Front.Comput.Sci.
DOI

RESEARCH ARTICLE

String Similarity Join With Different Similarity Thresholds Based On Novel Indexing Techniques

Chuitian Rong (✉)[1], **Yasin N. Silva**[2], **Chunqing Li**[1]

1 School of Computer Science & Software Engineering, Tianjin Polytechnic University,Tianjin 300387, China
2 Arizona State University, Tempe, AZ 85281, USA

Abstract String similarity join is an essential operation of many applications that need to find all similar string pairs from two given collections. A quantitative way to determine whether two strings are similar is to compute their similarity based a certain similarity function. The string pairs with similarity above a certain threshold are regarded as results. The current approaches to solve the similarity join problem use a unique threshold value. There are, however, several scenarios that require the support of multiple thresholds. For instance, when the dataset includes strings of various lengths. In this scenario, longer string pairs typically tolerate many more typos than shorter ones. Therefore, we proposed a solution for string similarity joins that supports different similarity thresholds in a single operator. In order to support different thresholds, we devised two novel indexing techniques: partition based indexing and similarity aware indexing. To utilize the new indices and improve the join performance, we proposed new filtering methods and index probing techniques. To the best of our knowledge, this is the first work that addresses this problem. Experimental results on real-world datasets show that our solution performs efficiently while providing a more flexible threshold specification.

Keywords Similarity Join, Similarity Aware Index, Similarity Thresholds

Received month dd, yyyy; accepted month dd, yyyy

E-mail: *chuitian@tjpu.edu.cn*

1 Introduction

String is a fundamental data type and widely used in a variety of applications, such as recording product and customer names in marketing, storing publications in academic research, and representing the content of web sites. Frequently, different strings from different sources may refer to the same real-world entity due to various reasons.

In order to combine heterogenous data from different sources and provide a unified view of entities, the string similarity join is proposed to find all pairs of strings between two string collections based on a string similarity function and a user specified threshold. The existing similarity functions fall into two categories: set-based similarity functions (e.g., Jaccard [1]) and character-based similarity functions (e.g., Edit Distance). The threshold is a value in [0,1] or a suitable integer according to the similarity function. In general, the implementation of the similarity join operation is different based on the type of similarity function that is used. For the character-based similarity functions, the implementations consider each string as a sequence of characters and employ a tree-based index structure, such as the B^+-tree [2, 3] and the Trie-tree [4]. 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 in the case of short strings. Thus, the tree-based indices are not suitable for processing similarity joins over very large string collections. For

Table 1 Data Sets

	ID	String
$\mathcal{R}$	r_1	Probabilistic set similarity joins
	r_2	Efficient exact set-similarity joins
	r_3	Efficient parallel set similarity joins using MapReduce
$\mathcal{S}$	s_1	Top-k set similarity joins
	s_2	Efficient exact set similarity joins
	s_3	Efficient parallel set-similarity joins using MapReduce

Table 2 Prefix Tokens Using Different Thresholds

θ	ID	String
0.8	s_1	**Top-k** similarity set joins
	s_2	**similarity** set joins exact Efficient
	s_3	**using set-similarity** MapReduce parallel joins Efficient
0.6	s_1	**Top-k similarity** set joins
	s_2	**similarity set** joins exact Efficient
	s_3	**using set-similarity MapReduce** parallel joins Efficient

the set-based similarity functions, the implementations usually divide each string into a set of tokens and extract its signatures, and then index the signatures using inverted indices [5, 6]. 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 two given collections, the methods that employ inverted indices follow a filter and refine process. In the filter step, they generate a set of candidate pairs that share a common token. In the verification step, they verify the candidate pairs to generate the final result. However, if the strings contain popular tokens, the number of qualified candidate pairs will be very large. To address this problem, the prefix filtering [7] method has been proposed. According to this technique, all the strings in the collections are sorted based on a global ordering and the first T tokens are selected as their prefix. The number of T is determined by $|s|$ (the number of words in s), the similarity function sim, and the user specified similarity threshold θ. Using the Jaccard similarity function as an example, $T = |s| - \lceil |s| * \theta \rceil + 1$. This means that for any other string r, the necessary condition of $sim(s, r) \geqslant \theta$ is that the prefixes of s and r must have at least one token in common [6–8]. Let us consider the Example 1 and apply a prefix filtering technique to generate its candidate pairs, based on the Jaccard similarity function.

Example 1 Table 1 lists two string collections of publication titles from two different data sources. We sorted the words of each string in $\mathcal{R}$ and $\mathcal{S}$ based on reverse alphabetical order and selected the first $(|s| - \lceil |s| * \theta \rceil + 1)$ words (prefixes) as their prefix. Then, we constructed two inverted indices for the prefix tokens of each string in $\mathcal{S}$ using $\theta = 0.8$ and $\theta = 0.6$, respectively. The prefix tokens of strings in $\mathcal{S}$ are given in Table 2. The inverted indices for prefix tokens of strings in $\mathcal{S}$ are shown in Figure 1.

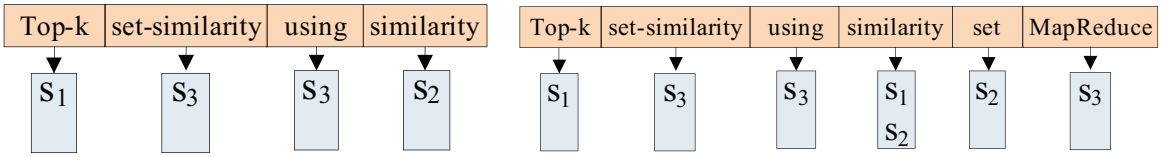
During the similarity join process, we use the prefix tokens of every string in $\mathcal{R}$ to probe the corresponding inverted index and generate candidate pairs. When $\theta = 0.8$, we have two candidate pairs $\{< r_2, s_3 >, < r_3, s_3 >\}$. When $\theta = 0.6$, we have seven candidate pairs $\{< r_1, s_1 >, < r_1, s_3 >, < r_2, s_1 >, < r_2, s_3 >, < r_3, s_3 >, < r_3, s_3 >, < r_3, s_2 >\}$ and two final results $< r_1, s_1 >$ and $< r_3, s_3 >$. When $\theta = 0.5$, we have an additional result pair $< r_2, s_2 >$. Observe that all the pairs $< r_i, s_i >$(i=1,2,3) refer to the same publications, respectively. In fact, the values of $sim(r_i, s_i)$(i=1,2,3) are different as there are some different tokens between r_i and s_i. In general, the same number of spelling differences will generate different values of similarity distance depending on the length of the strings. This can be observed in Table 3. For instance, the string pairs $< r_2, s_2 >$ and $< r_3, s_3 >$ have three different tokens from each other but the similarity value of the longer string pair $(< r_3, s_3 >)$ is greater than the shorter one $(< r_2, s_2 >)$. All the previously proposed methods use a predefined or unique similarity threshold. By doing this, they can lose some promising results. We propose a solution that supports the use of different thresholds for different strings. In this paper, we focus primarily on solutions to support different similarity thresholds, and leave the problem of assigning suitable thresholds to strings as a future work.

