Approximate String Similarity Join using Hashing Techniques under Edit Distance Constraints

Peisen Yuan[a], Haoyun Wang[a], Jianghua Che[a], Shougang Ren[a], Huanliang Xu[a], Dechang Pi[b]
[a] College of Information Science and Technology, Nanjing Agricultural University, Nanjing 210095, China
peiseny@gmail.com
[b] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China

***Abstract*— The string similarity join, which is employed to find similar string pairs from string sets, has received extensive attention in database and information retrieval fields. To this problem, the *filter-and-refine* framework is usually adopted by the existing research work firstly, and then various filtering methods have been proposed. Recently, tree based index techniques with the edit distance constraint are effectively employed for evaluating the string similarity join. However, they do not scale well with large distance threshold. In this paper, we propose an efficient framework for approximate string similarity join based on Min-Hashing locality sensitive hashing and *trie*-based index techniques under string edit distance constraints. We show that our framework is flexible between trading the efficiency and performance. An empirical study using the real datasets demonstrates that our framework is more efficient and scales better.**

***Index Terms*— Approximate String Similarity Join, Locality Sensitive Hashing, Min-Hashing, String Edit Distance, Trie Join**

I. INTRODUCTION

This article is a revised and expanded version of a paper entitled Hash^{ed}-Join:Approximate String Similarity Join with Hashing presented at the DASFAA 2014 [1]. We have carefully revised the original conference version according to the comments made by reviewers, and give observations based on the string datasets using in our experiment, theory analysis and conduct more experiment etc.

String is one of the most important data types in modern data processing systems [2]. It is ubiquitous in the computer applications, such as in the Web pages, DNA sequence data, XML documents, relational database tables etc, which has attracted lots of attention of computer science researchers.

One of the important researches on string is the similarity join, i.e., finding all the similarity string pairs from the two string sets, which is a key operation in many real-world applications, such as data integration for resolving the entity [3], data cleaning [4], duplicate detection [5] and so on. It has been applied in the industry field as well, for example, Google employs similarity join for duplicate web page detection [6] and Microsoft proposes primitive similarity join operator SSJOIN [4]. The latter has been used in Data Debugger project [7]. The similarity join has been received extensive attention from the database and information retrieval fields and there are extensive literatures for addressing this problem [5], [8].

For measuring the string similarity, the string edit distance is used by most previous studies. However, the time and space complexity of evaluating string edit distance between two strings s_1 and s_2 is $O(|s_1||s_2|)$ [9], where $|s|$ denoted the length of the string. Even the fastest algorithm has been reported still requires quadratic complexity $O(|s|^2/log(|s|))$ [10]. Therefore, most existing work mainly focus on the *filter-and-refine* framework. According to this framework, strings are firstly filtered by some filtering techniques in a heuristic way, and the edit distance evaluation is applied on remaining candidate string set subsequently.

In term of string similarity join with edit distance constraint, recent researches [8], [11]–[13] explore the tree index techniques. J. Wang et al. propose the Trie-Join [8] for string similarity join using a trie tree, which processes string similarity join efficiently and the space cost of the trie indexing structure is much smaller than existing work. Some pruning techniques are also proposed to improve the performance of the string join with edit distance constraint on short string datasets. However, the performance of the Trie-Join degrades for long strings and when the string edit distance threshold increases.

Based on the analysis of the Trie-Join technique, in this paper, we take the locality sensitive hashing (LSH for short) [14] and trie join techniques into account and propose a framework **Hash^{ed}-Join** for approximate string similarity join under edit distance constraint. Our framework employs the hashing techniques and trie structure index, which improves the performance of Trie-Join greatly with the increasing of the edit distance threshold. The processing procedure of Hash^{ed}-Join is shown in

Manuscript received May XX, 2014; revised XX XX, 2014; accepted XX XX, 2014. © 2014 IEEE.
This work was supported by the Fundamental Research Funds for the Central University (No. KYZ201421), the Natural Science Foundation of Jiangsu Province (No. Bk2012363) and the Three Agricultural Projects of Jiangsu province of China (SXGC(2013)372).Corresponding should be addressed to Peisen Yuan (peiseny@gmail.com).

Fig.1, which can be divided into four steps.

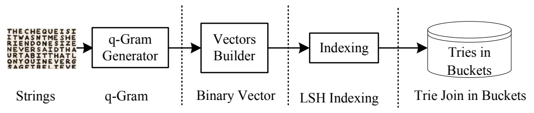


Figure 1: Processing Procedure of Hash^{ed}-Join

In Fig.1, the q-grams of each string in the string set are extracted in the first place. In the second step, the binary vectors of the strings are built based on the their corresponding q-grams set. In the third step, the Min-Hashing based LSH technique is employed, which hashes the similar strings into the same bucket, i.e., the string sharing q-grams. We prove that the strings satisfying the edit distance threshold are guaranteed to fall into the same bucket with high probability theoretically. In the forth step, a trie structure index is built in each collision bucket and the trie join technique is applied. The similar string pairs in each bucket are merged together and the final result for the string sets is returned finally.

Since Hash^{ed}-Join is loosely coupled with the string join processing techniques, it can be used with other index techniques developed for string similarity join in the hash bucket of LSH, such as B^{ed}-tree index [11] et al.

To summarize, the main contributions of this paper are briefly outlined as follows:

- The framework Hash^{ed}-Join using LSH and trie join techniques for the approximate string similarity join problem is proposed.
- The characteristics of string similarity join on real datasets is analyzed.
- Complexities of Hash^{ed}-Join are analyzed theoretically.
- The join quality of Hash^{ed}-Join are analyzed with the parameters of Min-Hashing LSH and string edit distance threshold.
- Extensive experiments on real datasets are conducted to demonstrate the effectiveness and efficiency of Hash^{ed}-Join.

The remainder of the paper is organized as follows. The preliminaries are presented in Section II. The Trie-Join technique for string similarity join and several observations of string similarity join on the real datasets are introduced in Section III . Based on the observations, the string similarity join method Hash^{ed}-Join is proposed in Section IV. In Section V, experimental evaluation on the real datasets is presented. We summarize the related work in Section VI and conclude the paper in Section VII.

II. Preliminaries

A. Problem Statement

Given two strings s_1 and s_2, a proposition formula $\mathcal{F}$ is defined, which is in the form of $sim(s_1, s_2) \geq \theta$, where $sim(s_1, s_2)$ is the similarity metric for strings s_1 and s_2, or $dist(s_1, s_2) \leq \tau$, where $dist(s_1, s_2)$ is the distance metric for strings s_1 and s_2. The task of string similarity join is retrieving the similar string pairs between two string sets that satisfying the proposition formula $\mathcal{F}$. The formal definition of the problem is presented as follows.

Definition 1: String Similarity Join

Given two string sets S_1, S_2 and proposition formula $\mathcal{F}$, the similarity join between S_1 and S_2 is denoted as $S_1 \bowtie_{\mathcal{F}} S_2$. The result of the join is denoted as $S_1 \bowtie_{\mathcal{F}} S_2$ = $\{< s_1, s_2 > |s_1 \in S_1 \ and \ s_2 \in S_2, \mathcal{F}(< s_1, s_2 >) \ is \ true\}$, where $< s_1, s_2 >$ is the similar pair that satisfying the proposition formula $\mathcal{F}$.

