Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 14, no. 2, February 2018, pp. 245-258
DOI: 10.23940/ijpe.18.02.p6.245258

A Survey on Set Similarity Search and Join

Lianyin Jia[a], Lulu Zhang[a], Guoxian Yu[b], Jinguo You[a], Jiaman Ding[a], Mengjuan Li[c,*]

[a]*Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming, 650500, China*
[b]*College of Computer and Information Science, Southwest University, Chongqing, 400715, China*
[c]*Library, Yunnan Normal University, Kunming, 650500, China*

Abstract

Set similarity search and join operations have a wide range of applications, including e-commerce, information retrieval, bioinformatics, and so on. Although extensive techniques have been suggested for set similarity search and join, there is literature on systematically categorizing these techniques and comprehensively comparing them. To bridge this gap, this paper provides a comprehensive survey of set similarity search and join. The survey starts from the basic definitions, framework, ordering and similarity functions of set similarity search and join. Next, it discusses the main filtering techniques used in the state-of-the-art algorithms, and analyses the pros and cons of these algorithms in detail. For set similarity join algorithms, we divide them into 2 main categories based on the key underlying techniques they use: prefix filtering based algorithms and partition based algorithms. Prefix filtering is the most dominant technique, so algorithms based on prefix filtering and their recent variants are analyzed thoroughly. Partition is also a promising technique as it can exploit comparisons between multiple partitions. Furthermore, some efforts on MapReduce based set similarity join algorithms are also discussed briefly. In the end, the open challenges and questions are presented for future pursue.

Keywords: set similarity search; set similarity join; prefix filtering; partition; similarity threshold; set-valued index

(Submitted on November 5, 2017; Revised on December 13, 2017; Accepted on January 17, 2018)

1. Introduction

Set similarity search (SSS) and set similarity join (SSJ) aim to find all similar sets for a set or all similar set pairs in a collection of sets. They have been used in a large variety of applications. The possible applications in these fields include: recommendation system [8,36], near duplication detection and deletion [16,25,42], query auto-completion [2,49], entity resolution [47], document clustering [53], plagiarism detection [6], data cleaning [7], social network mining [41] and gene sequences comparison [45], etc.

Taking the recommendation systems for example, recommendation systems need to recommend goods for users with similar interests, or divide users into groups to make precision marketing. SSS and SSJ can be resorted to efficiently accomplish these recommending tasks.

In information retrieval (IR), IR engines have to extract and index huge amounts of web pages. Due to site mirror, information reproduction and multi-version, extracted pages may contain a large number of duplicated or near duplicated records, thus increasing the overhead of indexes and degrading the performances. These duplicated or near duplicated pages can be efficiently cleaned by SSJ algorithms.

Gene sequence comparisons are important tasks in bioinformatics [45]. We need to estimate the similarity of two gene sequences and discover whether they are homologous or derived from a common ancestor gene. With the exponentially growth of gene sequences, manually comparing is unfeasible. SSJ algorithms can help to do these jobs much faster.

* Corresponding author.
E-mail address: lmjlykm@163.com

Efficient set indexes should be designed to speed up SSS and SSJ, since it is not feasible to enumerate and compute similarities for every two sets in massive sets. Earlier researches in set containment search and join have designed lots of set indexes, such as sequential signature file [19], hashing index [14], bitmap index [22], inverted index [15], and so on. But, there are obvious differences between set containment and set similarity. Set containment focuses on whether a set contains other sets or is completely contained in another set, whereas set similarity tries to find the containment degree. So, most indexes for set containment are not suitable or not efficient for set similarity.

There exist some works close to ours. Mann *et al*. [30] gave an empirical evaluation of some SSJ algorithms, but these evaluated algorithms are all prefix filtering based SSJ algorithms. SSS algorithms and very recently partition based SSJ algorithms are omitted in their work. Yu *et al*. [51] and Wandelt *et al*. [44] survey the works on string similarity. However, string similarity is different from set similarity for their different working manner and similarity functions [44].

Though there are lots of efforts in these fields, comprehensive classification and comparisons of these efforts are almost untouched. It is necessary to classify and compare SSS and SSJ algorithms in detail based on key techniques they used, and thus to differentiate the essence of different algorithms. For this purpose, this paper summarizes a large range of SSS and SSJ algorithms and discusses the research progresses and problems faced with them.

The remainder of this paper is organized as follows. Section 2 gives an overview about SSS and SSJ. Section 3 presents SSS algorithms and section 4 divides SSJ algorithms into prefix based algorithms and partition based algorithm, and then discusses them in detail. Section 5 introduces MapReduce based parallel SSJ algorithms and discusses some recent efforts on MapReduce platform. Section 6 summarizes the challenges in these fields. The final section concludes the paper and suggests areas for future research.

2. Brief overview of SSS and SSJ

2.1. Related definitions

A dataset R is defined as a collection of sets, and each set $r \in R$ has a unique ID ranging from 0, 1, …, $|R|-1$. The elements of a set are taken from a finite universe denoted as U. We use $|U|$ to represent the number of distinct elements in U and $|R|$ to represent the number of sets in the dataset. The basic definitions of SSS and SSJ are listed as follows.

Definition 1. *SSS*. Given a query s, finding all sets $r \in R$ with similarity between r and s qualifying threshold τ.

Definition 2. *SSJ*. Given two datasets R and S, finding all pairs $<r,s>$ ($r \in R$ and $s \in S$) with similarity between r and s qualifying threshold τ.

Note that, for Definition 1 and 2, "**qualifying threshold** τ" means differently for different similarity functions. For T-Overlap, Jaccard, Cosine and Dice, it means "not greater than τ", whereas it means "not less than τ" for Hamming Distance. The detailed description of these similarity functions will be given in subsection C of this section.

Specifically, for SSJ, we call a join self-join if $R = S$ holds, otherwise we call it R-S join. In this paper, we mainly focus on self-join as discussed in most SSJ algorithms.

2.2. Orderings

Ordering is necessary for most algorithms mentioned blow, as many algorithms use some certain orderings to reach their peak performances. There are two kinds of ordering in general: set ordering and element ordering.

Length ordering (LO) is the most common set ordering; it sorts sets by their lengths and is extensively used in most subsequent algorithms.

There are 3 common element orderings: dictionary ordering (DO), frequency ordering (FO), and enumeration ordering (EO). DO sorts elements of a set based on their lexicalorders. FO sorts elements of a set based on their frequency in the underlying dataset. EO firstly generates an ordering by enumerating universe U and then sorting elements of a set by the ordering.