Table 3 The Variation of Similarity Value

Similarity Value	Length of r_i	Length of s_i	♯ Different Tokens
$sim(r_1, s_1)$=0.6	4	4	2
$sim(r_2, s_2)$=0.5	4	5	3
$sim(r_3, s_3)$=0.625	7	6	3

The use of a single threshold, considered in previous methods, simplifies the design of efficient algorithms. The support of different similarity thresholds present three challenges that need to be addressed: (1) As there are a variety of different thresholds, the widely used prefix filtering method can not be applied for inverted index construction; (2) Since the thresholds are not know before the join operation, we should devise new indexing mechanisms; (3) We need to explore new index probing techniques to improve the performance.

In summary, this paper makes the following contributions:



(a)$\theta = 0.8$ (b) $\theta = 0.6$

Fig. 1 Inverted Index For Prefix Tokens of DataSet $\mathcal{S}$

- We proposed a solution for string similarity join with different similarity thresholds. This is the first work that explores similarity joins with diverse thresholds in one operation.
- We devised two novel indexing structures, partitioned-based index and similarity-aware index, to support similarity joins with different thresholds.
- We provide new index probing techniques and filtering mechanisms to improve the join performance.

The rest of the paper is organized as follows. The related work is given in Section 2. Section 3 presents the problem definitions and preliminaries. Section 4 describes the index structures. Section 5 presents how to perform similarity joins by employing the proposed indexing structures and new probing techniques. Experimental evaluation is given in Section 6. Section 7 concludes the paper.

2 Related Work

String similarity join is a primitive operation in many applications such as merge-purge [9], record linkage [10], object matching [11], reference reconciliation [12], deduplication [13, 14] and approximate string join [15]. In order to avoid verifying every pair of strings in the dataset and improve performance, string similarity join typically follows a filtering and refine process [16, 17]. In the filtering step, the signature assignment process or blocking process is invoked to group the candidates by using either an approximate and exact approach, depending on whether some amount of error could be tolerated or not. In the past two decades, more than ten different algorithms have been proposed to solve this problem [18]. In [18], the authors evaluated existing algorithms under the same experimental framework and reported comprehensive findings. 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 structures. In [19], 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 [4], in which sub-trie pruning techniques are applied. In [2], 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. In [3], the authors partitioned the string collection into a number of groups according to a set of reference strings and index all the strings into a B^+-tree based on the distances to their reference strings. However, these algorithms are constrained to in-memory processing, not efficient and scalable for processing large scale dataset. In [20],the authors proposed the pivotal prefix to shorten the prefix length by a dynamic-programming algorithm. They also applied an alignment filter to prune candidate pairs.

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 [6], the prefix filtering was proposed to prune false positives [6–8, 15]. In these methods, the partial result of the filtering step is a superset of the final result. The *AllPairs* method proposed in [7] builds the inverted index for prefix tokens and each string pair in the same inverted list is considered as a candidate. This method can reduce the false positives significantly compared to the method that indexes all tokens of each strings [5]. In order to prune false positives more aggressively, the *PPJoin* method uses 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 [8]. In [21], the authors observed that prefix lengths have significant effect on pruning false positives and the join performance. They proposed the *AdaptJoin* method by utilizing different

prefix lengths. [22] proposed the *MGJoin* method that is based on multiple prefix filters, each of which is based on a different global ordering. [23] studied the problem with synonyms by utilizing a novel index that combines different filtering strategies. In [24], the authors proposed a prefix tree to support string similarity join and search on multi-attribute data. They proposed a cost model to guide the prefix tree construction process.

All the aforementioned works applied a predefined and unique threshold when performing joins. Furthermore, the index that is used to accelerate the join process is build using an specific distance threshold. If the strings have significantly different lengths, the same number of different tokens between pairs of strings will generate different effects on the similarity function value (see Table 3). So, applying a unique threshold may lose some results that are still considered very similar in practical scenarios. In this paper, we first propose a partition based inverted index to support string similarity joins with different similarity thresholds. In order to improve the performance further, we devise a similarity-aware index and new probing techniques.

3 Preliminary

In this section, we provide the problem definition followed by a description of similarity measures and prefix filtering.

3.1 Problem Definition

In this paper, we consider a string as set of tokens, each of which can be a word or n-gram. For example, the tokens set of r_2 in $\mathcal{R}$ (Table 1) is {*Efficient, exact, set, similarity, joins*}. The string similarity join is defined as follows.

Definition 1 String Similarity Join
Given two string collections $\mathcal{R}$ and $\mathcal{S}$, a similarity function *Sim* and each $r \in \mathcal{R}$ has its own join threshold $r.\theta$, the string similarity join finds all the string pairs (r, s), such that $r \in \mathcal{R}$, $s \in \mathcal{S}$, and $Sim(r, s) \geqslant r.\theta$.

3.2 Similarity Measures

A similarity function measures how similar two strings are. There are two main types of similarity functions for strings, set-based similarity functions and character-based similarity functions. In this paper, we utilize three widely used set-based similarity functions, namely Dice [25], Jaccard [1], and Cosine [26], whose computation problem can be reduced to the set overlap problem [7]. They are based on the fact that similar strings share common components. Their definitions are summarized in Table 5, in which T_r denotes the token set of r, V_r denotes the vector transformed from T_r and $|\cdot|$ denotes the size of a set. Unless otherwise specified, we use Jaccard as the default function, i.e., $sim(r, s) = sim_{jaccard}(r, s)$. For example, $sim_{jaccard}(r_2, s_2) = \frac{3}{6}$.