Example 1: Given two string sets S_1 = {"*microsoft*", "*applies*", "*informix*", "*tree*"} and S_2 = {"*apple*", "*infromix*", "*google*", "*trie*", "*mcrosoft*"}, and the proposition formula $\mathcal{F}$ is defined as the string edit distance defined in Section II-C, $dist_{ed}(s_1, s_2) \leq 1$. The similarity join result on the two string sets is $S_1 \bowtie_{\mathcal{F}} S_2$ = {<"*microsoft*", "*mcrosoft*">, <"*trie*", "*tree*">}.

B. q-gram

Given a string s and an integer q, its q-grams can be obtained by a sliding window on s with length q, contiguously splitting the string into a group of substrings. For a given string s, its q-grams may occur multiple times, we treat the duplicate q-grams as new ones by inserting an integer representing the occurrence order [5]. In this way, a string s can generate $l = |s| - q + 1$ q-grams.

Example 2: Consider the string s = "*mathematics*", let q = 2, the set of 2-grams of s, $\Phi_2(s)$ = { "*ma*", "*at*", "*th*", "*he*", "*em*", "*ma2*", "*at2*", "*ti*", "*ic*", "*cs*"}. The q-gram such as "*ma*"reoccurs later is appended with the order of the occurrence number to differentiate it. The length of $\Phi_2(s)$, i.e., $|\Phi_2(s)|$ is 10.

Given a string set S, for each string $s \in S$, we extract its q-grams and denoted as $\Phi_q(s)$. Let $\mathcal{U}$ be the universal of the q-grams of S, i.e., $\mathcal{U} = \bigcup_{i=1}^{|S|} \Phi_q(s_i) = \{g_1, \cdots, g_{|\mathcal{U}|}\}$, where $i = 1, \cdots, |S|$. Then for a string s, it can be represented by a binary vector $\vec{v}_b^s$ with the vector length $|\mathcal{U}|$, where $\vec{v}_b^s[j] = 1$, if $g_j \in \Phi_q(s)$; otherwise $\vec{v}_b^s[j] = 0$, for $j = 1, \cdots, |\mathcal{U}|$. In this paper, the string is taken as the binary vector of its q-gram and they are used interchangeably if not confused.

C. Similarity Metrics

There are many similarity metrics that can be used for measuring string similarity, such as string edit distance, Jaccard similarity, Hamming distance and cosine similarity etc. In this paper, string edit distance and Jaccard similarity [15] are used and they are introduced in the following section.

1) Jaccard Similarity: The Jaccard similarity is also known as Jaccard coefficient [15]. Given two sets A and B, their Jaccard similarity is defined as

$$sim_{jacc}(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

In many applications, the Jaccard distance is usually used as the distance measure for set similarity, which is defined as Eq.2.

$$dist_{jacc}(A, B) = 1 - sim_{jacc}(A, B). \tag{2}$$

2) String Edit Distance: The string edit distance is also called as the *Levenshtein distance* [16], which is a metric used for measuring the differences between two strings. The evaluation of string edit distance is based on three primitive operations: insertion, deletion and substitution, which are denoted as op_i, op_d and op_s. The definition is defined as follows.

Definition 2: String Edit Distance

Given two strings s_1 and s_2, the edit distance between s_1 and s_2 is defined as the *minimum* number of the three primitive operations needed to transform s_1 to s_2, which is denoted as $dist_{ed}(s_1, s_2) = \min \sum (c_i * op_i + c_d * op_d + c_s * op_s)$, where c_i, c_d and c_s are the cost of the operation respectively.

In this paper, the cost of each operation is set to 1. For instance, given two strings s_1 = "*goodl*"and s_2 = "*google*". s_1 can be transformed to s_2 by substituting d for g and inserting e. Thus the edit distance between s_1 and s_2 $dist_{ed}(s_1, s_2) = 2$. The string edit distance satisfies the symmetry property, i.e., $dist_{ed}(s_1, s_2) = dist_{ed}(s_2, s_1)$.

D. Min-Hashing

The Min-Hashing is used for approximately set similarity evaluation [17]. It has the property that the probability of two sets have the same value of Min-Hashing is equal to their Jaccard similarity, and the formal definition used in this paper is given as follows.

Given a random hash function $h : \mathcal{S} \to I$, where $\mathcal{S}$ is the domain of the string set, and I is an integer set. For a string $s \in \mathcal{S}$, which is represented with the binary vector $\vec{v}$, the Min-Hashing function is defined as $m_h(\vec{v}) = arg\ min\{h(\vec{v}_i) \mid \vec{v} \in \vec{V_b}$, $\vec{V_b}$ is binary vectors for all the strings, $\vec{v}_i$ is the i-th index of $\vec{v}$ if $\vec{v}[i] = 1$, for $0 \le i \le |\vec{v}| - 1\}$.

According to the property of Min-Hashing, for two strings s_1 and s_2 and their q-gram sets $\Phi_q(s_1)$ and $\Phi_q(s_2)$, the binary vectors of them are represented by $\vec{v}_b^{s_1}$ and $\vec{v}_b^{s_2}$ respectively. Their Jaccard similarity can be approximately computed by Eq.3.

$$sim_{jacc}(\Phi_q(s_1), \Phi_q(s_2)) = \mathbf{Pr}[m_h(\vec{v}_b^{s_1}) = m_h(\vec{v}_b^{s_2})]. \tag{3}$$

To reduce the probability of false positive retrieval, a random hash family $\mathcal{H} : \mathcal{D} \to I$ is usually used. Given a hash family $\mathcal{H}$, n Min-Hashing signatures are computed for each string. Thus the binary vector $\vec{v}_b^s$ of the string s is transformed into $g(\vec{v}_b^s)$ and represented as Eq.4.

$$g(\vec{v}_b^s) =< m_{h_1}(\vec{v}_b^s), m_{h_2}(\vec{v}_b^s), \cdots, m_{h_n}(\vec{v}_b^s) >, m_{h_i} \in \mathcal{H}, \tag{4}$$

for $i = 1, \cdots, n$.

E. Locality Sensitive Hashing

The concept of locality sensitive hashing (LSH) is introduced in [14] and widely used for approximate nearest neighbor search of high dimensional data etc. The basic definition of LSH can be formalized as follow.

Definition 3: Let $\mathcal{O}$ be the domain of the objects, $o_1, o_2 \in \mathcal{O}$, and $d_1 < d_2$ be two distances according to the distance metric $dist(o_1, o_2)$. A function family $\mathcal{H}$ is said to be (d_1, d_2, p_1, p_2)-*sensitive*, if each $h \in \mathcal{H}$ satisfies the following two conditions:

- If $dist(o_1, o_2) \le d_1$, then $\mathbf{Pr}_{\mathcal{H}}[h(o_1) = h(o_2)] \ge p_1$;
- If $dist(o_1, o_2) \ge d_2$, then $\mathbf{Pr}_{\mathcal{H}}[h(o_1) = h(o_2)] \le p_2$, where $p_1 > p_2 \in [0, 1]$.