In addition, set ordering can be combined with element ordering and generate much more orderings, such as LO_A-FO_D, LO_A-DO_D, where the subscripts A and D mean ascending and descending, respectively.

2.3. Similarity Functions

The degree of similarity between two sets can be measured by a particular metric function. Many functions can be used, such as T-overlap, Jaccard, Cosine, Dice, Hamming Distance as shown in Equations (1)-(5).

$$T-overlap(r,s)=|r\cap s| \tag{1}$$

$$Jac(r,s)=\frac{|r\cap s|}{|r\cup s|}=\frac{|r\cap s|}{|r|+|s|-|r\cap s|} \tag{2}$$

$$Cos(r,s)=\frac{|r\cap s|}{\sqrt{|r|*|s|}} \tag{3}$$

$$Dice(r,s)=\frac{2*|r\cap s|}{|r|+|s|} \tag{4}$$

$$Ham(r,s)=|r\cup s|-|r\cap s|=|r|+|s|-2*|r\cap s| \tag{5}$$

Note that the numeric values of T-overlap and Hamming Distance are natural numbers, whereas they are float numbers for the other three functions.

Also note that, different from the other 4 functions, the similarity decreases as Hamming Distance increases. In other words, the bigger the Hamming Distance between two sets is, the more dissimilar the two are.

We now state some important lemmas related to these functions, which will be used in the later discussions.

Lemma 1. Minimum Overlap of a set. Consider a set s and a similarity threshold τ. Any sets with common elements less than τ, $\tau|s|$, $\tau^2|s|$ and $\frac{\tau}{2-\tau}|s|$ for T-overlap, Jaccard, Cosine, and Dice respectively or greater than $|s|-\tau$ for Hamming Distance cannot be similar with s for threshold τ.

The minimum overlap of a set s for a certain threshold τ, denoted as $minover(s)$ [1], is the minimum number of possible common elements r should have with s for a certain threshold τ.

Lemma 2. Length Bound. Consider a set s and a similarity threshold τ. Any sets with length smaller than the lower length bound or bigger than upper length bound of s cannot be similar to s for a certain threshold τ. The lower length bounds and upper length bounds for T-overlap, Jaccard, Cosine, Dice, Hamming Distance are $[\tau,\infty)$, $\left[\tau|s|,\frac{|s|}{\tau}\right]$, $\left[\tau^2|s|,\frac{|s|}{\tau^2}\right]$, $\left[\frac{\tau}{2-\tau}|s|,\frac{2-\tau}{\tau}|s|\right]$, $[|s|-\tau,|s|+\tau]$ respectively.

The upper length bound and lower length bound of s are denoted as $s_{\min}$ and $s_{\max}$, respectively.

Specifically, if $|r|$ and $|s|$ are known beforehand, we can compute a stricter minimum overlap, $\mathrm{minOver}(r,s)$, as shown in Lemma 3.

Lemma 3. Minimum Overlap of a pair of sets. Consider two sets r, s and a similarity threshold τ. If the common elements of these two sets are less than τ, $\frac{\tau}{1+\tau}(|r|+|s|)$, $\tau\sqrt{|r|*|s|}$, and $\frac{\tau}{2}(|r|+|s|)$ for T-overlap, Jaccard, Cosine and Dice respectively or greater than $\frac{|r|+|s|-\tau}{2}$ for Hamming Distance, the two sets cannot be similar for a certain threshold τ.

[1] For ease of description, we assume that threshold τ always exists.

Minimum overlap and length bound play important roles for the follow-up filtering techniques. Table 1 shows the comparisons between different similarity functions.

Table 1. Comparisons of different similarity functions

Function	Similarity	MinOverlap(s)	MinOverlap(r,s)	Length bound
T-Overlap	$\geq \tau$	τ	τ	$[\tau, \infty)$
Jaccard	$\geq \tau$	$\tau \lvert s \rvert$	$\frac{\tau}{1+\tau}(\lvert r \rvert + \lvert s \rvert)$	$\left[\tau \lvert s \rvert, \frac{\lvert s \rvert}{\tau}\right]$
Cosine	$\geq \tau$	$\tau^2 \lvert s \rvert$	$\tau\sqrt{\lvert r \rvert * \lvert s \rvert}$	$\left[\tau^2 \lvert s \rvert, \frac{\lvert s \rvert}{\tau^2}\right]$
Dice	$\geq \tau$	$\frac{\tau}{2-\tau}\lvert s \rvert$	$\frac{\tau}{2}(\lvert r \rvert + \lvert s \rvert)$	$\left[\frac{\tau}{2-\tau}\lvert s \rvert, \frac{2-\tau}{\tau}\lvert s \rvert\right]$
Hamming	$\leq \tau$	$\lvert s \rvert - \tau$	$\frac{\lvert r \rvert + \lvert s \rvert - \tau}{2}$	$[\lvert s \rvert - \tau, \lvert s \rvert + \tau]$

2.4. Filtering Techniques

Before discussing the algorithms, we first give the relevant lemmas of major filtering techniques.

Lemma 4. PreFilter. If the $|r| - \lceil \min Over(r) \rceil + 1$ prefixes of r and the $|s| - \lceil \min Over(s) \rceil + 1$ prefixes of s have no common element, r and s cannot have a similarity greater than τ.

Lemma 5. LenFilter. If $|r| \leq s_{\min}$ or $|r| \geq s_{\max}$, r and s cannot have a similarity greater than τ.

Lemma 6. PosFilter.Assume e is the *i*-th element in r and the *j*-th element in s, if $O_l + \min(|r| - i, |s| - j)$ is smaller than $\min Over(r, s)$, r and s cannot have a similarity greater than τ, where O_l represents the number of common elements before e (including e) in r and s.

Figure 1 gives a sketch description of PreFilter. The main idea behind PreFilter is to map threshold τ to constraints on set prefixes. PreFilter can be used to filter sets having no common elements with the query s. In the latter description, we denote $pref(s)$ as the prefix of s. LenFilter can be used to filter sets with length out of certain range. PosFilter can filter sets with position difference of common element greater than a certain value.

