Computing and Informatics, Vol. –, –, 1–20, V 2016-May-24

AN EFFICIENT STRING EDIT SIMILARITY JOIN ALGORITHM

Karam Gouda[†], Metwally Rashad[*]

[†] *Faculty of Computers & Informatics Benha University, Benha, Egypt*
[*] *Faculty of Information Technology, University of Pannonia, Hungary.*
e-mail: `karam.gouda@fci.bu.edu.eg, metwally.rashad@virt.uni-pannon.hu`

Abstract. String similarity join is a basic and essential operation in many applications. In this paper, we investigate the problem of string similarity join with edit distance constraints. A trie-based edit similarity join framework has been proposed recently. The main advantage of existing trie-based algorithms is support for similarity join on short strings. The main problem is when joining long and distant strings. These methods generate and maintain lots of similar prefixes called active nodes which need to be further removed in a subsequent pruning phase. With large edit distance, the number of active nodes becomes quite large. In this paper, we propose a new trie-based join algorithm called `PreJoin`, which improves upon current trie-based join methods. It efficiently finds all similar string pairs using a novel active-node generation method, which minimizes the number of generated active nodes by applying the pruning heuristics early in the generation process. The performance of `PreJoin` is scaled in two different ways: First, a dynamic reordering of the trie index is used to accelerate the search for similar string pairs. Second, a partitioning method of string space is used to improve performance on large edit distance thresholds. Experiments show that our approach is highly efficient for processing short as well as long strings, and outperforms the state-of-the-art trie-based join approaches by a factor five.

Keywords: String data, edit distance, trie-based approaches, similarity join

Mathematics Subject Classification 2010: 68–P20

1 INTRODUCTION

In the modern society, string data are becoming ubiquitous, and its management has taken on particular importance in the past few years. Similarity join of string data has become an essential operation in many applications, including data integration [4], data cleaning [9], web page detection [8], record linkage [16], pattern recognition [11], etc. It is also adopted in the industry solutions.

Given two string collections, string similarity join finds similar string pairs from both collections. Many similarity functions have been proposed to quantify the similarity among strings, such as Jaccard similarity, Cosine similarity and edit distance. In this paper, we study string similarity join with edit distance constraints (abbreviated *string edit similarity join* [3]). Given two strings, their string edit distance measures the minimum number of edit operations (insertion, deletion and substitution) performed on one of them to get the other. Edit distance has two distinctive advantages over alternative distances or similarity measures: (a) it reflects the ordering of tokens in the string; and (b) it allows non-trivial alignment. These properties make edit distance a good measure in many application domains, e.g., to capture typographical errors for text documents, and to capture similarities for Homologous proteins or genes.

Many of the existing algorithms to the string edit similarity join problem, such as Part-Enum [1], All-Pairs-Ed [2], Ed-Join [18], Pass-Join [12] and MassJoin [5], employ the filter-and-verify paradigm, where q-gram inverted indexes are used to quickly filter out many of the unpromising string pairs and generate candidate pairs. Then, these candidates are verified, i.e., whether each pair is within the edit threshold, by a string edit distance algorithm. The q-gram based methods have the following disadvantages. First, they are inefficient with short strings, since they cannot select high-quality signatures (q-grams) for short strings; thus, they may generate a large number of candidate pairs which need to be further verified. Second, they cannot support dynamic data updates. Since the data updates change the weights of signatures, the methods need to reselect signatures, rebuild indexes and rerun their algorithms from scratch. Finally, they involve large index sizes as there could be large numbers of signatures.

Recently, to address the above-mentioned problems, a trie-based edit similarity join framework has been proposed [15, 6]. The idea at the heart of this approach is based on the observation that similar strings should have similar prefixes. Thus, if two different strings are not similar on a particular pair of their prefixes, they are not similar strings. As many data strings have common prefixes, using the trie structure to index data strings not only minimizes the index size but allows for efficient string similarity join using prefix pruning. Several algorithms have been introduced to traverse the trie index to find similar string pairs [15, 6]. While traversing the trie index, these methods generate and maintain similar prefixes called active nodes. If a similar prefix pair correspond to data strings, this pair is reported as an answer. Although trie-based similarity join is free from the costly verification phase, the major challenge facing current trie-based join methods is when joining long and distant strings. With large edit distance thresholds, the number of active nodes becomes quite large. For example, if we allow three edit errors, all the trie nodes on the highest four levels will be active nodes. The situation will be even worse for the applications where strings have large-sized alphabets; e.g., Unicode or CJK characters [17].

