the co-occurrence probability. Given a global ordering $\mathcal{O}_1$, the prefix token set $\bigcup T_{s_i}^p(\mathcal{O}_1)(s_i \in \mathbb{S})$ can be derived by canonicalization each string in the dataset $\mathbb{S}$. If we want to get another global ordering $\mathcal{O}_2$, under which the prefix token set is defined as $\bigcup T_{s_i}^p(\mathcal{O}_2)(s_i \in \mathbb{S})$. In order to prune more false positives, we want the tokens in $\bigcup T_{s_i}^p(\mathcal{O}_2)$ has low co-occurrence probability with that in $\bigcup T_{s_i}^p(\mathcal{O}_1)$. Comparing with the above three methods, the methods that based on statistical information will incur the dataset scanning and co-occurrence probability calculation to determine a new global ordering. This process will take much more time than similarity join itself.

5 Handling Similarity Join in MapReduce Framework

Recently, there has been a considerable interest in extending existing methods to support string similarity join using MapReduce [6], [22]. In this section, we present how to extend our *MGJoin* approach in MapReduce framework.

The execution process of a typical MapReduce job consists of two phases: Map and $Reduce$. In MapReduce, the data set is partitioned into a number of input splits based on a fixed size and distributed over data nodes. Each Map in a data node reads the key/value pairs from the local split and shuffles newkey/list (values) to reducers. Note that key/value pairs with the same key will be shuffled to the same reducer and these pairs are sorted based on the keys. To apply *MGJoin* to the MapReduce framework, the main challenge is how to assign keys to candidate string pairs. As a candidate pair of strings must have one common prefix token under any global ordering, we choose prefix tokens of a string as its keys and the string content as the value so that strings with the same prefix token can be shuffled to the same reducer.

This proposed solution has three MapReduce procedures. The first procedure is used to generate token universe and the term frequency order. The second one is to canonicalize the dataset according to a designated global ordering and balance the dataset on all data nodes according to the lengths of strings. The last one is to implement the *MGJoin* process, whose algorithm is given in Algorithm 5. The data flow in the *MGJoin* of an example is illustrated by Figure 4, where only two global orderings are applied for concise description.

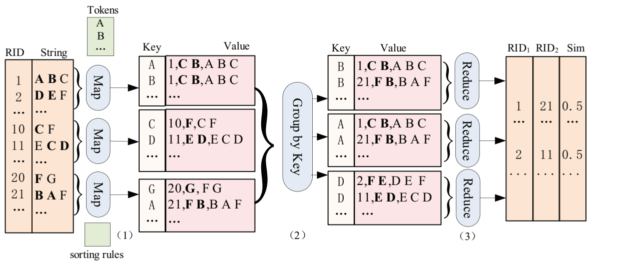


Fig. 4. Data Flow of MGJoin Using MapReduce

Algorithm 4: $Setup(context)$

```
input : $context$: the context of current job
        $distributed\ cache\ files$
output: $tokenWeight$ : tokens with weight for each
        ordering rule
1  $G \leftarrow$ global ordering number;
2  foreach $token \in sorted\ token\ file$ do
3      $tokenWeight[0] \leftarrow <token, weight>$;
4      $tokensVector.add(token)$;
5  for $i = 1 \rightarrow G$ do
6      $stdFile \leftarrow$ read rules for sorting tokens;
7      $tokenWeight[i] \leftarrow$
       $sortTokens(tokensVector, stdFile)$;
```

Algorithm 5: $MGJoinOnCloud$

```
1  Setup(context):
2  read cache files and initialize weight of tokens;
3  Map(key,value):
4  $tokenlist \leftarrow Tokenize(value)$;
5  $G \leftarrow$ global ordering number;
6  while $i < G$ do
7      prefixTokens[i++] $\leftarrow getPrefixTokens(tokenlist,$
       $tokenWeight[i++])$;
8  foreach $token \in prefixTokens[0]$ do
9      i $\leftarrow$ 1;
10     while $i < G$ do
11         $newvalue \leftarrow append(prefixTokens[i++])$;
12     $newvalue \leftarrow append(value)$;
13     write($token$,$newvalue$);
14 Reduce(key,values):
15 foreach $val_i, val_j \in values$ do
16     $prefixTokens_i[] \leftarrow getPrefixTokens(val_i)$;
17     $prefixTokens_j[] \leftarrow getPrefixTokens(val_j)$;
18     $result = Testify(prefixTokens_i, prefixTokens_j)$;
19     if $result$ ==$true$ then
20         $sim \leftarrow sim_{jaccard}(val_i, val_j)$;
21         if $sim \geq \theta$ then
22             $newvalue \leftarrow (val_i, val_j)$;
23             write($key$,$newvalue$);
```

Before conducting similarity join, a setup procedure is invoked on each data node to load token file and sorting rules used to generate global orderings (line 1 of Algorithm 5). The token file and sorting rules are distributed as cache files before executing the task.