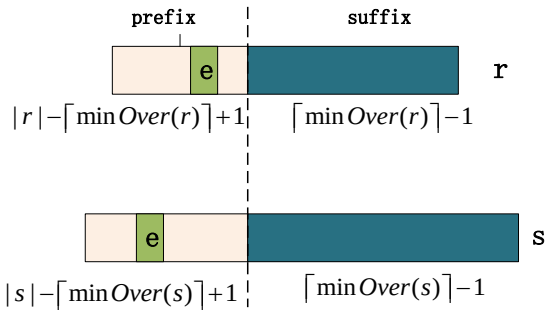

Figure 1. Prefix filtering

2.5. Filtering Techniques

Algorithm 1 gives the general frameworks of SSS. SSS consists of two independent phases: index creation and similarity search. In index creation phase, we pre-process datasets and index the elements of each set. In the similarity search phase, we execute searches on created indexes, and then generate and verify candidate sets.

CreateIndex Phase
CreateIndex(R)
//Input: R , a dataset
//Output: an index I
1. Preprocess R
2. for each r in R
3. Index the elements of r to I

SSS Phase
SSS(s, τ)
//Input: s ,a query set.
τ ,a similarity threshold.
//Output: all IDs satisfying threshold with s no less than τ
1. Retrieve s in index I and generate candidates
2. Verify candidates and get final results

Algorithm 1. The general framework of SSS

Algorithm 2 gives the general frameworks of SSJ. SSJ only has a single phase that integrates index probing and index creation together.

SSJPhase
SSJ(R, τ)
//Input: R , a dataset
τ , a similarity threshold.
//Output: all ID pairs satisfying threshold with s no less than τ
1. I$\leftarrow \phi$
1. preprocess R
2. for each set s in R
3. query s in index I and get candidate pairs < r , s >
4. index partial elements of s to I
5. verify candidate pairs and get final results

Algorithm 2. The general framework of SSJ

SSS and SSJ are much the same, but they also have the following 3 obvious differences.

- *Execution process*: SSS has two independent phases whereas SSJ has only one integrated phase.
- *Index*: SSS usually indexes all elements of a set statically, whereas SSJ indexes partial elements of a set dynamically.
- *Threshold*: Indexes created for SSS should support queries with different thresholds. As for SSJ, a single join needs only to support a certain threshold.

Algorithms efficient to SSJ do not necessary work for SSS, but SSS algorithms can be used to realize SSJ by nested loop join. SSJ creates dynamic indexes on partial data. The index size is much smaller, so its efficiency is higher than nested loop joins based algorithms.

3. Research Progress on SSS

SSS is a well-studied subject and indexes are vital to existing SSS algorithms, so most existing researches are devoted to designing efficient indexes. In this paper, we divide the state-of-the-art SSS algorithms into inverted index based and trie-based algorithms.

3.1. Inverted index based Algorithms

In addition to being used in information retrieval, inverted indexes have also been widely used in set similarity related tasks. Most existing SSS algorithms are inverted index based and work for T-overlap search.

Sarawagi *et al.* proposed ProbeCount [37] to solve T-Overlap search by creating heap for frontiers of all relevant lists and by counting the occurrences of each element.

In order to eliminate the expensive heap operations, ScanCount [26] is proposed. ScanCount uses an array to count the occurrences of each element, so it has a higher efficiency than ProbeCount.

Both ProbeCount and ScanCount need to scan all *ID*s in relevant lists, and thus have high complexities. To solve this problem, MergeOpt [37], the first algorithms adopting the idea of prefix filtering, was proposed. Its main idea is to divide the inverted lists into $|s|-T+1$ shorter lists and $T-1$ longer lists. Each *ID* satisfying threshold *T* should appear in at least one shorter list. *ID*s only appearing in longer lists can be pruned safely.

To decrease the number of accessed *ID*s, MergeSkip [26] introduced a list-skip strategy to skip unnecessary *ID*s and significantly improve the efficiency.

SSS algorithms, more specifically T-overlap search algorithms, can get final results directly by scanning the created indexes, so they can avoid the overhead of verifying candidates commonly used in prefix filtering based and partition based algorithms, which will be discussed later.

3.2. Trie based Algorithms

When there is a lot of duplicated or near duplicated sets in the underlying dataset, the inverted index based algorithms turn to be rather inefficient. To solve this problem, Jia *et al.* [20] designed an Expanded Trie Indexes (ETI) structure to map common prefixes of different sets to a common trie path. Based on ETI, T-Similarity algorithm was proposed to convert the T-Overlap search problem into a problem of finding query nodes with query depth equal to *T*. Approach in [23] uses a compact trie to decrease the memory consumption. Table 2 gives brief comparisons of SSS algorithms.

Table 2. Comparisons of SSS algorithms

Algorithms	Index	Operations	Ordering	Advantages	Disadvantages
ProbeCount [21]	Inverted index	T-Overlap search	DO for *ID*s in list	Heap based list scan	Inefficient for list scan and heap operations
ScanCount [22]	Inverted index	T-Overlap search	#NA	Lists scan	Inefficient for list scan
MergeSkip [22]	Inverted index	T-Overlap search	DO for *ID*s in list	Heap based list scan, list-skip	Inefficient for lower threshold
MergeOpt [21]	Inverted index	T-Overlap search	List length ascending order	List PreFilter	Inefficient for lower threshold
T-Similarity [23]	Expanded trie index	T-Overlap search	FO_D	Find T- Nodes	High memory overhead for sparse dataset

4. Research progress on SSJ

In recent years, most work has focused on SSJ and many algorithms have been proposed. To facilitate discussion, we divide the state-of-the-art algorithms into 2 categories based on key techniques they use: prefix filtering based and partition based algorithms.

4.1. Prefix Filtering based Algorithms

Most prefix filtering based algorithms adopt a filtering-and-verification framework to generate candidates in the filtering phase and verify candidates in the verification phase.

The number of candidates has an important impact on the performance: too large a number of candidates increases the verification overhead, thus deteriorating performance. To decrease the number of candidates, length filtering [1,13] (LenFilter), prefix filtering [7,37] (PreFilter), positional filtering [13,50] (PosFilter), suffix filtering [50] (SufFilter) and many other filtering techniques have been proposed. Among these techniques, PreFilter is the dominant one. Algorithms based on

PreFilter or its variants combined with other filtering techniques are commonly used in SSS and SSJ. So, we first present main filtering techniques and then discuss the PreFilter based algorithms. In the end, we present some improvements on PreFilter.

4.1.1. Algorithms

Most PreFiltering based algorithms are SSJ algorithms. The execution processes of these algorithms include 4 main phases: dataset preprocessing, index probe, index creation and candidate verification. Figure 2 gives a brief framework on these algorithms.