Table 4 Symbols and Definitions

Symbols	Definition
$\mathcal{R}, \mathcal{S}$	collections of strings
$\|\cdot\|$	the element number of a set
t	a token of s
T_r	set of tokens for string r
V_r	T_r transformed to Vector Space Model
T_r^p	the first p tokens of string r
θ	pre-assigned threshold
$sim(r_i, s_j)$	similarity between r_i and s_j
O	global ordering
$T_r^p(O)$	T_r^p under O

Table 5 Similarity Functions

Similarity Function	Definition	Prefix Length
$sim_{dice}(r_i, s_j)$	$\frac{2\times\|T_{r_i}\cap T_{s_j}\|}{\|T_{r_i}\|+\|T_{s_j}\|}$	$\|T_s\| - \lceil\|T_s\| * \theta\rceil + 1$
$sim_{jaccard}(r_i, s_j)$	$\frac{\|T_{r_i}\cap T_{s_j}\|}{\|T_{r_i}\cup T_{s_j}\|}$	$\|T_s\| - \lceil\|T_s\| * \theta\rceil + 1$
$sim_{cosine}(r_i, s_j)$	$\frac{V_{r_i}\cdot V_{s_j}}{\sqrt{\|V_{r_i}\|\times\|V_{s_j}\|}}$	$\|T_s\| - \lceil\|T_s\| * \theta^2\rceil + 1$

3.3 Prefix Filtering

The prefix filtering technique is commonly used in the filtering step to generate candidate pairs that share common prefix tokens. In [6, 7], the methods sort the tokens of each string based on some global ordering 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. Its has been proved that the necessary condition for every two strings r and s to be a candidate pair is that their prefixes T_r^p and T_s^p must have at least one token in common. The number of tokens in the prefix for each string can be computed using the formulas shown in Table 5. Essentially, the prefix length relies on the similarity function, the join threshold, and the length of the string.

By applying the prefix filtering technique, candidate

[1)] When there is no ambiguity, we will simply refer to token prefixes as prefixes.

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 to be selected as prefixes.

3.4 Inverted Index

The naive method to perform joins is to enumerate all pairs $< r, s > \in \mathcal{R} \times \mathcal{S}$ and verify whether they share common prefix tokens. This approach is rather expensive when processing large scale datasets. In order to improve the performance, the inverted index is widely employed to find all pairs $< r, s > \in \mathcal{R} \times \mathcal{S}$ that share common prefix tokens. We first construct an inverted index for prefix tokens of each string in one of the collections, e.g., $\mathcal{S}$. By doing so, the strings that share the same prefix tokens are mapped into the same inverted list. Then, we can process each string $r \in \mathcal{R}$ to get its prefix tokens, and then to merge the corresponding inverted lists for the prefix tokens to get all candidate pairs. Take r_3 as an example, assume $r_3.\theta = 0.6$, its prefix tokens are $\{using, similarity, set\}$. We merge the inverted lists for these three prefix tokens, as shown in Figure 1(b), and get three candidate pairs: $\{< r_3, s_1 >, < r_3, s_2 >, < r_3, s_3 >\}$.

4 Index Technique to Support Different Similarity Thresholds

All the previous work on similarity joins use a predefined and unique threshold for all the objects in the input collections. When a unique threshold is used, it is easy to implement and optimize the similarity join algorithms. The application of a unique threshold may lose some promising results or be too rigid to express the nature of similarities that users want to identify on the datasets. The prefix filtering technique is widely used in existing works due to its effective pruning power. The prefix filtering is based on a necessary condition for similar pairs of strings, which is that they must share at least one prefix token when sorted by the same global order. The prefix tokens of two strings are determined by their length and the unique threshold. Supporting different thresholds in similarity join has its own challenges. Particularly, when constructing the inverted index for prefix tokens of each string $s \in \mathcal{S}$, we cannot predict the threshold of $r \in \mathcal{R}$ that will be used to perform the join with s. The problem is that we cannot exactly identify the number of prefix tokens of $s \in \mathcal{S}$. In this section, we propose the index techniques that can support different similarity thresholds.

4.1 Straightforward Approach

As stated previously, since we cannot predict the threshold of $r \in \mathcal{R}$, we cannot identify the number of prefix tokens of $s \in \mathcal{S}$ and cannot construct the inverted index as in the case of single threshold approaches. The straightforward approach is to map all tokens of $s \in \mathcal{S}$ to inverted lists. When performing joins, we process each string $r \in \mathcal{R}$ and get its prefix tokens. Then, we can get the candidate pairs by merging the corresponding inverted lists. Obviously, this approach will map unnecessary tokens into inverted lists and increase the index size. As a result, this approach incurs on unnecessary overhead when probing the inverted lists and thus the similarity join performance is degraded.

4.2 Partition-Based Inverted Index

The similarity join performance is mainly determined by the prefix filtering power and the efficiency of merging inverted lists. In order to decrease the unnecessary index probing cost, we proposed a **P**artition **B**ased inverted **I**ndex, denoted as $\mathcal{PBI}$ for abbreviation.

When using different thresholds during the join operation, different strings may have different thresholds. As the threshold is a value in [0,1], we choose some representative values within the interval and build incremental inverted indices with them. For example, if we choose 0.8, 0.6 and 0.4, we build the inverted index in the following way. First, we map the prefix tokens of each $s \in \mathcal{S}$ using $\theta = 0.8$ into inverted lists as done in existing works (we denote this inverted index as $I^{0.8}$). Then, we map the prefix tokens using $\theta = 0.6$ into inverted index $\Delta I^{0.6}$, excluding the tokens that have been mapped into $I^{0.8}$. Similarly, we map the prefix tokens using $\theta = 0.4$ into inverted index $\Delta I^{0.4}$, excluding the tokens that have been mapped into $I^{0.8}$ and $\Delta I^{0.6}$. Figure 2 shows the partition-based inverted index for the collection $\mathcal{S}$ in Table 1.

The pseudo-code of the $\mathcal{PBI}$ construction algorithm is shown in Algorithm 1. The algorithm takes a string collection $\mathcal{S}$ that is sorted by string length, a global ordering O, and the number of representative thresholds N as input. Its output is a $\mathcal{PBI}$, such as $I^{\theta}, \Delta I^{\theta_1}, ..., \Delta I^{\theta_{N-1}}$. It first selects N

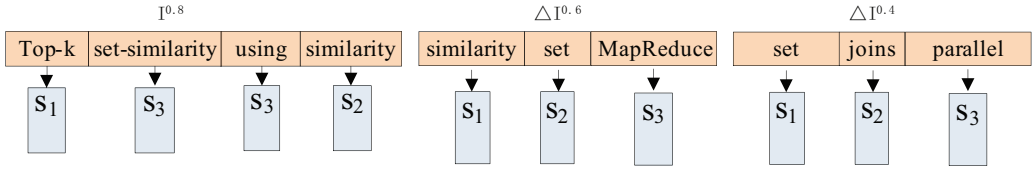


Fig. 2 Partition-Based Inverted Index

Algorithm 1: *PartitionBasedIndex*$(\mathcal{S}, \mathcal{O}, \mathcal{N})$

Input :
$\mathcal{S}$: the collection of string
$\mathcal{O}$: one global ordering
$\mathcal{N}$: the number of representative thresholds
Output:
PBI: $I^{\theta}, \Delta I^{\theta_1}, ..., \Delta I^{\theta_N}$