In Map, for each string, a prefix token set is derived for each global orderings (line 7). In the paper, the ascending order of token's term frequency is applied as the first global ordering to generate signatures for each string (line 8). The other prefix token sets generated by other global orderings are used as filtering features, which are treated as a piggyback of the string (lines 9-13). Take the first string of Figure 4 for example, each character represents a word. The prefix tokens under two order strategies are AB and CB respectively. The first string will generate two key/value pairs with A and B is two keys. The value is a new string constituted by three parts: record identity (1 in this example), other prefix token sets (C, B), and original record content $A\ B\ C$. Therefore, the two new key/value pairs generated from the first string in Map are $< A\ 1, CB, ABC >$ and $< B\ 1, CB, ABC >$. Following the Map phase, the string pairs with the same key will be shuffled to the same node.

In $Reduce$, the strings with the same key will be verified. To improve performance of verification, multiple prefix filtering will be applied in the $Reduce$ procedure. If two strings of a candidate pair don't share any common prefix token for one global ordering, they will not be similar and are pruned directly (line 20). The *Testify* function is the same with Algorithm 2.

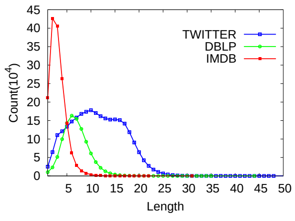
5.1 Optimizations

When applying multiple prefix filtering, the simple way is to obtain prefix tokens using each global ordering and append them to the record. However, this will incur much cost for data transferring. In order to achieve better performance, we adopt some strategies to optimize the implementation.

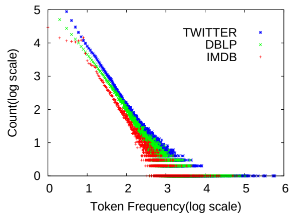
- **Balancing the Workload**
  As the processing cost and output on each node are related to the length of records, we partitioned the dataset into multiple parts in the second MapReduce procedure. In all these splits, the records with different length are evenly distributed into each split.
- **Reducing Sorting Cost**
  For each global ordering, its reverse ordering is also applied so that we only sort the records once for these two global orderings.
- **Reducing the Shuffle Cost**
  In order to minimize the cost of shuffling, we don't extract all prefix tokens using all global orderings and append them to the record. Conversely, we canonicalize the record using another global ordering and output it as the new record. In Reduce, we can easily extract the prefix tokens for the global ordering and its reverse ordering.
- **Ranking Records over the same Reducer**
  As MapReduce framework shuffles the key/value pairs according to the keys, the records that with the same key(signature) will arrive at the reduce node randomly. In order to ensure the records that with the same prefix token can arrive at the reduce node according to their length. We use the composite key for each record, including the generated signature and the length of the record. The records that with the same generated signature (prefix token) will be shuffled to the same node, then the records arriving at the same node will be grouped together and sorted by their length automatically.

TABLE 7
Data Set Information

Data Set	Scale Times	Record Number	Size
IMDB	1x	1,568,893	43M
	5x	7,844,465	215M
DBLP	1x	1,021,062	218M
	5x	5,105,310	1090M
	10x	10,210,620	2.12G
	15x	15,315,930	3.19G
	20x	20,421,240	4.26G
	25x	25,526,550	5.32G
	30x	30,631,860	6.39G
	40x	40,842,480	8.5G
Twitter	1x	2,753,005	250M
Yago	1x	17,630,489	1G
	10x	176,304,890	10G
	20x	352,609,780	20G
	30x	528,914,670	30G



(a) Record Length



(b) Term Frequency

Fig. 5. Distribution Statistic

6 EXPERIMENTS

In this section, we compare our proposed approach, $MGJoin$ with the state-of-the-art methods (using publicly available implementation) under the same conditions.