The LSH index is a data structure using a family of LSH functions $\mathcal{H}$, which is constructed in the following two steps [18]:

1 Given an integer r, define a function family $\mathcal{G} = \{g : \mathcal{O} \to I^r\}$, and for $g \in \mathcal{G}$, $g(o) = < h_1(o), ..., h_r(o) >$, where $h_i \in \mathcal{H}$ for $1 \le i \le r$.

2 For an integer b, randomly choose $g_1, ..., g_b$ from $\mathcal{G}$. Construct a hash table for each g_i, for $1 \le i \le b$.

In order to construct a Min-Hashing based LSH index, the *signature* matrix is divided into b bands with r Min-Hashing signatures in each band, i.e., $n = b * r$. The signatures of a string in each band are concatenated and hashed into M collision buckets, where M is an integer. If Jaccrad distance is used as the distance metric defined in Eq.2, then Min-Hashing LSH is $(d_1, d_2, 1-(1-p_1^r)^b, 1-(1-p_2^r)^b)$-*sensitive* [19].

The parameter r control the filtering effectiveness and b controls the approximation factor. i.e., the bigger the parameter r is, the more non-similar strings are filtered; the bigger of b, the better approximation to the real result. Suppose the Jaccard similarity of two objects is θ, the probability that they can be retrieved with Min-hashing LSH is equal to Eq.5.

$$p = 1 - (1 - \theta^r)^b. \tag{5}$$

In order to achieve a false negative rate δ, it demands b bands, where b satisfies Eq.6 [20].

$$b = \frac{1}{\theta^r} \log \frac{1}{\delta}. \tag{6}$$

III. Trie Join and Observations

A. Trie Join

A trie is a tree structure and used for indexing the strings. The path of the trie tree from the root to the leaf represents a sequence of string, and the nodes in the trie tree indicate the substring which is associated with. All the descendants of a trie node have a common prefix of the string associated with the node, and the trie root is associated with an empty string. String values are normally associated with leaf nodes.

Fig.2 demonstrates a trie tree of the running example strings in Table I. The numbers in bold near the nodes in Fig.2 indicate the corresponding node ID, and the node ID of the root is set to 0. For example, node 11 in Fig.2 denotes substring "*goo*", which is the prefix of "*google*"and "*good*".

Given a string s, node n in the trie tree is called an *active node* of s if $dist_{ed}(s, n) \le \tau$ [8]. The active node

TABLE I.: Strings of the Running Example

String ID	Sample Strings
s_1	*apple*
s_2	*applies*
s_3	*good*
s_4	*google*
s_5	*tree*
s_6	*trie*

set of s is denoted as $A(s)$ and $n \in A(s)$ if n is an active node of s. In Fig.2, the node set enclosed by the $\{\}$ is the active node set of the corresponding node nearby with the string edit distance threshold τ = 1. For example, the trie node 10 represents the substring "*go*"and the active node set of which is $\{9, 10, 11\}$. The active nodes 9, 10, 11 represent the substrings "*g*","*go*"and "*goo*"respectively. Their string edit distance with "*go*"is not bigger than 1.

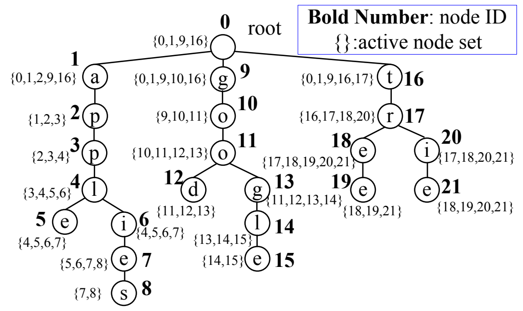


Figure 2: An Example of Trie Tree with $\tau = 1$ on the Running String Set

To evaluate the active node set for each node in the trie, the active node of the root node is computed firstly. Then for each internal node in the trie, its active node set can be evaluated using its parent's one. This is guaranteed by the Lemma 1 proposed in [8].

Lemma 1: Given a trie node n, let p be the parent of node n, for each node $a' \in A(n)$, there exists a node $p' \in A(p)$, such that p' is an ancestor of a'.

For example, in Fig.2, for evaluating the active node set of node 11, i.e., $A(11)$, the descendants of nodes in the active node set of its parent node $A(10) = \{9, 10, 11\}$ are needed to be verified whether they are the active nodes of node 11.

After obtaining the active node set for each node in the trie, the similar pairs can be evaluated with the active node sets. For each leaf node n_l in the trie, the active node $a' \in A(n_l)$ is verified whether a' is a leaf. If a' is a leaf node, then $< n_l, a' >$ is similar pair. For example, in Fig.2, node 19 is a leaf node, in the active node set $A(19)$, there is a node 21, which is a leaf node. Therefore, <19, 21> is a similar pair for the edit distance threshold τ = 1.

B. Problems of Trie Join

According to the evaluation procedure of the Trie-Join [8], we observe that dividing the string set into groups can reduce the computation overhead, especially with the increasing of the edit distance threshold τ. This can be illustrated by Fig.3. According to Trie-Join, the trie nodes in the trie tree with the length τ in the first branch will be the active nodes of the other branches with the length no large than τ, and the nodes in other branches will be the active nodes of other branches as well. However, lots of the nodes in the active node set do not contribute to the finally result, thus it wastes much computation.

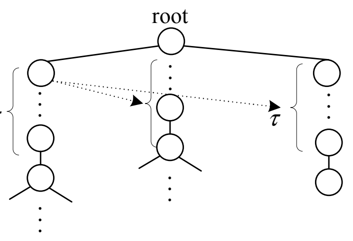


Figure 3: Illustration of Trie-Join with Edit Distance

Take Fig.2 as an example, the active nodes of the trie branches in Fig.2 are shown in Fig.4. The the total active node number of Fig.2 is 82, nevertheless, the total active node number of the three branches is $\sum(29, 24, 25)$ = 78, which is less than the former. When τ is 2 , the active nodes of the three branches is $\sum(45, 38, 37)$ = 120, which is much less than the total number of the trie tree 142 in Fig.2.

The above example indicates that dividing the string set into groups can reduce the size of the active node set. The key point of this method is that by dividing the string set larger than the edit distance threshold into different groups, the unnecessary computation can be avoided. We now introduce an our solution which hashes the strings within the edit distance threshold into the same groups with high probability.

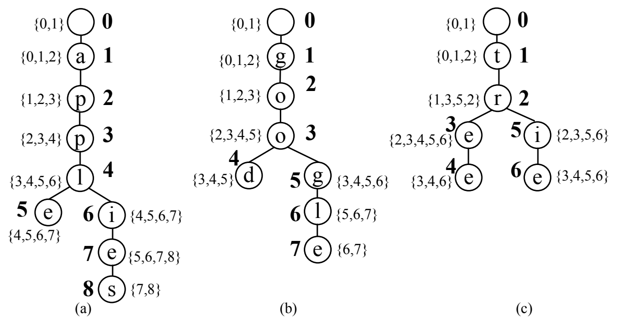


Figure 4: Active Node of the Trie Tree Branches with Edit Distance $\tau = 1$

C. Observations

Given a string s, the size of q-grams of s is $|\Phi_q(s)|$ = $|s| - q + 1$, which is far less than the cardinality of the q-gram universal $|\mathcal{U}|$, i.e. $|\Phi_q(s)| \ll |\mathcal{U}|$. Thus the feature space of a string is rather sparse against $\mathcal{U}$ and the dimension of the binary vector is rather high. Therefore, comparing the Jaccard similarity using the binary vector directly is not efficient.