1 $\theta[\mathcal{N}] \leftarrow$ select $\mathcal{N}$ representative thresholds from [0,1];
2 **foreach** $s \in \mathcal{S}$ **do**
3 $TokenList$(s) $\leftarrow$ Tokenize(s,$\mathcal{O}$);
4 **for** $i = 0 \rightarrow \mathcal{N}$ **do**
5 $T_s^p(\theta[i]) \leftarrow$ getPrefix($TokenList(s), \theta[i]$);
6 **if** $i == 0$ **then**
7 **foreach** $t \in T_s^p(\theta[i])$ **do**
8 Map t into $I^{\theta[i]}[t]$;
9 **else**
10 **foreach** $t \in \{T_s^p(\theta[i]) - T_s^p(\theta[i-1])\}$ **do**
11 Map t into $\Delta I^{\theta[i]}[t]$;
12 **foreach** $t \in \{T_s - T_s^p(\theta[N-1])\}$ **do**
13 Map t into $\Delta I^{\theta[N]}[t]$;

representative thresholds that are used to construct the partition based inverted index (Line 1). For each $s \in \mathcal{S}$, it first generates a sorted token list using the function *Tokenize*, which tokenizes the string s into a token set and then sorts it by the global ordering $\mathcal{O}$ (Line 2). For the sorted token list, the algorithm gets different prefix tokens using different thresholds in $\theta[\mathcal{N}]$. It maps each prefix token $t \in T_s^p(\theta[0])$ into inverted list $I^{\theta}[t]$, which belongs to I^{θ}. For other thresholds in $\theta[\mathcal{N}]$, it maps each prefix token in $T_s^p(\theta[i]) - T_s^p(\theta[i-1])$ into inverted list $\Delta I^{\theta[i]}[t]$, which belongs to $\Delta I^{\theta[i]}$ (Lines 5-10). Finally, it maps the remaining tokens in $T_s - T_s^p(\theta[N-1])$ into inverted index $\Delta I^{\theta[N]}$.

The partition-based inverted index can decrease the probing cost significantly, as it partitions the inverted list of the same token into several smaller lists, as shown in Figure 2. For $r \in \mathcal{R}$, assume $r.\theta = 0.8$, when performing joins we can only probe and merge the inverted lists in $I^{0.8}$. If $r.\theta = 0.7$, we should probe and merge the inverted lists $I^{0.8}$ and $\Delta I^{0.6}$.

Algorithm 2: *SimilarityAwareIndex*$(\mathcal{S}, \mathcal{O})$

Input :
$\mathcal{S}$: the collection of string
$\mathcal{O}$: one global ordering
Output:
SAI: similarity aware index

1 **foreach** $s \in \mathcal{S}$ **do**
2 $TokenList$(s) $\leftarrow$ Tokenize(s,$\mathcal{O}$);
3 $\mathcal{N} \leftarrow$ TokenList.$size$;
4 **for** $n = 0 \rightarrow N$ **do**
5 $\mathcal{TUB}(n) \leftarrow \frac{\mathcal{N}-n+1}{\mathcal{N}}$;
6 $t \leftarrow$ TokenList[n];
7 Map $< t, \mathcal{TUB}(n) >$ into $SAI[t]$;

4.3 Similarity Aware Index

The partition-based inverted index can decrease the unnecessary probing cost to some extent, but there are still ways to improve the performance. In order to do this, we proposed a **S**imilarity **A**ware **I**ndex that exploits the relationship of token positions and similarity thresholds, denoted as $\mathcal{SAI}$.

Definition 2 Threshold Upper Bound
When a string $r \in \mathcal{R}$ (or $s \in \mathcal{S}$) is tokenized into a token set T_r and sorted using the global ordering $\mathcal{O}$, the tokens that are selected as prefix tokens are determined by the value of the threshold. The maximum threshold for a token to be selected as a prefix token is referred to as the threshold upper bound, denoted as $\mathcal{TUB}$.

Theorem 1 *Let token $t \in T_r$ be located at position n when T_r is sorted by global ordering $\mathcal{O}$. If t is a prefix token, $\mathcal{TUB}(n) = \frac{|T_r|-n+1}{|T_r|}$.*

Proof For the token set T_r, the number of prefix tokens $|T_r^p|$ is determined by the formula.

$$|T_r^p| = |T_r| - \lceil |T_r| * \theta \rceil + 1$$

Let $|T_r^p| = n$, that is to say the last prefix token is located at the position n. Then, we can get

$$|T_r| - n < |T_r| * \theta \leqslant |T_r| - n + 1$$

We can get $\theta \leqslant \frac{|T_r|-n+1}{|T_r|}$. So,

$$TUB(n) = \frac{|T_r|-n+1}{|T_r|}$$

□

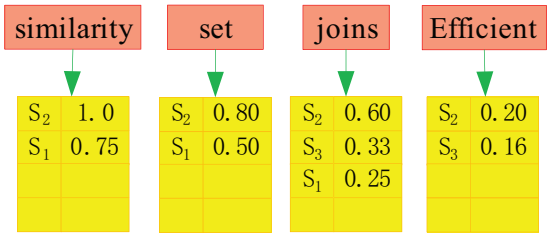


Fig. 3 Similarity Aware Inverted Index

Based on the above analysis, we can conclude that the selection of a token $t \in T_r$ as a prefix token depends on its position n and $\mathcal{TUB}$(n) when $|T_r|$ is sorted by a global ordering. So, we can map both the token t and $\mathcal{TUB}$(n) to the inverted index when constructing the index for collection $\mathcal{S}$. When probing the inverted index, it can support any threshold of $r \in \mathcal{R}$. Unlike the partition-based inverted index that sorts the inverted lists by the length of the strings, we sort the inverted lists by the value of $\mathcal{TUB}$(n). Figure 3 shows part of the similarity-aware inverted index for the collection $\mathcal{S}$ in Table 1.

The pseudo-code of the similarity aware index construction algorithm is shown in Algorithm 2. For each string $s \in \mathcal{S}$, it generates a sorted token list by applying the *Tokenize* function. Then, for each token in *TokenList*, it computes the $\mathcal{TUB}$(n) based on the token's position n and the size of *TokenList*. Finally, it maps the pair $< t, \mathcal{TUB}(n) >$ into $\mathcal{SAI}$[t]. Observe that, each inverted list is sorted by the value of $\mathcal{TUB}$.

5 String Similarity Join Processing

In this section, we describe how similarity join operation can be performed using the proposed indexing mechanism. We present the details about how to handle different thresholds in one join procedure.

In general, the similarity join methods employing inverted indexes follow a filter and refine framework and consist of three phases.