6.1 Experiments Setup

We select four publicly available real data sets and multiple synthetic data sets in the experiment. They cover a wide range of data distributions and are widely used in previous studies.

- **DBLP** is a snapshot of the bibliography records downloaded from DBLP website[2]. It contains 1,021,062 records, each of which is the concatenation of author name(s) and the title of a publication. The minimum, maximum, average length

2. http://www.informatik.uni-trier.de/~ley/db

(number of tokens) of records in this data set are 2, 207, 13, respectively.

- **IMDB** is a snapshot of movie names taken from the IMDB website[3]. It contains 1,568,893 records. The minimum, maximum, average length of records in this data set are 1, 29, 3, respectively.
- **Twitter** is extracted from the Twitter website[4]. It contains 2,753,005 records. The minimum, maximum, average length of records in this data set are 1, 33, 10, respectively.
- **Yago** is downloaded from the website[5]. It contains 17,630,489 records. It is widely used in semantic web research.
- **Synthetic Data Sets** are generated with different scales based on the above two real data sets according to the generation rule [22]. The generation rule is described as follows. We first compute the token set U that contains all tokens of strings in the dataset, and rank tokens in U based on some order $\mathcal{O}$. For each string s, we split it into a set of tokens, and rank these tokens based on $\mathcal{O}$. The first synthetic string s_1 based on s is generated as follows: (1) for each token t in T_s, we select the next token of t in $\mathcal{O}$ as a new token t'; (2) these tokens constitute the token set of s_1. Also, the second synthetic string s_2 can be generated similarly by selecting the next next token of t as the new token t'. Note that for a token t in s, if there does not exist the new token t', then t will be set as t'. Obviously, according to this generation rule, different scales of the synthetic datasets are able to be generated. Figure 7 shows the synthetic datasets based on DBLP and IMDB that will be used in the experiments. In which, $5\times$ is to indicate that the synthetic data set is 5 times larger than original data set.

The Figure 5 shows the distribution of record lengths and term frequency in DBLP, IMDB and Twitter respectively. We conduct all experiments in both the centralized system and distributed system.

- **Centralized System:** All experiments in the centralized system are carried out on a single machine with AMD 15×4 cores 1GHz and 60GB main memory. The operating system is *Redhat Enterprize Linux AS 4*. All algorithms are implemented using *C++* and compiled with *GCC4.3.0*.
- **Distributed and Parallel System:** The performance of $MGJoin$ is evaluated on Hadoop platform. The cluster consists of 41 computer nodes, among which one is configured as the master and the others are configured as slaves. All nodes have the same configurations as follows. (1) CPU: Intel(R) Xeon(R) 2.40GHz with 4×4 cores; (2) Main memory: 8GB; (3)Disk: 160GB; (4) OS: CentOS; (4) software environment: Hadoop-0.20.2 and JDK-1.6.0.

Unless otherwise specified, we use Jaccard as the default similarity function, and the join threshold is set to 0.8. The global orderings are selected according to *TF as* 1^{st} and three pairs of them are applied in the centralized system experiments.

6.2 Experiments on the Centralized System

We use the following algorithms in the experiment.

- **AllPairs** *AllPairs* [3] algorithm takes two records that share at least one common token in their prefixes as a candidate pair, and verify each candidate pair by directly computing their similarity.
- **PPJoin** Different from *AllPairs*, in the candidate generation phase, *PPJoin*[27] utilizes the position information of the common tokens for each candidate pair so as to apply some pruning techniques before adding to candidate set.
- **PPJoin+** Based on *PPJoin*, in *PPJoin+* [27], extra pruning techniques based on the suffix tokens (tokens that ranked after the prefix tokens) is used to prune more false positives.

6.2.1 Number of Global Orderings

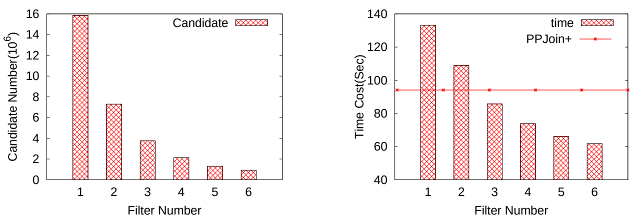
We study the effect of the number of global orderings for *MGJoin* on the synthetic dataset of $5\times$ DBLP. The number of global orderings varies from 1 to 6, and they are generated as follows. The first global ordering is based on the ascending order of token frequency in the data set, and the second is its reverse order. The rest of global orderings are two randomly selected orders and their reverse orders.