Observation 1: Strings binary vector is sparse with high-dimensionality.

Another observation is that there are only a few similarity pair instances in the real string dataset. Table II reveals our observation on three real datasets. The 2nd to the 4th columns of the table show the ratio of the similarity pairs in the datasets under the edit distance threshold τ. The ratio is evaluated by Similarity Pair Ratio defined as

$$Ratio_{sp} = \frac{|similarity\ pairs|}{|S|(|S|-1)/2}, \qquad (7)$$

where S is the string dataset and $|S|(|S|-1)/2$ is the total number of ordered string pairs, i.e., $s_i, s_j \in S$, string pair $< s_i, s_j >$ and $i < j$.

From Table II, observation can be seen that the similar string pair ratio $Ratio_{sp}$ within the edit distance threshold from 1 to 3 on real dataset is extremely sparse. Even through the edit distance threshold enlarged, there are only few similar pairs. Thus we conclude our observation in a formal way.

Observation 2: In reality settings, the similar string pairs are extremely sparse under the string edit distance constraint.

From the above observations, we can see that the retrieving of the string similarity pairs in the real dataset is like finding the needle in a haystack. How to effectively find the needle in a haystack? The Min-Hashing can give an effective way for retrieving the objects with high similarity in high-dimensional space [21].

TABLE II.: Similarity Pairs Ratio with Edit Distance on Real Datasets

Dataset	$\tau = 1$	$\tau = 2$	$\tau = 3$
DBLP Author	5.82E-7	2.60E-6	1.24E-5
DBLP TA [1]	1.28E-9	2.25E-9	3.88E-9
DNA Seq [2]	5.89E-6	1.79E-5	2.58E-5

[1] Title and Author.
[2] DNA Sequences.

Base on the above observations, we propose the Hash^{ed}-Join framework. Firstly, the Min-Hashing based LSH technique is applied, which hashes the similar strings into the same bucket approximately. Then the Trie-Join technique is employed for evaluating the similar string pairs within each bucket.

Due to the well scalability of LSH, our framework can easily be extended to run in a parallel way on cluster with large string sets, such as MapReduce framework [22], [23].

IV. Hash^{ed}-Join

In this section, the approach of Hash^{ed}-Join is presented. First, the algorithm of q-gram binary vector building is proposed. Then Hash^{ed}-Join is introduced.

A. String Binary Vector Building

According to the processing procedure in Fig.1, the q-grams of each string are extracted firstly, then the binary vector for each string is constructed. The algorithm is summarized in Algorithm 1.

```
Algorithm 1: String Binary Vectors Building
  Input: String Set $S$; Integer $q$.
  Output: Compressed Binary Vectors.
1 Vector $\vec{V}$ = $\emptyset$;
2 foreach $s \in S$ do
3     $\Phi_q(s) = qgramGen(s)$;
4 $\mathcal{U} = \cup_{i=1}^{|S|} \Phi_q(s_i)$;
5 foreach $\Phi_q(s)$ do
6     Vector $\vec{v}$ = new Vector();
7     foreach $g \in \mathcal{U}$ do
8         if $g \in \Phi_q(s)$ then
9             $\vec{v}$.add(1);
10        else
11            $\vec{v}$.add(0);
12    $\vec{v}_c = VectorCompressor(\vec{v})$;
13    $\vec{V} = \cup \vec{v}_c$;
14 return $\vec{V}$;
```

The input of Algorithm 1 is the string set S and the parameters q. It outputs the compressed binary vectors as the result.

In Algorithm 1, the q-grams of each string in the string set are extracted firstly (lines 2-3). After generating the q-grams, the binary vector for string s is generated based on the q-gram universal $\mathcal{U}$ of the string set and the q-gram set $\Phi_q(s)$ of string s (lines 4-11).

Due to the high dimensionality of the binary vectors, for reducing the storage overhead, the each binary vector is divided into several groups with length L, where $L \leq 64$ and $L \leq |\mathcal{U}|$, for the reason that the binary can be compressed into integers. In each group, the binary vector is transformed into an integer (line 12). Finally, the compressed binary vectors are returned by the algorithm (line 14).

B. Hash^{ed}-Join Algorithm

After generating the binary vectors for the string sets, the main algorithm of Hash^{ed}-Join is proposed in the following section.

The Hash^{ed}-Join algorithm is outlined in Algorithm 2. The input of the algorithm includes compressed binary vectors, string set list SL and edit distance threshold τ. The output of the algorithm is the similar string pairs. In this paper, we only take self-join into account, i.e., $S_1 \bowtie_{\mathcal{F}} S$. The join between two different string sets, $S_1 \bowtie_{\mathcal{F}} S_2$, can be can be easily extended according to Trie-Join [8].

In this paper, Hash^{ed}-Join uses Min-Hashing based LSH index for dividing the string set into groups, which

Algorithm 2: Hash^{ed}-Join Algorithm

```
Input: Compressed String Binary Vectors $\vec{V}$; String Set List $SL$; Edit Distance Threshold $\tau$.
Output: Similar String Pairs within Threshold $\tau$.
1  MinHashClass minHash = new MinHashClass();
2  $\vec{V'} = decompress(\vec{V})$; /* Decompress the binary vectors*/;
3  $LSHIndex$ = $minHash(\vec{V'}, SL)$; /*Generate the Min-Hashing LSH index for the string set*/;
4  foreach band $b \in LSHIndex$ do
5      $bandNo = getBandNo(b)$;
6      foreach bucket $buc \in b$ do
7          $bucketNo = getBucketNo(bandNo)$;
8          $S' = getString(bandNo \oplus bucketNo, SL)$; /*Get the subset $S'$ of the string set $S$ from the collision bucket */;
9          if ($|S'|$ == 1) then
10             $sp$ = null; /*similarity pair */;
11         else
12             $trie = buildTrieTree(S')$; /*Build a trie tree index for $S'$*/;
13             $sp = trie.GenerateSimiPair(trie, \tau)$; /* call Trie-Join Algorithm [8]*/;
14         $SP = MergeSimilarPairs(sp)$; /*Similarity Pair Set */;
15 return $SP$;
```

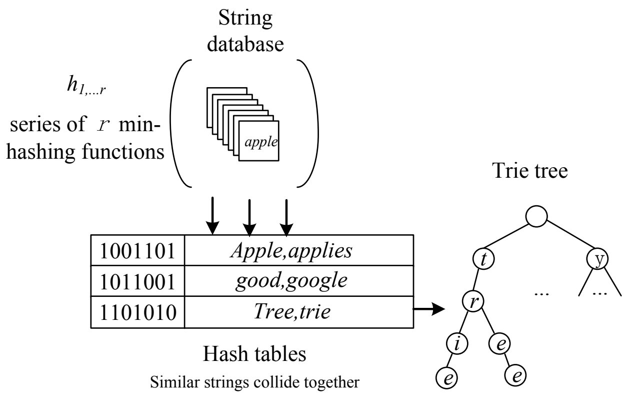


Figure 5: Hash^{ed}-Join based on LSH and Trie join

ensures the similar strings to fall into the same group with high probability. Fig.5 illustrates this procedure. In fig.5, the strings of the string database is hashed by r Min-Hashing functions, and the columns represent strings. Then the r Min-hashing values of a string are concatenated to form a key, i.e., for string s, $key(s) = m_{h_1}(s) \oplus \cdots \oplus m_{h_r}(s)$, where $\oplus$ join the r hash keys into one key. The concatenated key is hashed into one of the M buckets for string s. Thus, the similarity strings are hashed into the same bucket, and the non-similar ones are filtered. In order to reduce the false negative rate, the above procedure will repeat b times, however, the min-hashing function is different for each iteration.