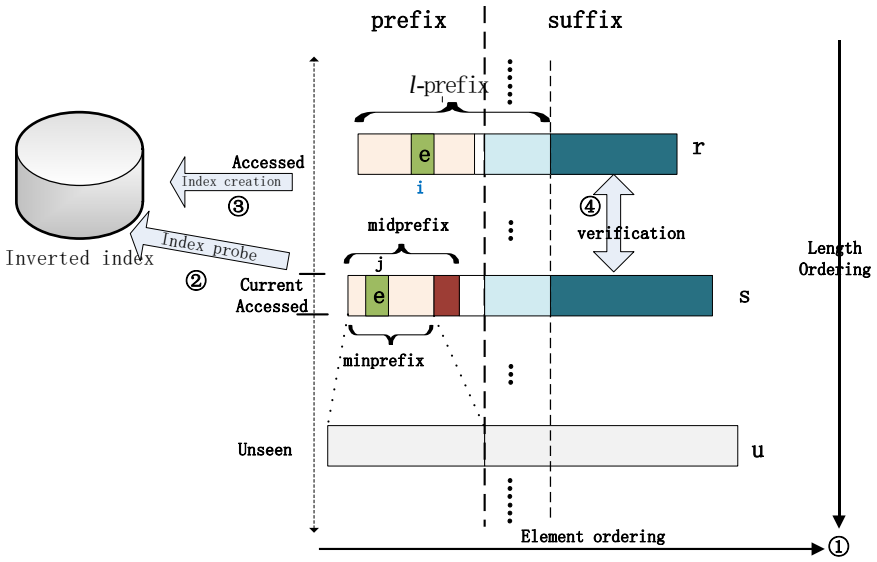

Figure 2. Brief framework of Prefix filtering based SSJ algorithms

As opposed to MergeOpt, which uses PreFilter to filter inverted lists, most algorithms use PreFilter to filter sets. SSJoin [7] first applies PreFilter to filter sets and develops a similarity primitive operator in data cleaning. This operator creates and probes index only on prefixes of sets. AllPair [3] sorts dataset in LO_A and combines LenFilter with PreFilter, thus increasing the pruning power of PreFilter. To further decrease the number of candidates, PPJoin [50] extends AllPair by adding PosFilter and FO_A. The indexed prefixes of s can be decreased to midprefix (the first $|s|-minOver(s,s)+1$ elements in s) for PPJoin by exploiting LO_A. Experiments on dataset DBLP show that the candidates of PPJoin are only one fifth of AllPair with Jaccard threshold of 0.8.

Other than prefixes, sets suffixes also contains abundant information and can be used to further filter candidates. Motivated by this observation, PPJoin+ [50] extends PPJoin by adding SufFilter, which uses divide-and-conquer policy to detect whether the hamming distance of two suffixes exceeds the computed threshold. Experiments on DBLP show the candidates are substantially reduced and only 2% candidates survived, compared with PPJoin with Jaccard threshold of 0.8.

4.1.2. Improvements on PreFilter

The prefix lengths have an important effect on filtering performance. Shorter prefix lengths tend to have good filtering performances, but lead to longer verification times. On the contrary, longer prefix lengths tend to have good pruning power, but tend to expense more filtering time. So PreFilter is not always the best choice and needs to be further optimized. We discuss the improvements on PreFilter from the following 4 aspects.

- PreFilter with reduced prefixes

MPJoin [32] proposes the concept of minprefix. The minprefix of s relative to u is $|s|-minOver(s,u)+1$, which is a dynamic concept and decreases with the increase of $|u|$. Based on this concept, MPJoin only probes the lists relevant to minprefix of s by deleting unnecessary index entries.

- PreFilter with extended prefixes

Wang *et al.* [48] proposed a concept of *l*-prefix that if r and s are similar, the $|r|-\lceil \min Over(r) \rceil + l$ prefixes of r and $|s|-\lceil \min Over(s) \rceil + l$ prefixes of s must have l elements in common. Different sets have a different optimal l, and they

designed a cost model and proposed a framework named AdaptPrefixScheme to choose the optimal *l*, thus to keep a tradeoff between filtering cost and verification cost. To efficiently generate candidates, an index named Delta inverted index is incrementally created on *l*-prefix. Based on Delta inverted index, AdaptJoin and AdaptSearch have been proposed to tackle SSJ and SSS, respectively.

- PreFilter with grouped prefixes

Bouros *et al.* [4] introduced the GPJoin algorithm based on the observation that different sets may have identical prefixes and can be grouped together. We can execute filtering on these prefixes in a batch manner. The main shortcoming of GPJoin is the extra overheads during verification phase to unfold sets in a group.

- PreFilter with Multiple prefix filtering

Unlike most PreFilter based algorithms, which generate a prefix for a set, MGJoin [34] generates multiple prefixes for a set using different EO. By sequentially applying PreFilter on these prefixes, the final candidates can be decreased.

The summary of PreFilter based algorithms is shown in Table 3. In summary, PreFilter is the dominant technique in SSS and SSJ, and plenty of efforts have been made in this field. But, there are still many avenues for future study:

1) Develop new filtering techniques to combine with existing techniques to further decrease generated candidates.

2) As shown in [30], AllPair wins on large datasets. We can draw a conclusion that filtering techniques are not "the more the better". We can move toward how to choose appropriate filtering techniques and how to choose an optimal execution order, and thus balance the filtering cost and verification cost.

Table 3. Comparisons of prefix filtering based algorithms

Algorithms	Index	Functions	Ordering	Advantages	Disadvantages
MergeOpt [21]	Inverted index on entire dataset	T-Overlap	List length ascending order	List PreFilter	Inefficient for lower threshold
SSJoin [11]	Inverted index on prefix	SSJ	none	PreFilter	Inefficient for a large number of candidates
AllPair [28]	Inverted index on prefix	SSJ	LO_A	PreFilter, LenFilter	
PPJoin [27]	Inverted index on midprefix	SSJ	LO_A-FO_A	PreFilter,LenFilter , PosFilter	Higher filtering costs
PPJoin+ [27]	Inverted index on midprefix	SSJ	LO_A-FO_A	PreFilter,LenFilter , PosFilter,SufFilter	Slow SufPrefilter
MPJoin [29]	midprefix with index removal	SSJ	LO_A-FO_A	minprefix	Higher filtering costs
AdaptJoin [30]	Deltainverted index for *l*-prefix	SSJ	LO_A-FO_A	*l*-prefix	Inefficient for zipf distribution datasets
GPJoin [31]	Inverted index on midprefix	SSJ	LO_A-FO_A	Grouped PreFilter	Inefficient when there are few duplicate prefixes
MGJoin [32]	Inverted index on prefix	SSJ	Multiple EOs	Multiple PreFilter	Extra time and space costs for Multiple EOs

4.2. Partition based Algorithms

The main idea behind partition based algorithms is to map similarity of two sets to constraints on their partitions. If two sets satisfy similarity threshold, then partitions of a set must satisfy certain constraints with partitions of the other set. Note that partitions satisfying the constraints do not mean the two sets are similar. So, similar to PreFilter based algorithms, partition based algorithms also adopt the filtering-and-verification framework.