We plot the number of candidate pairs after applying multiple global orderings in Figure 6(a). We observe that when the number of global orderings increases, the number of candidate pairs drops exponentially. Nevertheless, there is an obvious trends of diminishing returns. Further, the diminishing trends of applying a reverse order (even numbers) is obviously larger than that of applying a new randomly generated order (odd numbers). This is because the order of tokens in the data set are completely changed after applying a reverse order, which leads to reset the prefix of each string. Figure 6(b) shows the running time of *MGJoin* by using multiple global orderings. We see that the trends of the running time is in consistency with that of the number of candidate pairs. The horizontal line in the figure indicates the time cost of *PPJoin+* with other things equal.

6.2.2 Effect of Thresholds

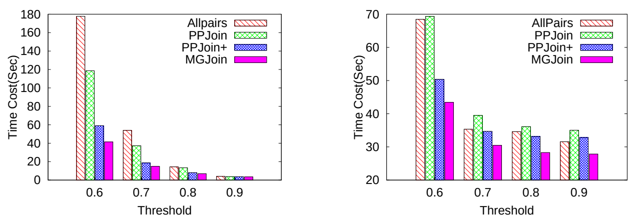
We compare the efficiency of *MGJoin* with other existing algorithms by varying the threshold from 0.6 to 0.9. Figure 7, Figure 8, Figure 9 and Figure 10 show the running time for these algorithms over DBLP, IMDB, $5 \times$ DBLP, $5 \times$ IMDB and Twitter, respectively. Figure

3. http://www.imdb.com
4. http://www.twitter.com
5. http://www.mpi-inf.mpg.de/yago-naga/yago



(a) # of candidate pairs (b) running time

Fig. 6. Effect of Global Orderings over 5×DBLP



(a) DBLP (b) IMDB

Fig. 7. Real Dataset

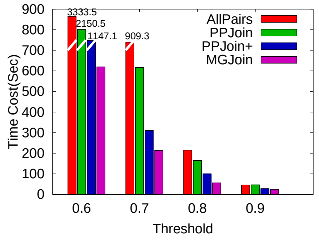


Fig. 8. 5× DBLP

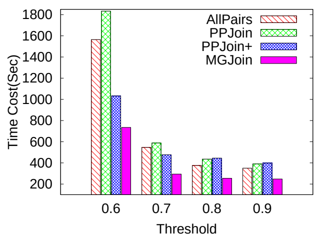


Fig. 9. 5× IMDB

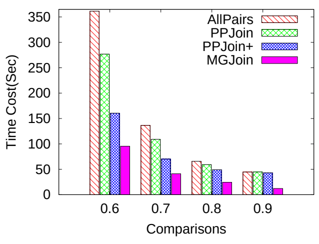


Fig. 10. Twitter

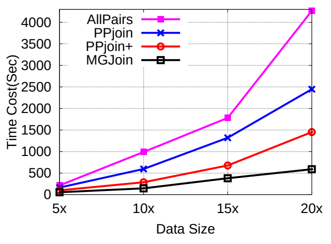


Fig. 11. Scalability

10 shows the results on Twitter. As the running time for *AllPairs* and *PPJoin* is extremely high in IMDB and 5 × IMDB data sets, we do not plot them in the figures so as to simplify the presentation. As we can see, $MGJoin$ always performs the best, followed by *PPJoin+*. When the threshold varies, *Allpairs* is the most sensitive to the threshold while *MGJoin* is the least.

6.2.3 *Time Cost Analysis*

We do experiments to analyze the time cost distribution on two phases: candidate generation and verification. The comparison of time cost between *PPJoin+* and *MGJoin* is given in Figure 12(a). It shows that *PPJoin+* spends too much time on pruning false positives. As the dataset size increases, the time cost in candidate generation phase for *PPJoin+* grows much faster than *MGJoin*. Although the time cost in verification of *MGJoin* is increasing more than *PPJoin*, the increasement is much smaller compared with the time saved in candidate generation phase. The variation of time cost distribution with the number of filters is plotted in Figure 12(b). Totally, the time cost in candidate verification decreases exponentially when use more filters, also the same to the total time cost. In the figure, when using one filtering, scanning inverted index with length filtering, the time cost in candidate generation is small. But the time cost in verification is very high, as too much false positives are not pruned. When using more filters, more false positives are pruned and the time spend in verification decreases exponentially.