Algorithm 3: minHash Indexing Algorithm

```
Input: Binary Vectors $\vec{V}$; String Set List $SL$.
Output: LSH index based on Min-Hashing.
1  generate the $b * r$ hash functions;
2  foreach string $s$ in $SL$ do
3      $s_v$ =$getVector(V, SL)$; /*get the string vector */;
4      for $i = 1; i \leq b; i++$ do
5          $key$ =0 ;
6          for $j = 1; j \leq r; j++$ do
7              $key_i$ = $m_h(s_v)$;
8              $key$ = $key \oplus key_i$ ;
9          $bandNum$ =$HASH_{fun}(key)$;
10         store the string $s$ in the $bandNum$-th bucket;
11 return $LSHIndex$;
```

The procedure is illustrated in Algorithm 3. A trie tree is constructed for the strings in the each bucket, and the similarity string pairs can be obtained with Tire-join algorithm in each bucket. The final result of the similar string pairs can be get by merge the similar pairs in each bucket. More details are presented as follows.

The input of Algorithm 2 are compressed string binary vectors $\vec{V}$, which is built with Algorithm 1,string set list SL and the edit distance threshold τ. The procedure of Algorithm 2 can be divided into four steps. In the first step, the Min-Hashing based LSH index is constructed for the string set with the decompressing the binary vectors (line 3), the index procedure is presented in Algorithm 3.

In the second step, strings in the same bucket are obtained with the band number and the bucket number of the LSH index (lines 4-7). If there is only one string in the bucket, then this bucket can be skipped for it cannot make up pairs (lines 8-10). One key problem in this step is that the strings within the string edit distance threshold τ can be fall into the same bucket in at least one band with high probability by choosing the proper parameter of b and r.

A trie structure is built for the strings in each bucket in the third step (line 12). In this step, the similar string pairs within the threshold τ are evaluated on the strings falling into the bucket by the Trie-Traversal algorithms proposed in [8] (line 13).

In the forth step, the similarity pairs in the buckets of each band are merged together and returned (lines 14, 15).

C. Complexity analysis

First, compute Min-Hashing LSH takes $O(|\mathcal{U}|brn)$, and the time complexity of Trie-Traverse is $O(\tau|A_T|)$, where $|A_T|$ is the sum of the numbers of the active-node sets of all the trie nodes in the trie T, and the space complexity is $O(|T| + C_{max})$, where $|T|$ is the size of trie T, and C_{max} is the maximal value of the sum of the active nodes of leave node [8].

In the Hash^{ed}-Join index, the trie tree is built within each bucket, thus the time complexity of Hash^{ed}-Join is $O(Mbr(|A_T^H|\tau) + |\mathcal{U}|brn)$, and space complexity is $O(Mbr(|T|^H + C_{max}^H))$, where $|T|^H$ is the tire tree in LSH, M is the buckets number, and A_T^H is the maximal sum of the numbers of the active-node sets of all the trie nodes in the trie T in Hash^{ed}-Join, and the C_{max}^H is the the maximal value of the sum of the active nodes of ancestors of leave node in all the buckets in Hash$^e d$-Join index. Due to $|T|^H \ll T$, thus $O(Mbr(|T|^H + C_{max}^H)) < O(|T| + C_{max})$ and $O(Mbr(|A_T^H|\tau) + |\mathcal{U}|brn) < O(\tau|A_T|)$.

V. EXPERIMENTS

We conduct extensive experiments on real datasets to evaluate the quality, efficiency and scalability of Hash^{ed}-Join. The setup of the experiment, including the datasets, experimental environment and the default parameters configuration are first described, then the experiment results are reported.

A. Experiment Setup

All of the algorithms are implemented in Java SDK1.6 and run on a computer which is configured with Intel duo core E6550 2.33GHz CPU and 4G main memory running Ubuntu 10.04.

The Trie-Traversal algorithm [8] and All-Pair-Ed [24] are used as the baseline for the performance comparison. The optimal algorithm of All-Pair-Ed is implemented. In our implementation, the first $q * \tau + 1$ q-grams of each string are selected as the prefix after being sorted by their Inverse Document Frequency (short for IDF) value [25]. Filtering and pruning techniques proposed in [8] are not take into consideration in our implementation. The default of q is set to 2, which is recommended in [26]. The default parameters of Min-Hashing b is configured with 1 and r with 3, and the default bucket number M is set to 20.

Three real datasets are employed for the experimental evaluation, which are described in the following.

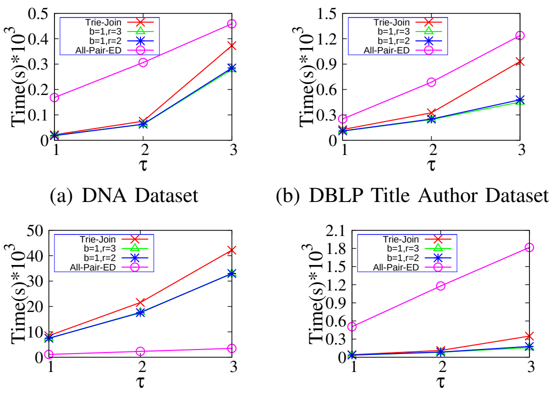
DBLP [1] is the bibliography records of computer fields, which includes more than 1.4 million publications and widely used in computer science fields as the benchmark, such as string similarity search [8]. Each record includes the title, the authors, and other information of the paper. We extract the dataset into two: one is consisted of the paper titles, denoted as DBLP Title; the other includes the title and the author names of the paper, denoted as DBLP TA.

TABLE III.: Statistics of the String Datasets

Dataset	$\|S\|$	Max Len	Min Len	Avg Len
DBLP Title	1158648	516	1	57.05
DBLP TA	1385925	1335	1	90.24
UniProt	508038	1992	2	341.06
DNA Seq	3190	60	60	60.0

UniProt [2] is a protein sequence database, which is widely used for sequence alignments, retrieval et al.

[1]http://www.informatik.uni-trier.de/ ley/db/
[2]http://www.uniprot.org/



(a) DNA Dataset
(b) DBLP Title Author Dataset
(c) UniProt Dataset
(d) DBLP Title Dataset

Figure 7: Performance Comparison on Datasets

DNA Sequences [3] has been developed to evaluate a *hybrid* learning algorithm (KBANN) that uses examples to inductively refine pre-existing knowledge. In total, 3190 DNA sequences are taken from Genbank 64.1, with average length 60.