- **Index Construction** In this phase, the strings in the collection $\mathcal{S}$ are sorted based on a global ordering, such as alphabetical order or token's term frequency. Then, a number of tokens will be extracted from each string according to predefined rules and mapped into inverted lists.
- **Candidate Pairs Generation** The strings in the collection $\mathcal{R}$ will be processed sequentially. For each string in $\mathcal{R}$, the string will be sorted and a number of tokens will be extracted from it as done in the index construction step. The extracted tokens are considered as the string's signatures and are used to probe corresponding inverted lists using filtering strategies. As a result, we can derive the candidate set for each string.
- **Verification** We compute the similarity between the string and its candidates one by one to find the final results. The final result will be composed of pairs with similarity not less than θ.

In order to support different similarity thresholds in string similarity joins, the straightforward approach is to extend the *PPJoin* and *PPJoin+* [8] methods using the index as described in section 1. As this approach maps all tokens into inverted lists, the join procedure will probe the index with unnecessary overhead.

5.1 Similarity Join on PBI

Using the partition-based index to support different similarity thresholds, the first problem is to select a number of representative values from [0,1]. The selected values will be used to partition the threshold range into several intervals, and then partition the inverted lists into several small inverted lists. As an example, we can select the following representative values 0.8, 0.6, 0.4. The index for string collection $\mathcal{S}$ is constructed as indicated by Algorithm 1.

After the index construction step, each string $r \in \mathcal{R}$ is sorted based on a global ordering O, and then the prefix tokens T_r^p are extracted according to the length and its own similarity threshold $r.\theta$, as described in section 3. For each prefix token, the corresponding inverted lists with representative thresholds not smaller than $r.\theta$ and the first inverted list with representative threshold equal or smaller than $r.\theta$ will be probed to generate candidates.

The details of similarity join using $\mathcal{PBI}$ is shown in Algorithm 3.

When performing similarity join operations, each string in collection $\mathcal{R}$ will be tokenized by the same global ordering O that was used on string collection $\mathcal{S}$ (Line 2). Take the string $r_3 \in \mathcal{R}$ and the selected representative values θ[3] ={0.8,0.6,0.4} for example. When $r_3.\theta = 0.8$, its prefix token set $T_{r_3}^p$ ={using, similarity} (Line 3). So, the two inverted lists $I^{[0.8]}[using]$ and $I^{[0.8]}[similarity]$ that belong to $I^{[0.8]}$ will be probed and merged. The acquired candidates will be added to C for verification (Lines 4-7). Finally, each candidate pair in C will be verified to check that their similarity is not less than $r_3.\theta$ using the selected

Algorithm 3: $SimJoinOnPBI(\mathcal{R}, \mathcal{S})$

Input :
$\mathcal{R}$: the string collection $\mathcal{R}$
$\mathcal{S}$: the string collection $\mathcal{S}$
Output:
$< r, s >$: similar pairs

1 **foreach** $r \in \mathcal{R}$ **do**
2 $TokenList$(r) ← Tokenize(r,$\mathcal{O}$);
3 T_r^p ← getPrefixToken(TokenList(r),$r.\theta$);
4 **foreach** $t \in T_r^p$ **do**
5 **while** $k < \mathcal{N} \wedge r.\theta \leqslant \theta[k++]$ **do**
6 results ← listProbing($\mathcal{PBI}[k][t]$);
7 $C.push$(results);
8 Verification(C);

similarity function. When $r_3.\theta = 0.7$, $T_{r_3}^p$ ={using, similarity, set}. The inverted lists $I^{[0.8]}[using]$ and $I^{[0.8]}[similarity]$ that belong to $I^{[0.8]}$, $I^{[0.6]}[similarity]$ and $I^{[0.6]}[set]$ that belong to $I^{[0.6]}$ will be probed and merged respectively, as shown in Figure 2.

5.2 Similarity Join on SAI

The inverted lists of $\mathcal{PBI}$ that were constructed for $\mathcal{S}$ are sorted in ascending order of the record's length. In order to avoid probing the inverted lists iteratively, the string collection $\mathcal{R}$ is also sorted by the same order before performing joins. As the $\mathcal{SAI}$ is different to $\mathcal{PBI}$, each inverted list is sorted in ascending order of the $\mathcal{TUB}$ values. In order to improve the efficiency of similarity joins using $\mathcal{SAI}$, the string collection $\mathcal{R}$ is sorted in descending order of the string thresholds. By doing that, we can utilize the $\mathcal{TUB}$ filter when probing the inverted lists.

Definition 3 $\mathcal{TUB}$ **Filter**
For a string $r \in \mathcal{R}$ with threshold $r.\theta$ and prefix token set T_r^p, if $\exists s \in \mathcal{S} \wedge sim(r, s) \geqslant r.\theta$, then $\exists t^{'} \in s \wedge \exists t \in T_r^p \wedge t^{'} = t \wedge \mathcal{TUB}(t^{'}) \geqslant r.\theta$.

The $\mathcal{TUB}$ filter is a necessary condition for two strings to be similar. It can be used to decrease the cost of probing inverted lists. By applying this filter during the join operations, we can just probe the first part of the corresponding inverted lists to get the results. The pseudo-code of similarity join on $\mathcal{SAI}$ is shown in Algorithm 4.

Each string $r \in \mathcal{R}$ is tokenized by the same global ordering $\mathcal{O}$ that was applied to $\mathcal{S}$. Its prefix tokens T_r^p can be acquired based on the number of tokens and its own threshold $r.\theta$ (Lines 2-3). Then, the inverted lists of tokens in T_r^p will be probed and merged. During the processing, two filters are applied. The first is to probe

Algorithm 4: $SimJoinOnSAI(\mathcal{R}, \mathcal{S})$

Input :
$\mathcal{R}$: the string collection $\mathcal{R}$
$\mathcal{S}$: the string collection $\mathcal{S}$
Output:
$< r, s >$: similar pairs

1 **foreach** $r \in \mathcal{R}$ **do**
2 $TokenList$(r) ← Tokenize(r,$\mathcal{O}$);
3 T_r^p ← getPrefixToken(TokenList(r),$r.\theta$);
4 **foreach** $t \in T_r^p$ **do**
5 **while** $i++ < I[t].size$ **do**
6 **if** $r.\theta > I[t].tub$ **then**
7 break;
8 **if** $r.length > \mathcal{S}[I[t].rid].lenght * \theta \;\|\;$
$r.length < \mathcal{S}[I[t].rid].lenght/\theta$ **then**
9 continue;
10 $C.push(\mathcal{S}[I[t].rid])$;
11 Verification(C);

Table 6 Data Sets Information

Data Set	♯ Records	Size	Distribution	Mean	Std
CiteSeer	568,237	69.7M	Normal	0.8078	0.0420
			Poisson	0.6865	0.0481
			Uniform	0.8000	0.1156
DBLP	1,021,062	218M	Normal	0.7774	0.0531
			Poisson	0.7227	0.0676
			Uniform	0.7923	0.1122