6.2.4 *Effect of Similarity Functions*

We compare the efficiency of *MGJoin* with other existing algorithms by varying different similarity functions over the DBLP data set. We plot the running time of these algorithms by applying Cosine and Jaccard similarity functions, as shown in Figure 13. We observe that *MGJoin* performs the best, followed by *PPJoin+*, *PPJoin*, and *Allpairs*. This indicates that our proposed technique is effective for both Cosine and Jaccard similarity functions. In addition, the running time for all algorithms almost keeps unchanged no matter which similarity function is adopted.

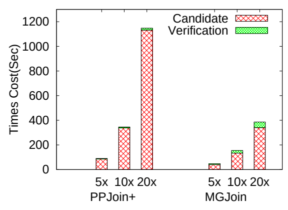
6.2.5 *Scalability Comparison*

We study the scalability by comparing *MGJoin* with other existing methods by varying different scales $\{5\times, 10\times, 15\times, 20\times\}$ of DBLP synthetic data set. The results are shown in Figure 11. As we can see, our method scales well, and the trend of differences for running time between *MGJoin* and other existing methods turns larger as the scale size increases.

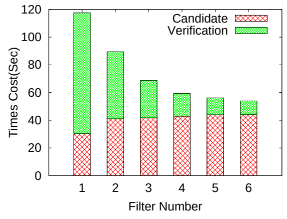
6.2.6 *Global Ordering Selection*

The key problem of applying multiple prefix filtering is to select appropriate global orderings. We compare four strategies for global ordering selection, including *Random*, *Random with Reverse* and two kinds of *TF as the* 1^{st} *Order* including *TF1* and *TF2*. *TF1* is the same to the definition of *TF as the* 1^{st} *Order*, but in *TF2* the reverse order of term frequency $\mathcal{O}_{tf}^{-1}$ is removed. *Random* and *Random with Reverse* take the same random global ordering as their first global ordering; *TF1* and *TF2* take the $\mathcal{O}_{tf}$ as their first global ordering. We study their efficiency in two aspects, time cost and candidate set reduction scale, using DBLP dataset. The results are given in Figure 14, Figure 15 and Figure 16.

In all these four methods, the first global ordering is applied to normalize each string into token sequence and build inverted index for prefix tokens. In Figure

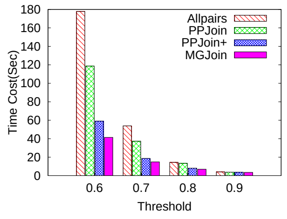


(a) On Different DataSet

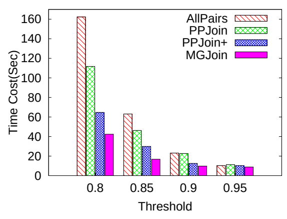


(b) With Filter Number

Fig. 12. Time Cost Distribution



(a) Jaccard



(b) Cosine

Fig. 13. Effect of Similarity Functions

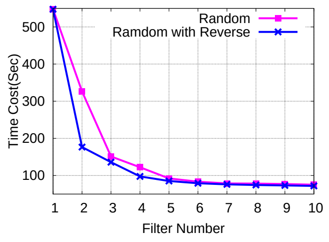


Fig. 14. Random Methods

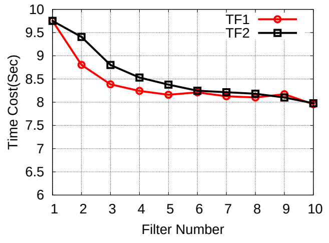


Fig. 15. TF as 1^{st} Methods

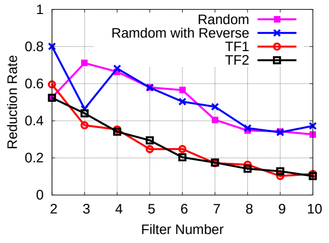


Fig. 16. Reduction Rate

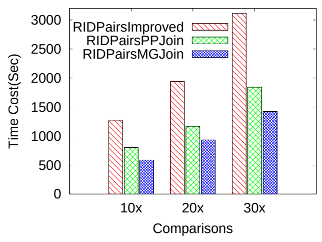


Fig. 17. Yago Dataset

14, it compares the time cost between *Random* and *Random with Reverse*. The results demonstrate that the *Random with Reverse* is more efficient than *Random*, especially when using the second(reverse) filtering. In Figure 15, it compares the time cost between two kinds of *TF as the* 1^{st} *Order*, *TF1* and *TF2*. From the figure we can observe that *TF1* is better than *TF2* when applying the second filtering. From Figure 14 and Figure 15, we observe that the difference in time cost between each pairs of selection methods become less as more filters are applied. In *TF1* and *TF2*, as the tokens with low frequency are to be selected as prefix tokens, most false positives are pruned off in the first two filters. *TF1* and *TF2* are far better than *Random* and *Random with Reverse*.