The length distribution of the strings and the statistical information of the string datasets are shown in Fig.6(a) and Table III respectively.

For the larger dataset, such as DBLP Title, DBLP TA and UniProt datasets, datasets are sampled for experiment evaluation. The length distribution and the statistical information of the sampled dataset is shown in Fig.6(b) and Table IV respectively. Without pointing out specifically, the experiments are conducted on the sampled datasets.

TABLE IV.: Statistics of the Sampled Datasets

Dataset	$\|S\|$	Max Len	Min Len	Avg Len
DBLP Title	5885	163	3	58.14
DBLP TA	7040	284	8	81.89
UniProt	2578	1882	5	367.70
DNA Seq	3190	60	60	60.00

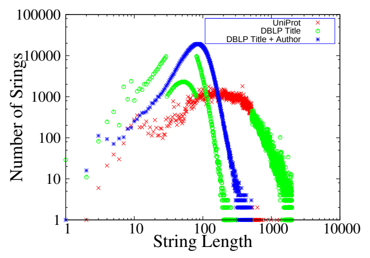
B. Efficiency

In this section, experiments for evaluating the efficiency of the Hash^{ed}-Join is conducted.

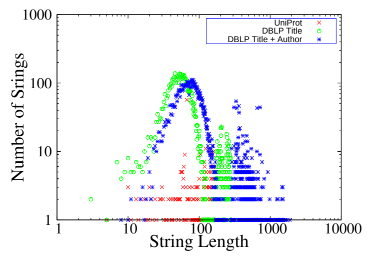
First the similarity join performance is conducted on the big string dataset DBLP Title. In this experiment, the size of string dataset is 579282 and the parameters of LSH is set to the default values. The string edit distance threshold τ varies from 1 to 2. The time cost of Hash^{ed}-Join on this dataset is 5100, 24073 seconds for τ is 1 and 2 respectively.

The efficiency are also compared Trie-Join and All-Pair-Ed [24] on the sampled datasets. Fig.7 illustrates the string join performance comparison with Trie-Join and All-Pair-Ed with different Min-Hashing parameters for the string edit distance threshold τ varying from 1 to 3.

[3]http://archive.ics.uci.edu/ml/datasets.html

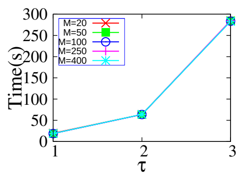


(a) Length Distribution of Datasets

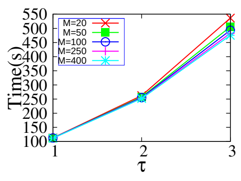


(b) Length Distribution of Sampled Datasets

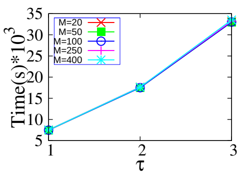
Figure 6: Length Distribution



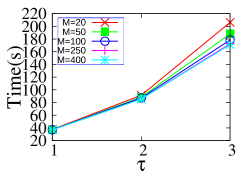
(a) DNA Dataset



(b) DBLP Title Author Dataset



(c) UniProt Dataset



(d) DBLP Title Dataset

Figure 8: Performance with Different Buckets on Datasets

The performance comparison in Fig.7 shows that, with the increasing of the threshold τ, the time cost of Trie-Join increases much faster than Hash^{ed}-Join. When τ is 3, Hash^{ed}-Join just takes about 50% $\sim$ 70% time of Trie-Join. This also confirms our observation in Section 3. With the increase of the hash function number in each band, the performance increases a little, because with the increasing of r, the probability of the strings hashed into the same bucket decreases. From the comparison result, we can see that both Trie-Join and Hash^{ed}-Join outperform the All-Pair-Ed on the datasets except the UniProt dataset. This is because that Trie-Join excels at short strings and the length of the UniProt dataset is 367, which is much longer. The average length of the string set is illustrated in Table IV.

From the above performance comparison experiment, conclusions can be drawn that:(1) Hash^{ed}-Join gains much performance improvement against Trie-Join with the increasing of the edit distance threshold; (2) As the Min-Hashing parameter r increases, the efficiency of the Hash^{ed}-Join also increases, however, the join quality will decrease, because the bigger of r, the less of the candidates in the collision bucket; (3) All-Pair-Ed is more effective than Trie-Join and Hash^{ed}-Join for long string set, which is illustrated by Fig.7(c).

We also evaluate the performance of Hash^{ed}-Join with different bucket number M with M varying from 20 to 400. Fig.8 illustrates the performance of Hash^{ed}-Join with different bucket number. When the of the bucket number increase, the performance of Hash^{ed}-Join increases a little, especially when the string edit distance threshold τ becomes larger.

C. *Quality*

The similarity join quality results of Hash^{ed}-Join on the 4 datasets are reported in this section. Within each bucket, Hash^{ed}-Join employs the trie join technique, therefore, the precision of Hash^{ed}-Join can not be computed directly. Therefore, we only take the *recall* measure into consideration, which is define as Eq.8.

$$recall = \frac{|retrieved \bigcap relevant|}{|relevant|}, \tag{8}$$

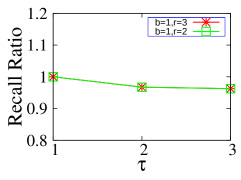
where the *relevant* refer to the string pairs in the datasets that satisfying the string edit distance threshold, i.e., $relevant = \{< s_i, s_j > | s_i, s_j \in S, i < j, dist_{ed}(s_i, s_j) \leq \tau\}$; and the *retrieved* represents the retrieved result using the Hash^{ed}-Join algorithm.

TABLE V.: Number of Similar String Pairs with Edit Distance τ

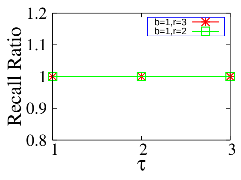
Dataset	$\tau = 1$	$\tau = 2$	$\tau = 3$
DBLP Title	1	3	7
DBLP TA	3	6	9
UniProt	1	252	402
DNA Seq	30	92	134

Table V summarizes the similar string pairs with different string edit distance τ varying from 1 to 3 on the 4 datasets, i.e., the number of the *relevant* result in the datasets. The *recall* ratio of Hash^{ed}-Join is demonstrated in Fig.9. On account of one band is enough for the join quality, the band number b is set to 1 for trading the performance.