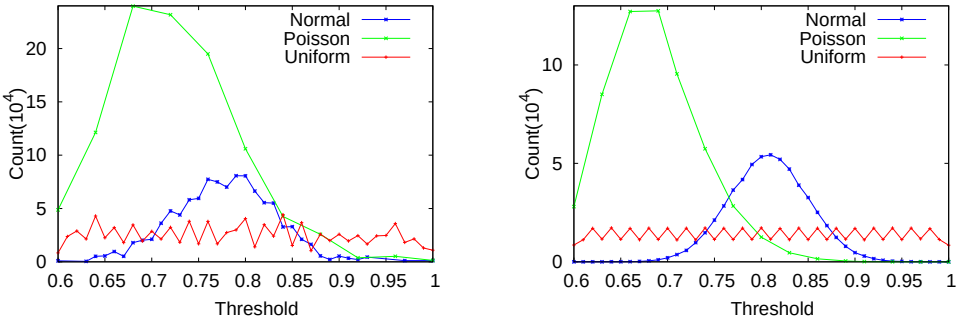
the inverted list until $r.\theta$ is not less than the token's $\mathcal{TUB}$. This filter assures that only the first part of inverted lists are used during probing (Line 6). The second is the length filter. If two strings are similar their length must satisfy a length constraint. If r is similar to string $s \in \mathcal{S}$, the length of r must be in $(s.length * \theta, s.length/\theta)$ (Line 8). Only the strings that pass the two filters are considered as candidates (to be later verified).

6 Experimental Evaluation

We conducted extensive experiments to evaluate the performance and scalability of our techniques to support similarity joins with different similarity thresholds.

6.1 Experiments Setup

We selected two publicly available real data sets of bibliography records from two different data sources for the experiments. They cover a wide range of data distributions and were widely used in previous studies. In order to evaluate the efficiency of our techniques to support similarity join with different similarity



(a) DBLP (b) CiteSeer

Fig. 4 Threshold Distribution Statistic

thresholds, we generated datasets with threshold values that were produced using three different distributions: *Uniform*, *Poisson* and *Normal*.

- **DBLP** is a snapshot of the bibliography records downloaded from the DBLP website[2]. It contains 1,021,062 records, each of which is the concatenation of the author name(s) and the title of a publication. The minimum, maximum, and average length (number of tokens) of the records in this dataset are 2, 207, 13, respectively.
- **CiteSeer** is also a snapshot of the bibliography records downloaded from the CiteSeer website[3]. It contains 568,237 records. Each record is a concatenation of the author names and the title of a publication. The minimum, maximum, and average length of records in this data set are 1, 84, 7, respectively.

Figure 4 and Table 6 show the distribution of the threshold values (θ), the number of records, and the size of the two datasets.

All the experiments were carried out on a single machine with AMD 15x4 cores 1GHz and 60GB main memory. The operating system is CentOS with installed GCC 4.3. The algorithms were implemented in C++ and compiled using GCC with -O3 flag.

The *PPJoin+* [8] is the state-of-art method for set-based string similarity join. However, as it based on prefix filtering, it requires a predefined and unique threshold before performing joins. Furthermore, the index that is used to accelerate the join operations is built for an specific threshold value and will need to be rebuilt for other threshold values. If multiple thresholds are supported, the index cannot be built as in [8] and thus the *PPJoin+* approach cannot be used to perform joins with different similarity thresholds directly. [21] proposed a delta-based-index method, in which the authors grouped the strings based on their lengths and built delta indices for the groups. The delta indices can support similarity search with multiple thresholds. However, it cannot support similarity join with multiple thresholds in one join procedure.

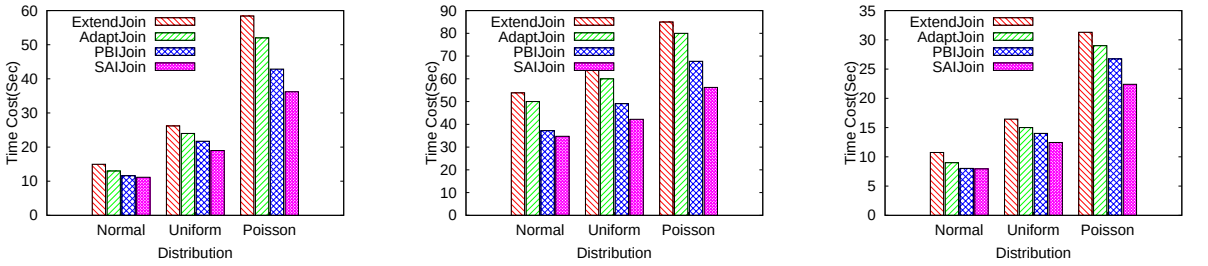
In this paper, we have extended the *PPJoin+* by constructing the index as discussed in Section 1 and modified the inverted lists probing techniques. We denoted the extended *PPJoin+* method as *ExtendedJoin*. We have also constructed the delta index and implemented the *AdaptJoin* approach [21] to support similarity joins in one procedure. In order to verify the efficiency of our proposed methods, we devised two kinds of indices: partition based inverted index and similarity aware inverted index. We denoted the join methods that are based on the two different indices as *PBIJoin* and *SAIJoin*, respectively.

6.2 Comparison on Different Indices

In this section, we evaluate the efficiency of four join methods based on different indices. In this experiments, we used two data sets *CiteSeer* and *DBLP*, each of which has three threshold distributions as shown in Figure 4. We conducted three experiments using the *Jaccard* similarity measure on two datasets under different threshold distributions, including one RS-Join ($CiteSeer \bowtie DBLP$), and two Self-Joins ($DBLP \bowtie DBLP$ and $CiteSeer \bowtie CiteSeer$). We plotted the running time cost of the four methods on each join type in Figure 5.

From the three sub-figures in Figure 5, we can see that *SAIJoin* outperforms *PBIJoin* followed by

[2] http://www.informatik.uni-trier.de/~ley/db
[3] http://citeseerx.ist.psu.edu



(a)CiteSeer $\bowtie_{Jaccard}$ DBLP (b) DBLP $\bowtie_{Jaccard}$ DBLP (c) CiteSeer $\bowtie_{Jaccard}$ CiteSeer

Fig. 5 Comparison On Different Indices

AdaptJoin and *ExtendedJoin*, regardless of the join type and threshold distribution. This is due to the overhead of the indexing structures that are used to accelerate the join operations. The *ExtendedJoin* is based on a straightforward indexing approach where all the tokens of each string are mapped into inverted lists. In order to utilize the length filter and improve the performance of *PPJoin+*, it requires sorting the dataset and the inverted lists by the record length. When applying a predefined and unique threshold, *PPJoin+* can probe the inverted list sequentially, as similar strings must be located nearby each other in inverted lists. However, when applying different thresholds, that list probing technique is not suitable. After sorting by the record length, the strings that are located near each other have similar length but possibly different thresholds. So, their length constraints to identify similar records can be very different. In order to get all possible candidates, each string $r \in \mathcal{R}$ that is joined with $\mathcal{S}$ must probe a large range of the inverted list, not sequentially. This problem can not be avoided in *AdaptJoin* and delta index. So, *AdaptJoin* and *ExtendedJoin* cannot performs as well as the other two methods.