Figure 16 demonstrates the variation of candidate set reduction rate of four methods. We can observe that the reduction rate of *TF1* and *TF2* are more stable than the other two methods. When using the second filtering, the reduction rate of *Random with Reverse* and *Random* is 0.8 and 0.52, respectively. So, one global ordering and its reverse ordering can get high prune power when applied in pairs. This situation can also be found in *TF1* and *TF2*. For *TF1* and *TF2*, we can found their lines in Figure 16 are interleaved with each other. This is due to the different sequence to apply the reverse global orderings. As more filters applied, more false positives are pruned off and the reduction rate difference between *TF1* and *TF2* become less.

6.2.7 Hybrid Method Study

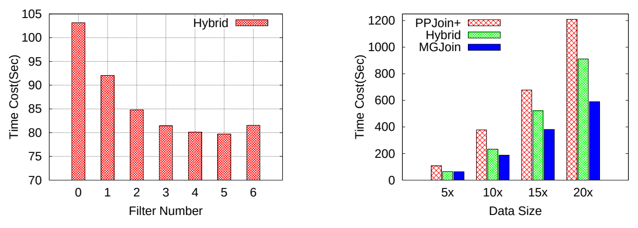
The multiple prefix filtering can be used as a complementary filtering method with other methods. In this experiment, we study the function of *MGJoin* used as a complementary method to *PPJoin+*. The time cost trend with the variation of filter number is plot in Figure 18(a). We do the experiment on $5\times$ DBLP dataset and set similarity threshold at 0.8. From the figure we can see that the time cost decreases firstly with the filter number increasing and reaches the lowest point when using the 6^{th} filter. But, when we use the 7^{th} filter the time cost trend become increasing.

We compared the hybrid method, *PPJoin+* complemented with *MGJoin*, with *PPJoin+* and *MGJoin*. We conduct the experiment on DBLP dataset with scale varying from $5\times$ to $20\times$ and set threshold to 0.8. The results are shown in Figure 18(b). The results demonstrate that *MGJoin* can improve the performance of *PPJoin* when used an complementary method. But, the hybrid method cannot outperform the *MGJoin*, as the expensive filter strategy in *PPJoin+*.

6.3 Experiments on the MapReduce System

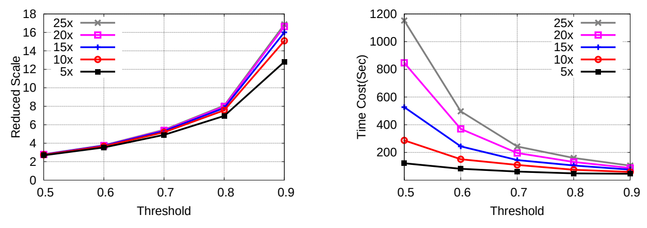
We extend *MGJoin* on Hadoop platform, called as *RIDPairsMGJoin*. We compare it with *RIDPairsImproved* and *RIDPairsPPJoin* [22] on the Hadoop system.

- **RIDPairsImproved** implements the *PPJoin+* on the MapReduce. In Map, it extracts the prefix tokens from each string and assign each of them as a key. In Reduce, the strings with the same key will be in the same group for further verification.
- **RIDPairsPPJoin** has the same implementation in the Map phase as the above method, but the records are sorted by their length during shuffle procedure, before arriving the Reduce node. In Reduce, the inverted index is built for each group to accelerate processing.