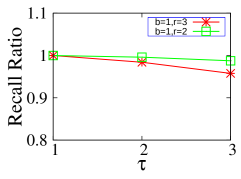
Fig.9 reveals the results of join quality with recall ratio metric. These results demonstrate that (1) with the increasing of the edit distance threshold τ, the recall ratio may reduce slightly. The reason is that strings within the edit distance threshold may be fallen into different buckets; (2) with the increase of the parameter r of Min-Hashing, the recall ratio may decrease, the Fig.9(d) shows this obviously. The result can be seen from Eq.5. The reason of recall ratio dropping is that with the increase



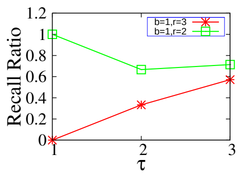
(a) DNA Dataset



(b) DBLP Title and Author Dataset

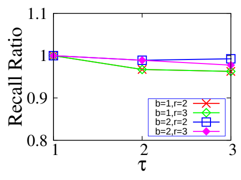


(c) UniProt Dataset

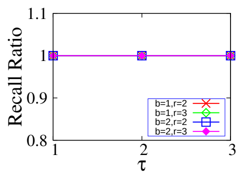


(d) DBLP Title Dataset

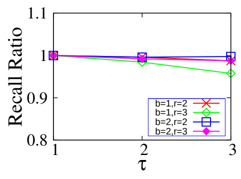
Figure 9: Join Quality with Hash Function Number r on Datasets



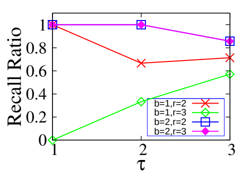
(a) DNA Dataset



(b) DBLP Title and Author Dataset

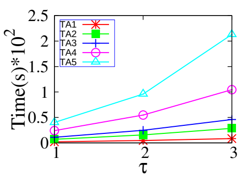


(c) UniProt Dataset

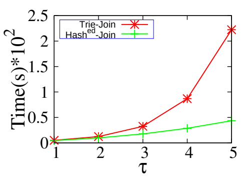


(d) DBLP Title Dataset

Figure 10: Join Quality with Band Number b on Datasets



(a) Scalability Evaluation of Hash^{ed}-Join



(b) Scalability Comparison

Figure 11: Scalability Evaluation

of the parameters r, the false negative may increase, thus the similar string pair may be filtered with the LSH.

However, the join quality in Fig.9(d) looks different with others when $r = 3$. The reason is that in DBLP Title dataset, there is only 1 similar string pair in the dataset when threshold τ is 1, which is illustrate in Table V. In addition, when $r = 3$, it may be missed with high probability, as a result, the recall ration is 0 if the similar pair missed. When the string edit distance threshold τ is equal to 2 and 3, there are 3 and 7 similar string pairs respectively in the dataset, and the recall can be increased in the experimental result. In consequence, the recall ratio looks different with Fig.9(c).

We also conduct the experiment to evaluate the join quality of Hash^{ed}-Join with different band number, which control the approximation factor of the result, and the result is demonstrated in Fig.10. The experiment results revel that increasing the band number will improve the join quality, and this conclusion is evidently shown in Fig.10(a) to Fig.10(d), especially for the DBLP Title dataset in Fig.10(d). However, when the band number b is 2, the quality improves slightly from Fig.10(a) to Fig.10(c), but this almost doubles up the time cost. Therefore, the band number $b = 1$ is enough for trading the quality and efficiency.

Of course, the quality increases by reducing the parameter r. Nevertheless, reducing the parameter r can increase the probability of non-similar strings as well, i.e., increasing the false positive. When the string edit distance threshold τ increases, by reducing the parameter r and increasing b, the result quality can be tuned.

D. Scalability

In this section, the scalability of Hash^{ed}-Join with respect to the size of the string datesets and the string edit distance threshold is evaluated. The default parameters of Min-Hashing LSH b, r and bucket number M are set to 1, 3 and 100 respectively. To evaluate the scalability, 5 groups of datasets are sampled from DBLP dataset with title and author, which are denoted as TA$_i$, $i = 1, ..., 5$, and the number of the strings varies from 2321 to 24314 respectively. Table VI outlines the statistical information of the 5 sampled datasets.

TABLE VI.: Statics of the Sampling Datasets for Scalability Experiment

Dataset	$\|S\|$	Max Len	Min Len	Avg Len
TA$_1$	2321	238	10	66.233
TA$_2$	4666	284	8	79.924
TA$_3$	7035	284	8	81.892
TA$_4$	14287	320	7	84.911
TA$_5$	24314	432	7	84.076

In term of the scalability with the data size, Fig.11(a) demonstrates the scalability of Hash^{ed}-Join on the datasets with different string numbers, the experimental result show that, with the increasing of the size of the dataset, Hash^{ed}-Join scales well. Fig.11(b) shows the scalability comparison with Trie-Join when τ varies from 1 to 5 on dataset TA$_2$. It can be seen that with the increasing of τ, the time cost of Trie-Join increases much faster than Hash^{ed}-Join. Fig.11(b) indicates that Hash^{ed}-Join scales much better than Trie-Join, especially when the string edit distance threshold becomes larger. For example, when $\tau = 5$, Hash^{ed}-Join takes less than 20% time cost of Trie-Join.

VI. Related Work

String similarity join has been studied extensively in database fields [4], [8], [26], [27]. It is important in many applications, such as data duplication detection [5], data cleaning [4], data integration [28].

S. Chaudhuri et al. [4] introduce the similarity join for data cleaning and implement it as a primitive operator *SSJoin* in the relation database and the k-prefix filter based on the pigeon hole principle is proposed. Ed-Join [27] explores the *mismatching* q-gram for the edit distance constraint: content mismatching and location mismatching. A. Arasu et al. [29] introduce a signature-based framework. Their signature is based the *partitioning* and *enumeration*. Based on the relationship between Hamming distance and Jaccard similarity, they propose PARTNUM for the candidate filtering. All-Pair-Ed [24] is based on the prefix filtering and follows an inverted list nested join style for the string similarity join. For each string, the first $q * \tau + 1$ q-grams are selected as the prefix, strings share q-gram with the prefix are verified by string edit distance evaluation. S.-H. Kim [30] studied the string matching under a restricted alphabet set.

Based the limitation of prefix filtering methods, [31]propose a cost model for selecting prefix. Top-k string similarity search with edit-distance constraints is proposed [32], which improves the pruning effective by using pivotal entries. String similarity joins with synonyms is proposed [33], which is based on the term expansion framework.

Landmark-Join [34] adopt the similar processing path as ours, based on hash join technique, which using q-bucket partitioning and local upper bound calculation for speed up similarity evaluation.

Recently, tree index based are proposed for string similarity join [8], [11], [12]. Z. Zhang et al. [11] propose the B^{ed}-tree with the edit distance constraint for string search. They employ three kinds of string orders: dictionary order, gram counting order and gram counting and location order for indexing the strings that satisfying the *lower bounding* for similarity string query. H. Lu [35] proposed a *filter-and-refine* framework based on hash-algorithm for probabilistic spatio-temporal joins, they use a R-tree variant for pruning candidates.

The trie structure is employed for searching the similar strings under string edit distance constraint [36]. The Trie-Join [8] introduces the trie structure for prefix pruning, which benefits the string similarity join with edit distance constraint on short strings. PreJoin [12] improve Trie-Join by reducing pruning step.

Parallel evaluation of massive string join on cluster are also studied using MapReduce [37]–[39].

VII. Conclusion and Future Work

In this paper, an approximate string similarity join framework Hash^{ed}-Join, using the Min-Hashing based LSH and the Trie-Join techniques, is studied.

Hash^{ed}-Join employs the LSH techniques to filter the similar strings firstly, within each bucket, the Trie-Join technique is applied for evaluating the similar string pairs that satisfying the edit distance constraint approximately. The relationship between the parameters of Min-Hashing LSH and the string edit distance is established, which ensures that the similar string with distance threshold can be retrieved with high probability theoretically. An empirical study on real datasets indicates that Hash^{ed}-Join can effectively processing string join with high quality and better performance. As a future work, parallel evaluating on cluster with massive strings will be considered.