How to select a suitable partition schema is a key task for partition based algorithms. For SSS and SSJ, we should investigate carefully how to partition to avoid destroying the constraints between partitions. In this paper, we focus on two partition strategies: equal-length partition (ELPart) and equal-number partition (ENPart).

4.2.1. ELPart

ELPart splits a set into multiple partitions with equal length (except for the last partition). In this case, the same element in different sets may be assigned to partitions with different partition number. As a result, one partition of s may overlap with

multiple partitions of r. Figure 3 gives an example ELPart partition schema with partition length 4. From Figure 3, we can see that element 6 is assigned in different partitions (partition 0 of r_1 and r_2 and partition 1 of r_3).

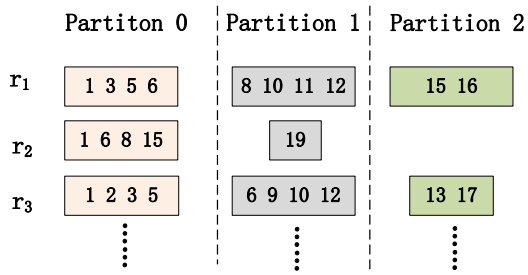

Figure 3. ELPart partition schema

Rong *et al.* [35] proposed BFJoin, an ELPart based algorithm, which splits s into $|s|-\lceil \min Over(s) \rceil+1$ partitions. If r and s are similar, then at least one partition of s must be a subset of r. The main drawback of BFJoin is to detect whether a partition is a subset of another set, which is a set containment problem with high computational complexity.

4.2.2. ENPart

Different from ELPart, ENPart splits each set into equal number of partitions. It first maps similarity threshold τ to partition threshold k and then evenly divides the universe Ω into $k+1$ partitions with size $l=\lceil |\Omega|/(k+1) \rceil$, and then each element x of a set is mapped into the corresponding partition with partition number $\lceil x/l \rceil$. If the similarity of r and s satisfies τ, there must be at least one partition of r and one partition of s exactly the same. For ENPart, an element in different sets is assigned to partitions with the same partition ID, so the *i*-th partition of r only possibly covers with the *i*-th partition of s. As a result, set containment detection can be avoided and a much better performance can be expected. Note that the *i*-th partitions of r and s do not necessarily have the same number of elements. Figure 4 shows an example of ENPart partition schema with $l=8$. In Figure 4, element 6 is in partition 0 of all sets.

PartEnum [1] splits sets into $n_1 * n_2$ partitions using a two level ENPart schema (n_1 for level 1 and n_2 for each of level 1 partition). It combines partition and enumeration together, and transforms a larger hamming distance similarity problem into a smaller hamming distance similarity problem on a smaller partition. The main drawback of PartEnum lies in that it needs to enumerate a large number of partition signatures, thus downgrading the efficiency.

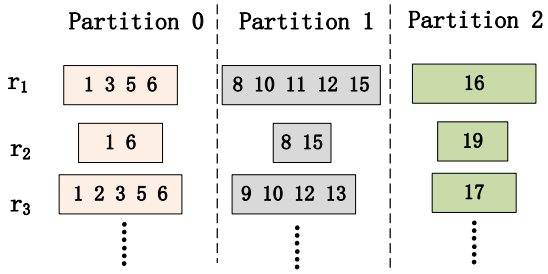

Figure 4. ENPart partition schema

To avoid the overheads of partition enumeration, Deng *et al.* [10] suggested a symmetric difference based partition SSJ algorithm (PartJoin). PartJoin uses ENPart to split sets. The theoretical upper bound of the symmetric difference of r and s is $k=|s \Delta r| \le \frac{1-\tau}{\tau}|s|$ for Jaccard function. When splitting each set into $k+1$ partitions, at least a partition of r equals to a partition of s if r and s are similar. An inverted index for partitions is then created by PartJoin to retrieve candidates. To further improve the filtering power, 1-deleted neighborhood index is designed to impose multi-partitions filtering strategy on partitions of r and s. Consequently, PartJoin can significantly reduce the number of candidates.

To conclude, compared with ELPart, ENPart is more suitable for SSJ, since it doesn't need subset detection. By using

inverted index and 1-deleted neighborhood index, PartJoin has much a better filtering power than BFJoin and PartEnum. However, for PartJoin, in extreme cases, there will be a large number of empty partitions or partitions with very few elements, which may result in a worse performance. Table 4 gives the comparisons of three partition based algorithms.

Table 4. Comparisons of partition based algorithms

Algorithms	Partition Strategies	Index	Advantages	Disadvantages
PartEnum [26]	Two level ENPart + enumeration	signature	ENPart	Inefficient for numeration
BFJoin [33]	ELPart + subset detection	Inverted index on prefix	ituitive	Inefficient for set containment detection
PartJoin [34]	ENPart + multi partition filtering	Partition inverted index, 1-deletion neighborhood index	Efficient for ENPart and multi partition filtering	Power law distribution of partition size

In summary, for PreFilter based algorithms, they filter sets with no common prefix elements with the current query set and index only the prefix of each set, which makes these algorithms rather efficient. The main drawback of PreFilter based algorithms lies in that their efficiencies are greatly affected by the number of candidates. Partition based algorithms split each set into a certain number of partitions and convert constraints between two sets into their corresponding partitions. They are efficient by exploiting comparisons between multiple smaller partitions, but inefficient when the size of the partitions is uneven (for ENPart) or complex subset check operations are needed (for ELPart).

Note that the algorithms discussed in section 3 and 4 are all exact SSS or SSJ algorithms. They can get all results qualifying the similarity threshold. Besides, some approximation SSS and SSJ algorithms have also been introduced. These algorithms can guarantee to obtain most results within a certain probability. Most of these algorithms are based on locality sensitive hashing (LSH) [12,18], whose basic principle is: two similar sets in the original data space are still similar in a new mapped data space with a high probability. Therefore, LSH based algorithms usually focus on finding a series of hash functions by which two similar sets can be hashed into the same bucket with a high probability. The typical examples of such algorithms are MinHash [5] and BayesLSH [39], and so on. However, most of these algorithms concentrate on the early phase of similarity researches. It was observed that the performance of AllPair is better than LSH based algorithms, so we don't discuss LSH based algorithms in detail.