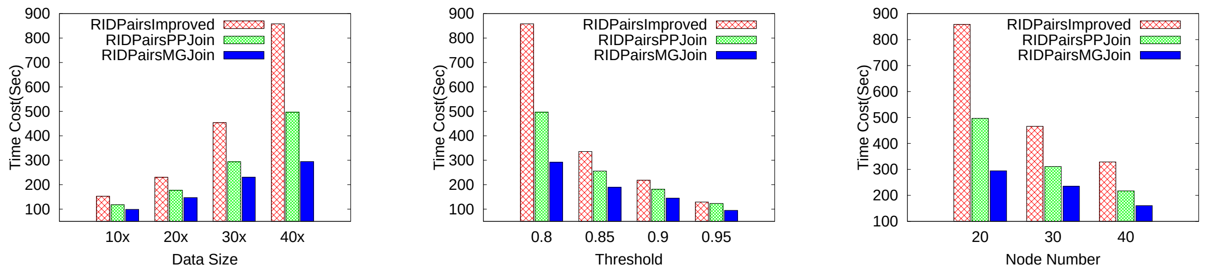
(a) Time Cost (b) Comparisons

Fig. 18. Hybrid Method Study



(a) Candidates Reduction (b) Time Cost

Fig. 19. MGJoin On Cloud



(a)Comparison on Data Scale (b) Scalability on Threshold (c) Scalability on Data Nodes

Fig. 20. Comparison On Cloud

6.3.1 $MGJoin$ in MapReduce

The filtering function of our proposed method is evaluated on $Hadoop$ platform, in which 20 data nodes are used. The experiments are carried on the synthetic datasets of DBLP with the scale from $5\times$ to $25\times$. As demonstrated in the first experiment, *MGJoin* outperforms all other methods when using three global orderings. In this experiment the ascending ordering of term frequency and its reverse order are applied, including the alphabetical ordering. The similarity threshold varies from 0.5 to 0.9.

The Figure 19(a) plots the decreasing ratio of candidates number by applying *MGJoin*. The more digit in y-axis indicates the more false positives are pruned out after applying *MGJoin*. In the figure, the digit 4 is to say that the candidate number become a quarter less after applying *MGJoin*. From the figure we observe that the filtering function of *MGJoin* is almost the same on different datasets. When the threshold increases, more false positives will be pruned, relatively.

Figure 19(b) shows the time cost of *MGJoin* on different scale data sets. The time cost is more when the data set scale is larger when using the same threshold. Also, the time cost is more when the similarity threshold is less on the same scale dataset, as more candidates are produced.

6.3.2 Efficiency Comparison

In this experiment, we compare the efficiency of *MGJoin* with state-of-the-art method, *PPJoin+*. The synthetic DBLP datasets are used with different scale from $10\times$ to $40\times$, and the synthetic Yago datasets with scale from $10\times$ to $30\times$. The experiment is carried on 20 data nodes with threshold set to 0.8. The Figure 20(a) and Figure 17 plot the time cost of the three methods on two kinds of datasets. From the figures we observe that *MGJoin* is more efficient than *PPJoin+* on different datasets. As the scale of dataset increasing, the time cost for these two methods are all increasing, but the time cost of *MGJoin* grows relatively slower than *PPJoin+*.

6.3.3 Scalability

In this subsection, we study the performance and scalability of the *MGJoin* by varying the similarity threshold from 0.8 to 0.95 on the $40 \times DBLP$ dataset. In this experiment, we use 20 nodes. The result is presented in Figure 20(b). Obviously, the total time of all three methods decreases as the similarity threshold increases. This is because the candidate number is decreased when the threshold increases. Among these methods, the *MGJoin* performs the best with any threshold.

We also plot the results of scalability by varying the number of data nodes used in the cluster from 20 to 40 in Figure 20(c). We set the similarity threshold to be 0.8, and use the $40\times DBLP$ dataset. It can be seen that the *MGJoin* consistently outperforms the other two methods with different node numbers in the cluster.

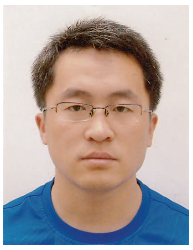
7 Conclusions and Future Work

In this paper, we propose a new string similarity join method called *MGJoin*. It is based on multiple prefix filters, each of which applies different global orderings. It's the first work to explore the function of multiple prefix filters with different global orderings. The extensive experiments are conducted on centralize system and distributed computing environment.

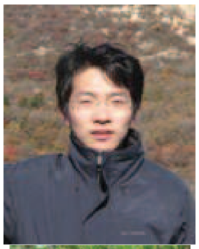
The experiment results demonstrate that our proposed method outperforms state-of-the-art methods in efficiency and scalability. In next work, the further research on global ordering number and sequence will be carried.