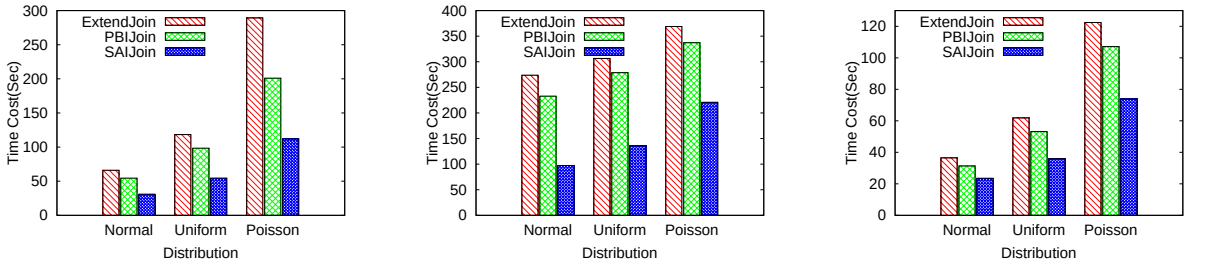
From Figure 5, we can see that *PBIJoin* outperforms *ExtendedJoin* under different experiment conditions. This is due to the advantages of the partition-based inverted index that was applied in *PBIJoin*. Before constructing the partition-based inverted index, a number of values from [0,1] are selected as representative thresholds. These selected representative values are used to partition the inverted index used in *ExtendedJoin* into several small inverted indices as shown in Figure 1. When performing joins, *PBIJoin* will select related parts of the index and probe related inverted lists according to $r.\theta$ and T_r^p. By doing so, much of the unnecessary list probing cost can be avoided.

SAIJoin has the best performance among the three methods. This is due to the similarity-aware inverted index and the $\mathcal{TUB}$ filter. In order to utilize the $\mathcal{SAI}$ index and $\mathcal{TUB}$ filter in an efficient way, the string collections are sorted by their thresholds, and also the inverted lists are sorted by the tokens' $\mathcal{TUB}$ values. This type of string collection arrangement and inverted lists sorting bring the promising results to the head of the inverted lists. For each string $r \in \mathcal{R}$, we can get its threshold $r.\theta$ and prefix token set T_r^p. When performing joins, we just probe the lists $I[t]$ of $t \in T_r^p$ and stop when $r.\theta > t.tup$. This makes *SAIJoin* probe a small part of the related lists and generates the best performance.

6.3 Comparison on Different Similarity Measures

In order to evaluate the scalability and robustness of our proposed methods, we conducted extensive experiments using another widely popular similarity measure, *Cosine*. As in the previous experiments, we used two datasets, *DBLP* and *CiteSeer*, each of which with three different threshold distributions. We carried out three types of joins, one RS-Join and two Self-Joins. The experiment results are plotted in Figure 6.

From the experiment results, we can observe that the *SAIJoin* outperforms *PBIJoin* and *ExtendedJoin*. *SAIJoin*'s execution time is about 50% of the one of *PBIJoin* for most experiments. This is because *ExtendedJoin* and *PBIJoin* apply the framework and filters of *PPJoin+*, which is based on a predefined and unique threshold. The filters are not efficient when the data has different similarity thresholds, especially when using *Cosine*. Under the different similarity thresholds, the $\mathcal{SAI}$ index and $\mathcal{TUB}$ filter achieve high pruning power. So, *SAIJoin* performs well.



(a)CiteSeer $\bowtie_{Cosine}$ DBLP (b) DBLP $\bowtie_{Cosine}$ DBLP (c) CiteSeer $\bowtie_{Cosine}$ CiteSeer

Fig. 6 Comparison Of Cosine Similarity Measure

Observe also that the performance of *SAIJoin* is better with *Cosine* than with *Jaccard* as shown in Figure 5 and Figure 6. This can be explained by the definition of similarity measures shown in Table 5. For the same threshold, the number of prefix tokens of string r is larger when using *Cosine* than that of using *Jaccard*. Since the join method using *Cosine* must probe more inverted lists for each string, it consumes more time than using *Jaccard*.

6.4 Comparison on Different Distributions of Threshold

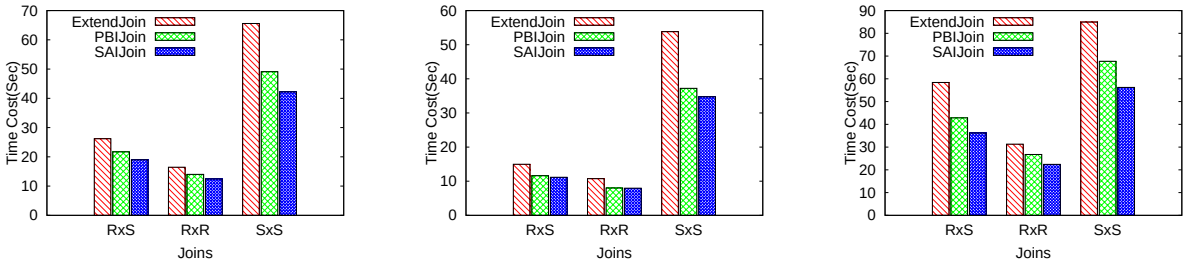
In this experiment, we verify the efficiency and robustness of our methods on different datasets with different thresholds. We conducted experiments on two datasets and performed three different join operations using *Jaccard*. The results are shown in Figure 7. In this figure, R represents *CiteSeer* and S represents *DBLP* for simplicity, and X denotes the join operation.

From Figure 7, we can observe that the *SAIJoin* outperforms *PBIJoin* by a wide margin followed by *ExtendedJoin* regardless of the kind of distribution. Before performing joins, *PBIJoin* and *ExtendedJoin* sorted the string collections by the string length. By doing that, the collections that are being joined can be scanned sequentially by applying length constraints. However, since the strings have different thresholds, the strings that are located as neighbors by the sorting step can have different length constraints. So, the join operation cannot be performed as *PPJoin* using the length filter on the sorted datasets. *PBIJoin* and *ExtendedJoin* must probe a range of inverted lists to get the exact results. While, *SAIJoin* performs joins using $\mathcal{SAI}$ index and applies the $\mathcal{TUB}$ filter to ensure that only a small section of the inverted lists are probed. So, it can get high performance under different threshold distributions.