Besides algorithms discussed above, there are many other similarity search or join algorithms, such as TrieJoin [46], [11], EDJoin [9], and partition based string similarity search and join. However, these algorithms are not set similarity search and join algorithms, but string similarity search and join [21,27,28], which mainly exploit the relationships between edit distance threshold and the *q*-gram length. So, they are not suitable for SSS and SSJ tasks.

5. MapReduce based Algorithms

The volume of data increases from GB to TB, even to PB level. For example, Google's 5-gram dataset [43] has nearly 1 trillion records. Most of the traditional algorithms are memory based and are difficult to support datasets at such a level. As a scalable shared-nothing parallel data processing platform, MapReduce is a good remedy to this problem and can execute data intensive tasks in parallel on thousands of nodes by mapping data to <key, value> pairs.

Vernica *et al.* [43] extended prefix filtering on MapReduce. The main idea is that r and s should be sent to the same reducer for similarity computing if they have at least one common prefix element. The main disadvantage of the algorithm lies in that if there are more than one common prefix elements between r and s, the pair $< r, s >$ will be sent to multiple reducers and its similarity is redundantly computed, thus dragging down the efficiency of the algorithm.

To solve this problem, Kolb *et al*. [24] suggested an approach to compute the similarity of r and s only by the reducer that handles their minimum common element. This can ensure that two sets are compared only once even if they share more than one common element.

Kim [23] also computed the similarity of $< r, s >$ pair only once by combing redundant removal and *l*-prefix, and then using the reducer to handle their *l*-th common elements. As a result, redundant computations can be avoided, and at the same time candidates are sharply reduced.

Other than prefix based MapReduce algorithms, Metwally *et al.* [31] introduced V-Smart-Join to solve the problem of SSJ on multisets. It constructs an inverted index on the whole dataset and computes the similarity of candidate pairs of any two sets in each inverted list. The algorithm is efficient only for sparse datasets with a large number of independent elements.

Also aimed at reducing redundant computation, FSJoin [33] adopts a vertical partition policy that divides each set into equal amounts of segments by using carefully selected pivots. A main difference between this policy and the aforementioned ENPart schema lies in that the former divides the universe Ω unevenly to guarantee load balancing. By putting all segments with the same ID to the same mapper, duplicates can be avoided. Besides, FSJoin adopts a horizontal partition and three segment-aware filters to further promote the efficiency.

Most MapReduce based algorithms use prefix filtering to generate the key-value pairs and then dispatch these pairs to the corresponding reducers. By dispatching one key-value pairs to only one reducer, these algorithms can run much faster. However, the filtering techniques, such as length filtering, have not been fully exploited in these algorithms as shown in [38], so the efficiencies of these algorithms can be further improved.

6. Challenges

With the rapid influx of big datasets, along with the increasing complexity of data and query, SSS and SSJ algorithms are faced with some new challenges.

6.1. Challenges of Data Size

As the sizes of datasets expand rapidly, SSS and SSJ face much greater challenges. Although a variety of MapReduce based algorithms have been proposed to tackle these challenges, most of them are prefix filtering based, leading to redundant computations or additional overhead to eliminate redundancy. Efficiently finding similar sets from massive data will be the focus of future research.

6.2. Challenges of Complexity

The challenge of complexity is reflected in two aspects:

6.2.1. Challenges of data complexity.

In addition to the rapid increase of dataset size, the complexities of datasets are also increasing.

- High dimensional set representation.

Currently, the dimension of sets is usually high. For example, the dimensions used in [52] are as high as 10^5 to 10^8. In this case, the curse of dimensionality problem is often unavoidable.

- Nested set representation.

Nested sets are popular in XML, JSON and other circumstances. One example for this is the set containment search based on nested sets [17]. Executing SSS or SSJ on nested set is rather expensive and more pioneer works are urgently called in this field.

6.2.2. Challenges of query complexity

The increase of data complexity will inevitably lead to the increase of query complexity. In fact, the query complexity itself also sharply increases.

- Supporting different queries.

The applications for set processing are becoming more and more complex. For example, in e-commerce sites, we need to query which users buy beer and diapers together (a subset search), and which users have similar shopping habits (a similarity search). Obviously, if the designed index cannot simultaneously support subset search and similarity search, we should independently deploy two indexes. So, it is a realistic demand to design a single index that has a comprehensive processing capability and can support various queries.

- Supporting SSS and SSJ on multi-attribute data.

[29] has shown the requirement of performing different queries on multiple set-valued attributes. Two sets are similar if and only if they meet the similarity threshold on all relevant attributes. Solving such queries is a hard challenge as we should filter candidates cooperatively on multiple attributes.

6.3. Challenges of Low Threshold

Most existing algorithms assume a higher threshold, but a low threshold is also necessary in many cases. As mentioned in [49], it is not strange to perform clustering on 12 million documents with a threshold 0.2 and reconstructing 3D scenes from a large collection of photos with thresholds between 0.025 to 0.1. Since candidates increase rapidly with a decrease of the threshold, executing queries with low threshold is always a big challenge.

7. Conclusions and future works

This paper presents a survey on SSS and SSJ in which we introduce the basic concept, framework, similarity functions and major SSS and SSJ algorithms. Although extensive works have promoted the efficiency of SSS and SSJ a lot, there are still some avenues for future research.

7.1. Multi Technology Fusion

Most SSS and SSJ algorithms are based on a certain technique. Combining different techniques together, e.g. prefix filtering and partition, may be more promising.

7.2. The Statistical Characteristics of Datasets.

There are a large number of datasets varying in data size, element type, distribution, etc., and it is well known that no algorithm is optimal for all datasets and all thresholds. Therefore, it is possible to select the appropriate algorithms for the specific application by analyzing the statistical characteristics of datasets.

7.3. Integration into Database

SSS and SSJ are widely used, but support in current commercial DBMS are still insufficient. Most of these algorithms are processed in applications rather than in DBMS. Integrating SSS and SSJ algorithms into database management systems can fundamentally accelerate most applications. Although there are already some works [40] that exist in these fields, integrating recent work into databases is always needed.