Acknowledgment

The authors are grateful to the anonymous referees for their valuable comments and suggestions to improve the presentation of this paper.

References

[1] P. Yuan, C. Sha, and S. Yi, "Hash^{ed}-join: Approximate string similarity join with hashing," in *Database Systems for Advanced Applications*. Springer, 2014, pp. 217–229.

[2] M. Hadjieleftheriou and D. Srivastava, "Weighted Set-Based String Similarity," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pp. 1–12, 2010.

[3] W. Wang, C. Xiao, X. Lin, and C. Zhang, "Efficient approximate entity extraction with edit distance constraints," in *SIGMOD*, 2009, pp. 759–770.

[4] S. Chaudhuri, V. Ganti, and R. Kaushik, "A Primitive Operator for Similarity Joins in Data Cleaning," in *ICDE*, 2006, pp. 5–5.

[5] C. Xiao, W. Wang, X. Lin, and J. Yu, "Efficient similarity joins for near duplicate detection," in *WWW*, 2008, pp. 131–140.

[6] M. Henzinger, "Finding near-duplicate web pages: a large-scale evaluation of algorithms," in *SIGIR*, 2006, pp. 284–291.

[7] S. Chaudhuri, V. Ganti, and R. Kaushik, "Data debugger: An operator-centric approach for data quality solutions," *IEEE Data Eng. Bull*, vol. 29, no. 2, pp. 60–66, 2006.

[8] J. Wang, J. Feng, and G. Li, "Trie-Join:Efficient Trie-based String Similarity Joins with Edit Distance Constraints," *VLDB*, vol. 1, no. 1, pp. 933–944, 2010.

[9] R. Wagner and M. Fischer, "The string-to-string correction problem," *JACM*, vol. 21, no. 1, pp. 168–173, 1974.

[10] W. Masek and M. Paterson, "A faster algorithm computing string edit distances," *Journal of Computer and System Sciences*, vol. 20, no. 1, pp. 18–31, 1980.

[11] Z. Zhang, M. Hadjieleftheriou, B. Ooi, and D. Srivastava, "B^{ed}-tree: an all-purpose index structure for string similarity search based on edit distance," in *SIGMOD*, 2010, pp. 915–926.

[12] K. Gouda and M. Rashad, "Prejoin: An efficient trie-based string similarity join algorithm," in *INFOS*. IEEE, 2012, pp. DE–37.

[13] J. Qin, X. Zhou, W. Wang, and C. Xiao, "Trie-based similarity search and join," in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. ACM, 2013, pp. 392–396.

[14] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *STOC*, 1998, pp. 604–613.

[15] C. Urbani, "A statistical table for the degree of coexistence between two species," *Oecologia*, vol. 44, no. 3, pp. 287–289, 1979.

[16] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet Physics Doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[17] A. Broder, "On the resemblance and containment of documents," in *Proceedings of Compression and Complexity of Sequences*, 1997, pp. 21–29.

[18] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe LSH: Efficient indexing for high-dimensional similarity search," in *VLDB*, 2007, pp. 950–961.
[19] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*, 2013. [Online]. Available: http://infolab.stanford.edu/~ullman/mmds.html
[20] M. Narayanan and R. Karp, "Gapped local similarity search with provable guarantees," *Algorithms in Bioinformatics*, pp. 74–86, 2004.
[21] A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher, "Min-Wise Independent Permutations," *Journal of Computer and System Sciences*, vol. 60, no. 3, pp. 630–659, 2000.
[22] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[23] P. Yuan, C. Sha, X. Wang, B. Yang, A. Zhou, and S. Yang, "XML Structural Similarity Search Using MapReduce," *WAIM*, pp. 169–181, 2010.
[24] R. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *WWW*, 2007, pp. 131–140.
[25] R. Baeza-Yates and B. Ribeiro-Neto, *Modern information retrieval*. Addison Wesley, 1999.
[26] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava, "Approximate string joins in a database (almost) for free," in *VLDB*, 2001, pp. 491–500.
[27] C. Xiao, W. Wang, and X. Lin, "Ed-Join: an efficient algorithm for similarity joins with edit distance constraints," *VLDB*, vol. 1, no. 1, pp. 933–944, 2008.
[28] W. Cohen, "Integration of heterogeneous databases without common domains using queries based on textual similarity," in *SIGMOD*, 1998, pp. 201–212.
[29] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *VLDB*, 2006, pp. 929–931.
[30] S.-H. Kim, C.-S. Ock, and H.-G. Cho, "An efficient composite-alphabet transform for string matching under a restricted alphabet set," *Journal of Computers*, vol. 8, no. 7, pp. 1804–1809, 2013.
[31] J. Wang, G. Li, and J. Feng, "Can we beat the prefix filtering?: an adaptive framework for similarity join and search," in *SIGMOD*, 2012, pp. 85–96.
[32] D. Deng, G. Li, J. Feng, and W.-S. Li, "Top-k string similarity search with edit-distance constraints." ICDE, 2013, pp. 925–936.
[33] J. Lu, C. Lin, W. Wang, C. Li, and H. Wang, "String similarity measures and joins with synonyms," *SIGMOD*, pp. 373–384, 2013.
[34] K. Narita, S. Nakadai, and T. Araki, "Landmark-join: Hash-join based string similarity joins with edit distance constraints," in *Data Warehousing and Knowledge Discovery*, ser. Lecture Notes in Computer Science, A. Cuzzocrea and U. Dayal, Eds. Springer Berlin Heidelberg, 2012, vol. 7448, pp. 180–191.
[35] H. Lu, B. Yang, and C. S. Jensen, "Spatio-temporal joins on symbolic indoor tracking data," in *ICDE*. IEEE, 2011, pp. 816–827.
[36] S. Chaudhuri and R. Kaushik, "Extending autocompletion to tolerate errors," in *SIGMOD*, 2009, pp. 707–718.
[37] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using mapreduce," in *SIGMOD*. ACM, 2010, pp. 495–506.
[38] D. Deng, G. Li, S. Hao, J. Wang, J. Feng, and W.-S. Li, "Massjoin: A mapreduce-based method for scalable string similarity joins," in *ICDE*. IEEE, 2014.
[39] A. Metwally and C. Faloutsos, "V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 704–715, 2012.

Peisen Yuan received his Ph.D. degree in Computer Software and Theory from Fudan University, Shanghai, China in 2011. He is currently lecture at the College of Information Science and Technology, Nanjing Agricultural University, Nanjing, China. His current research interests include Web data management, big data techniques and agricultural information etc.

Shougang Ren received his B.S. degree in welding technology from Nanjing University of Aeronautics and Astronautics, China in June 1999 and his M.S. degree and his Ph.D. degree in mechatronic engineering from Nanjing University of Aeronautics and Astronautics, China in June 2005. His current research interest includes artificial intelligence and software engineering.

Dechang Pi received the B.S., M.S., and Ph.D. degrees from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 1994, 1997, and 2002, respectively. He is currently a Professor and the Associate Dean with the College of Computer Science and Technology. His current research interests include data mining and big data analysis.