For all the three join methods, they perform best on the dataset whose threshold distribution is *Normal*, followed by *Uniform* and *Poisson*. This phenomenon can be observed in Figure 5, Figure 6 and Figure 7. This can be explained using the threshold distribution features. For the DBLP dataset, when the threshold distribution is *Normal*, the average of threshold values is 0.8078 and std is 0.0420. This is, the threshold distribution is more compact than in the other two cases and the average threshold value is higher then the ones of the other two distributions. Higher threshold values generate smaller number of prefix tokens and less time on probing the inverted list. Consequently, the join operation takes less time with *Normal* distribution than with the other two distributions. When the distribution is *Uniform*, the threshold values are uniformly distributed. The average thresholds is 0.8000, while the std is 0.1156. This is, the threshold values varied significantly. For the *Poisson* distribution, the average threshold value is 0.6865 and the std is 0.0481. The threshold values are distributed more compactly and most of them are small threshold values. Since small threshold values generate large number of prefix tokens, the join operations must probe more inverted lists. Thus, all the three join methods spent more time on the dataset with *Poisson* distribution.

7 Conclusions

In this paper, we have studied the problem of similarity joins with different similarity thresholds. We devised two new indexing techniques that can support different thresholds to process join operations. Although the proposed $\mathcal{PBI}$ index reduces the unnecessary probing cost in comparison with the straightforward approach,



(a)Uniform (b) Normal (c) Poisson

Fig. 7 Comparison On Different Threshold Distribution

there is still much space for improvement.

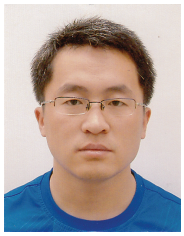
In order to improve the performance, we proposed the $\mathcal{SAI}$ index and a new index probing technique. The experimental results on real-world datasets show that our proposed solution can support different similarity thresholds efficiently.

For our future work, we plan to address the problem of assigning suitable similarity thresholds to different strings.

Acknowledgements This material is based upon work supported by the National Natural Science Foundation of China under grant No.61402329 and No.51378350 .

References

1. Monge A, Elkan C. The field matching problem: Algorithms and applications. In: SIGKDD. 1996, 267–270
2. Zhang Z, Hadjieleftheriou M, Ooi B, Srivastava D. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In: SIGMOD. 2010, 915–926
3. Lu W, Du X, Hadjieleftheriou M, Ooi B C. Efficiently supporting edit distance based string similarity search using b+-trees. TKDE, 2014, 26(12): 2983–2996
4. Wang J, Feng J, Li G. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. VLDB, 2010, 3(1-2): 1219–1230
5. Sarawagi S, Kirpal A. Efficient set joins on similarity predicates. In: SIGMOD. 2004, 743–754
6. Chaudhuri S, Ganti V, Kaushik R. A primitive operator for similarity joins in data cleaning. In: ICDE. 2006, 61–72
7. Bayardo R, Ma Y, Srikant R. Scaling up all pairs similarity search. In: WWW. 2007, 131–140
8. Xiao C, Wang W, Lin X, Yu J. Efficient similarity joins for near duplicate detection. In: WWW. 2008, 131–140
9. Hernández M, Stolfo S. The merge/purge problem for large databases. In: SIGMOD. 1995, 127–138
10. Winkler W. The state of record linkage and current research problems. In: Statistical Research Division. 1999
11. Sivic J, Zisserman A. Video google: A text retrieval approach to object matching in videos. In: Computer Vision. 2003, 1470–1477
12. Dong X, Halevy A, Madhavan J. Reference reconciliation in complex information spaces. In: SIGMOD. 2005, 85–96
13. Sarawagi S, Bhamidipaty A. Interactive deduplication using active learning. In: SIGKDD. 2002, 269–278
14. Arasu A, Ré C, Suciu D. Large-scale deduplication with constraints using dedupalog. In: ICDE. 2009, 952–963
15. Gravano L, Ipeirotis P, Jagadish H, Koudas N, Muthukrishnan S, Srivastava D. Approximate string joins in a database (almost) for free. In: VLDB. 2001, 491–500
16. Elmagarmid A, Ipeirotis P, Verykios V. Duplicate record detection: A survey. TKDE, 2007, 19(1): 1–16
17. Naumann F, Herschel M. An Introduction to Duplicate Detection. Synthesis Lectures on Data Management, 2010, 2(1): 1–87
18. Jiang Y, Li G, Feng J, Li W S. String similarity joins: An experimental evaluation. In: PVLDB. 2014, 625–636
19. Chaudhuri S, Kaushik R. Extending autocompletion to tolerate errors. In: SIGMOD. 2009, 707–718
20. Deng D, Li G, Feng J. A pivotal prefix based filtering algorithm for string similarity search. In: SIGMOD. 2014, 673–684
21. Wang J, Li G, Feng J. Can we beat the prefix filtering? an adaptive framework for similarity join and search. In: SIGMOD. 2012, 85–96
22. Rong C, Lu W, Wang X, Du X, Chen Y, Tung A K. Efficient and scalable processing of string similarity join. TKDE, 2013, 25(10): 2217–2230
23. Lu J, Lin C, Wang W, Li C, Wang H. String similarity measures and joins with synonyms. In: SIGMOD. 2013, 373–384
24. Li G, He J, Deng D, Li J. Efficient similarity join and search on multi-attribute data. In: SIGMOD. 2015, 1137–1151
25. Salton G, McGill M. Introduction to modern information retrieval. McGraw-Hill, Inc., 1986
26. Witten I H, Moffat A, Bell T C. Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition. Morgan Kaufmann, 1999



Chuitian Rong is an associate professor at Tianjin Polytechnic University. He received his Ph.D. degree from Renmin University of China in 2013. His research interests are database system, information retrieval, big data analysis.

Yasin N. Silva is an Assistant Professor of Applied Computing in the School of Mathematical & Natural Sciences at Arizona State University. He received his Ph.D. (2010) and M.S. (2006) in Computer Science from Purdue University and his B.S. in Computer Engineering from the Pontificia Universidad Catolica, Peru (2000). Yasin's research areas deal with data management systems and privacy preservation in general. More specifically, he has been working on the areas of query processing and optimization, privacy assurance in database systems, Big Data management systems, scientific database systems, and the integration of new data processing technologies into the computing curricula.

Chunqing Li is a professor at Tianjin Polytechnic University. His research interests are database system and applications, big data analysis, network management and applications.