Acknowledgements

The research is supported by Grants from the National Natural Science Foundation of China (No. 61562054, 51467007, 61462050), and the Personnel Training Project of Yunnan Province(No.KKSY201603016).

References

1. A. Arasu, V. Ganti, and R. Kaushik, "Efficient Exact Set-Similarity Joins," presented at the Proceedings of the 32nd international conference on Very large data bases, Seoul, Korea, 2006.
2. H. Bast and I. Weber, "Type Less, Find More: Fast Autocompletion Search with a Succinct Index," in *International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2006, pp. 364-371.
3. R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up All Pairs Similarity Search," presented at the Proceedings of the 16th international conference on World Wide Web, Banff, Alberta, Canada, 2007.
4. P. Bouros, S. Ge, and N. Mamoulis, "Spatio-Textual Similarity Joins," *Proc. VLDB Endow.*, vol. 6,no. 1, pp. 1-12, 2012.
5. A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-Wise Independent Permutations," *J. Comput. Syst. Sci.*, vol. 60,no. 3, pp. 630-659, 2000.
6. S. Burrows, S. M. M. Tahaghoghi, and J. Zobel, "Efficient Plagiarism Detection for Large Code Repositories," *Softw. Pract. Exper.*, vol. 37,no. 2, pp. 151-175, 2007.
7. S. Chaudhuri, V. Ganti, and R. Kaushik, "A Primitive Operator for Similarity Joins in Data Cleaning," presented at the Proceedings of the 22nd International Conference on Data Engineering, 2006.
8. A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google News Personalization: Scalable Online Collaborative Filtering," presented at the Proceedings of the 16th international conference on World Wide Web, Banff, Alberta, Canada, 2007.
9. D. Deng, G. Li, and J. Feng, "A Pivotal Prefix Based Filtering Algorithm for String Similarity Search," presented at the Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, Snowbird, Utah, USA, 2014.
10. D. Deng, G. Li, H. Wen, and J. Feng, "An Efficient Partition Based Method for Exact Set Similarity Joins," *Proc. VLDB Endow.*, vol. 9,no. 4, pp. 360-371, 2015.
11. J. Feng, J. Wang, and G. Li, "Trie-Join: A Trie-Based Method for Efficient String Similarity Joins," *The VLDB Journal*, vol. 21,no. 4, pp. 437-461, 2012.
12. A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions Via Hashing," presented at the Proceedings of the 25th International Conference on Very Large Data Bases, 1999.