REFERENCES

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929. ACM, 2006.
[2] A. Arasu, C. Ré, and D. Suciu. Large-scale deduplication with constraints using dedupalog. In *ICDE*, pages 952–963, 2009.
[3] R. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140. ACM, 2007.
[4] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 61–72. IEEE, 2006.
[5] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD*, pages 707–718. ACM, 2009.
[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[7] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, pages 85–96. ACM, 2005.
[8] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *WWW*, pages 613–622. ACM, 2001.
[9] A. Elmagarmid, P. Ipeirotis, and V. Verykios. Duplicate record detection: A survey. *TKDE*, pages 1–16, 2007.
[10] I. Fellegi and A. Sunter. A theory for record linkage. *ASA*, 64(328):1183–1210, 1969.
[11] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500. ACM, 2001.
[12] S. Guha, N. Koudas, A. Marathe, and D. Srivastava. Merging the results of approximate match operations. In *VLDB*, pages 636–647. ACM, 2004.
[13] M. Hernández and S. Stolfo. The merge/purge problem for large databases. In *SIGMOD*, pages 127–138. ACM, 1995.
[14] R. Kumar and S. Vassilvitskii. Generalized distances between rankings. In *WWW*, pages 571–580. ACM, 2010.
[15] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266. IEEE, 2008.
[16] A. Monge and C. Elkan. The field matching problem: Algorithms and applications. In *Proceedings of the second international Conference on Knowledge Discovery and Data Mining*, pages 267–270, 1996.
[17] F. Naumann and M. Herschel. An Introduction to Duplicate Detection. *Synthesis Lectures on Data Management*, 2(1):1–87, 2010.
[18] G. Salton and M. McGill. Introduction to modern information retrieval. 1986.
[19] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *SIGKDD*, pages 269–278. ACM, 2002.
[20] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, pages 743–754. ACM, 2004.
[21] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *Computer Vision*, pages 1470–1477. IEEE, 2003.
[22] R. Vernica, M. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD*, pages 495–506. ACM, 2010.
[23] C. Wang, J. Wang, X. Lin, W. Wang, H. Wang, H. Li, W. Tian, J. Xu, and R. Li. Mapdupreducer: detecting near duplicates over massive datasets. In *SIGMOD*, pages 1119–1122. ACM, 2010.
[24] J. Wang, J. Feng, and G. Li. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *VLDB*, 3(1-2):1219–1230, 2010.
[25] W. Winkler. The state of record linkage and current research problems. In *Statistical Research Division*, 1999.
[26] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann, 1999.
[27] C. Xiao, W. Wang, X. Lin, and J. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140. ACM, 2008.
[28] Z. Zhang, M. Hadjieleftheriou, B. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*, pages 915–926. ACM, 2010.



Chuitian Rong is a Ph.D. student in School of Information, Renmin University of China. His main research interests include database system, information integration and cloud computing.



Wei Lu is a research fellow at NUS, Singapore. He received his Ph.D degree in computer science from Renmin University of China in 2011. His research interest includes query processing in the context of spatio-temporal, cloud database systems and applications.

Xiaoli Wang received her B.S. from School of Information Science and Engineering, Northeastern University, Shenyang, China, in 2008. She is current a Ph.D. student and Research Assistant in School of Computing, National University of Singapore. Her research interests are mainly in indexing and query processing on the complex structure, such as sequence, tree and graph.

Xiaoyong Du is a professor at Renmin University of China. He received his Ph.D. degree from Nagoya Institute of Technology in 1997. His research focuses on intelligent information retrieval, high performance database and unstructured data management.

Yueguo Chen received the BS and Master degree in Mechanical Engineering and Control Engineering from Tsinghua University, Beijing, in 2001 and 2004. He earned his Ph.D. degree in Computer Science from National University of Singapore in 2009. He is currently an Assistant Professor in the Key Laboratory of DEKE, Renmin University of China. His recent research interests include massive knowledge bases and semantic search.

Anthony K.H. Tung received the BSc (Second Class Honor) and MSc degrees in computer science from the National University of Singapore (NUS) in 1997 and 1998, respectively, and the PhD degree in computer sciences from Simon Fraser University (SFU) in 2001. He is currently an Associate Professor in the Department of Computer Science, National University of Singapore. His research interests involve various aspects of databases and data mining (KDD) including buffer management, frequent pattern discovery, spatial clustering, outlier detection, and classification analysis.