13. L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava, "Approximate String Joins in a Database (Almost) for Free," presented at the Proceedings of the 27th International Conference on Very Large Data Bases, 2001.
14. S. Helmer, R. Aly, T. Neumann, and G. Moerkotte, "Indexing Set-Valued Attributes with a Multi-Level Extendible Hashing Scheme," presented at the Proceedings of the 18th international conference on Database and Expert Systems Applications, Regensburg, Germany, 2007.
15. S. Helmer and G. Moerkotte, "A Performance Study of Four Index Structures for Set-Valued Attributes of Low Cardinality," *The VLDB Journal The International Journal on Very Large Data Bases*, vol. 12,no. 3, pp. 244-261, 2003.
16. M. Henzinger, "Finding near-Duplicate Web Pages: A Large-Scale Evaluation of Algorithms," in *International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2006, pp. 284-291.
17. A. Ibrahim and G. H. L. Fletcher, "Efficient Processing of Containment Queries on Nested Sets," presented at the Proceedings of the 16th International Conference on Extending Database Technology, Genoa, Italy, 2013.
18. P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," presented at the Proceedings of the thirtieth annual ACM symposium on Theory of computing, Dallas, Texas, USA, 1998.
19. Y. Ishikawa, H. Kitagawa, and N. Ohbo, "Evaluation of Signature Files as Set Access Facilities in Oodbs," *SIGMOD Rec.*, vol. 22,no. 2, pp. 247-256, 1993.
20. L. Jia, J. Xi, M. Li, Y. Liu, and D. Miao, "Eti: An Efficient Index for Set Similarity Queries," *Frontiers of Computer Science*, vol. 6,no. 6, pp. 700-712, 2012.
21. Y. Jiang, D. Deng, J. Wang, G. Li, and J. Feng, "Efficient Parallel Partition-Based Algorithms for Similarity Search and Join with Edit Distance Constraints," presented at the Proceedings of the Joint EDBT/ICDT 2013 Workshops, Genoa, Italy, 2013.
22. O. Kaser and D. Lemire, "Compressed Bitmap Indexes: Beyond Unions and Intersections," *Softw. Pract. Exper.*, vol. 46,no. 2, pp. 167-198, 2016.
23. C. Kim and K. Shim, "Supporting Set-Valued Joins in Nosql Using Mapreduce," *Information Systems*, vol. 49,no. pp. 52-64, 2015.
24. L. Kolb, A. Thor, and E. Rahm, "Don't Match Twice: Redundancy-Free Similarity Computation with Mapreduce," presented at the Proceedings of the Second Workshop on Data Analytics in the Cloud, New York, New York, 2013.
25. J. P. Kumar and P. Govindarajulu, "Duplicate and near Duplicate Documents Detection: A Review," *European Journal of Scientific Research*, vol. 32,no. pp. 1450-216, 2009.
26. C. Li, J. Lu, and Y. Lu, "Efficient Merging and Filtering Algorithms for Approximate String Searches," presented at the Proceedings of the 2008 IEEE 24th International Conference on Data Engineering, 2008.
27. G. Li, D. Deng, and J. Feng, "A Partition-Based Method for String Similarity Joins with Edit-Distance Constraints," *ACM Transactions on Database Systems*, vol. 38,no. 2, pp. 1-33, 2013.
28. G. Li, D. Deng, J. Wang, and J. Feng, "Pass-Join: A Partition-Based Method for Similarity Joins," *Proc. VLDB Endow.*, vol. 5,no. 3, pp. 253-264, 2011.
29. G. Li, J. He, D. Deng, and J. Li, "Efficient Similarity Join and Search on Multi-Attribute Data," presented at the Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, 2015.
30. W. Mann, N. Augsten, and P. Bouros, "An Empirical Evaluation of Set Similarity Join Techniques," *Proc. VLDB Endow.*, vol. 9,no. 9, pp. 636-647, 2016.
31. A. Metwally and C. Faloutsos, "V-Smart-Join: A Scalable Mapreduce Framework for All-Pair Similarity Joins of Multisets and Vectors," *Proc. VLDB Endow.*, vol. 5,no. 8, pp. 704-715, 2012.
32. L. A. Ribeiro, T. H, #228, and rder, "Generalizing Prefix Filtering to Improve Set Similarity Joins," *Inf. Syst.*, vol. 36,no. 1, pp. 62-78, 2011.
33. C. Rong, C. Lin, Y. N. Silva, J. Wang, W. Lu, and X. Du, "Fast and Scalable Distributed Set Similarity Joins for Big Data Analytics," in *IEEE International Conference on Data Engineering*, 2017, pp. 1059-1070.
34. C. Rong, W. Lu, X. Wang, X. Du, Y. Chen, and A. K. H. Tung, "Efficient and Scalable Processing of String Similarity Join," *IEEE Trans. on Knowl. and Data Eng.*, vol. 25,no. 10, pp. 2217-2230, 2013.
35. C. Rong, T. Xu, and X. Du, "Partition-Based Set Similarity Join," *Jorunal of Computer Research and Development*, vol. 49,no. 10, pp. 2066-2076, 2012.
36. M. Sahami and T. D. Heilman, "A Web-Based Kernel Function for Measuring the Similarity of Short Text Snippets," presented at the Proceedings of the 15th international conference on World Wide Web, Edinburgh, Scotland, 2006.
37. S. Sarawagi and A. Kirpal, "Efficient Set Joins on Similarity Predicates," presented at the Proceedings of the 2004 ACM SIGMOD international conference on Management of data, Paris, France, 2004.
38. A. D. Sarma, Y. He, and S. Chaudhuri, "Clusterjoin: A Similarity Joins Framework Using Map-Reduce," *Proc. VLDB Endow.*, vol. 7,no. 12, pp. 1059-1070, 2014.
39. V. Satuluri and S. Parthasarathy, "Bayesian Locality Sensitive Hashing for Fast Similarity Search," *Proc. VLDB Endow.*, vol. 5,no. 5, pp. 430-441, 2012.
40. Y. N. Silva, W. G. Aref, and M. H. Ali, "The Similarity Join Database Operator," in *IEEE International Conference on Data Engineering*, 2010, pp. 892-903.
41. E. Spertus, M. Sahami, and O. Buyukkokten, "Evaluating Similarity Measures: A Large-Scale Study in the Orkut Social Network," presented at the Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, Chicago, Illinois, USA, 2005.
42. M. Theobald, J. Siddharth, and A. Paepcke, "Spotsigs: Robust and Efficient near Duplicate Detection in Large Web Collections," presented at the Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval, Singapore, Singapore, 2008.
43. R. Vernica, M. J. Carey, and C. Li, "Efficient Parallel Set-Similarity Joins Using Mapreduce," presented at the Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, Indianapolis, Indiana, USA, 2010.
44. S. Wandelt, D. Deng, S. Gerdjikov, S. Mishra, P. Mitankin, M. Patil, *et al.*, "State-of-the-Art in String Similarity Search and Join," *SIGMOD Rec.*, vol. 43,no. 1, pp. 64-76, 2014.
45. S. Wandelt, J. Starlinger, M. Bux, and U. Leser, "Rcsi: Scalable Similarity Search in Thousand(S) of Genomes," *Proc. VLDB Endow.*, vol. 6,no. 13, pp. 1534-1545, 2013.
46. J. Wang, J. Feng, and G. Li, "Trie-Join: Efficient Trie-Based String Similarity Joins with Edit-Distance Constraints," *Proc. VLDB Endow.*, vol. 3,no. 1-2, pp. 1219-1230, 2010.
47. J. Wang, T. Kraska, M. J. Franklin, and J. Feng, "Crowder: Crowdsourcing Entity Resolution," *Proc. VLDB Endow.*, vol. 5,no. 11, pp. 1483-1494, 2012.
48. J. Wang, G. Li, and J. Feng, "Can We Beat the Prefix Filtering?: An Adaptive Framework for Similarity Join and Search," presented at the Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, Scottsdale, Arizona, USA, 2012.
49. C. Xiao, J. Qin, W. Wang, Y. Ishikawa, K. Tsuda, and K. Sadakane, "Efficient Error-Tolerant Query Autocompletion," *Proc. VLDB Endow.*, vol. 6,no. 6, pp. 373-384, 2013.

50. C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang, "Efficient Similarity Joins for near-Duplicate Detection," *ACM Trans. Database Syst.*, vol. 36,no. 3, pp. 1-41, 2011.
51. M. Yu, G. Li, D. Deng, and J. Feng, "String Similarity Search and Join: A Survey," *Front. Comput. Sci*., vol. 10,no. 3, pp. 399-417, 2016.
52. J. Zhai, Y. Lou, and J. Gehrke, "Atlas: A Probabilistic Algorithm for High Dimensional Similarity Search," presented at the Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, Athens, Greece, 2011.
53. J. Zhou, X. Nie, L. Qin, and J. Zhu, "Web Clustering Based on Tag Set Similarity," *Journal of Computers*, 2011.

Lianyin Jia is an Associate Professor in the Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming, China. He Received his Ph.D. degree in Computer Science from South China University of Technology, Guangzhou, China in 2013. His current research interests include database, data mining, information retrieval and parallel computing.

Lulu Zhang is an MPhil student in the Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming, China. Her current research interests include database, information retrieval, parallel computing.

Guoxian Yu is an Associate Professor in the College of Computer and Information Science, Southwest University, Chongqing, China. He received his Ph.D. degree in Computer Science from South China University of Technology, Guangzhou, China in 2013. His current research interests include data mining and bioinformatics. Dr. Yu has served as PC member for KDD, ICDM and SDM, ISBRA and other conferences. He is a recipient of Best Poster Award of SDM2012 Doctoral Forum, Best Student Paper Award of IEEE ICML2011 (International Conference on Machine Learning and Cybernetics) and Excellent Student Paper Award of China Conference on Machine Learning (with his student), 2015.

Jinguo You is an Associate Professor in the Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming, China. He Received his Ph.D. degree in Computer Science from South China University of Technology, Guangzhou, China in 2009. His current research interests include data warehouse and data mining.

Jiaman Ding is an Associate Professor in the Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming, China. He now is also a Ph.D. candidate in Kunming University of Science and Technology. His current research interests include data mining, cloud computing.

Mengjuan Li is a Librarian in the Department of Technology, Library, Yunnan Normal University, Kunming, China. She Received her master degree in Computer Science from Kunming University of Science and Technology, Kunming, China in 2008. Her main research interests include information retrieval, parallel computing.