To meet this challenge, current methods [15, 6] use many pruning techniques such as *length* pruning, *single-branch* pruning and *count* pruning to minimize the number of active nodes. Unfortunately, these pruning techniques are used in a separate phase subsequent to the generation phase. The computation and space overheads caused by these two phases makes the existing approaches inefficient for processing large data sets with long strings and higher edit distance thresholds. In this paper, we propose a new trie-based edit similarity join algorithm called `PreJoin`, which minimizes the number of generated active nodes. To the best of our knowledge, we are the first to address this problem. `PreJoin` uses preorder traversal combined with a novel active node generation method. Instead of generating and maintaining active nodes, and then refining them in a subsequent phase, the new method encapsulates the previous pruning techniques to generate and maintain the actual active nodes. Thus, the space required for maintaining active nodes is minimized and the overhead of applying the pruning techniques in a subsequent phase is removed. Moreover, `PreJoin` assumes that the active nodes of each trie node to be available when it is visited. Thus, the generation method chooses to generate active nodes of all children of a currently processed node at once, which reduces *redundant computations* inherent to active nodes computation.

The performance of `PreJoin` is scaled in two different ways: First, we modify the tree traversal in `PreJoin` to be *dynamic preorder*, that is, at each node, the next sub-trie to be processed is determined according to an effective ordering methodology. Second, the string space is partitioned in order to improve the performance on large edit distance thresholds. The extended version of `PreJoin` is called `PreJoin-Plus`. A preliminary idea of the work presented here is appeared as an abstract in [7]. The work in the current paper has significantly extended the idea with respect to the underlying methodology and the experimental evaluation.

To summarize, we make the following contributions:

1. We propose a new trie-based edit similarity join algorithm called `PreJoin`, which improves upon current trie-based join methods. It efficiently finds all similar string pairs using a new active-node set generation method.
2. We scale the performance of `PreJoin` in two different ways: First, a dynamic reordering of the trie index is used to accelerate the search for similar string pairs. Second, a partitioning method of string space is used to improve performance on large edit distance thresholds. We call this extension `PreJoin-Plus` algorithm.
3. Experimental results show that our algorithms are highly efficient for processing short as well as long strings, and outperform the state-of-the-art trie-based join approaches.

The rest of the paper is organized as follows: Section 2 introduces preliminaries such as problem statement and the working principle of trie-based similarity join. Section 3 presents the basic PreJoin algorithm and the novel active nodes generation method. Section 4 presents the scaling methodologies to `PreJoin`, such as the dynamic reordering of the index and partitioning of the string space. Experimental results are given in Section 5. Finally, Section 6 concludes the paper.

2 PRELIMINARIES

2.1 Problem statement

Let Σ be a finite alphabet of symbols σ_i $(1 \leq i \leq |\Sigma|)$; each symbol is also called a character. A string s is an ordered array of symbols drawn from Σ. We use $|s|$ to denote the length of s, and $s[i]$ to denote the *i-th* character of s, and $s[1..j]$ to denote the prefix of s, i.e., a substring of s starting from its beginning character to its j-th character. Each string s is also assigned an identifier *sid*. The edit distance between two strings s_1 and s_2, denoted as $ed(s_1, s_2)$, is the minimum number of single-character edit operations, including insertion, deletion and substitution, needed to transform s_1 to s_2, or vice versa. For example, the edit distance between $s_1 =$ "Jim Gray" and $s_2 =$ "Jim Grey" is 1, since s_1 could be transformed to s_2 with a minimal of a substitution operation that replaces the character at position 7, $s_1[7]$, with the new character e. The edit distance between two strings s_1 and s_2 can be computed in $O(|s_1||s_2|)$ time and $O(\min(|s_1|, |s_2|))$ space using the standard dynamic programming [14].

Given two sets of strings $\mathcal{R}$ and $\mathcal{S}$, a similarity join with edit distance threshold τ (or string edit similarity join [3]) returns pairs of strings, one from each set, such that their edit distance is less than τ, that is, string edit similarity join of $\mathcal{R}$ and $\mathcal{S}$ returns $\{\langle r, s\rangle : ed(r, s) \leq \tau, r \in \mathcal{R}, s \in \mathcal{S}\}$. The best known edit distance computation algorithm tests if $ed(r, s) \leq \tau$ in $O(\tau \cdot \min(|r|, |s|))$ time [13]. In this paper, for the ease of exposition, we focus on the self-join case, i.e., $\mathcal{S} = \mathcal{R}$.

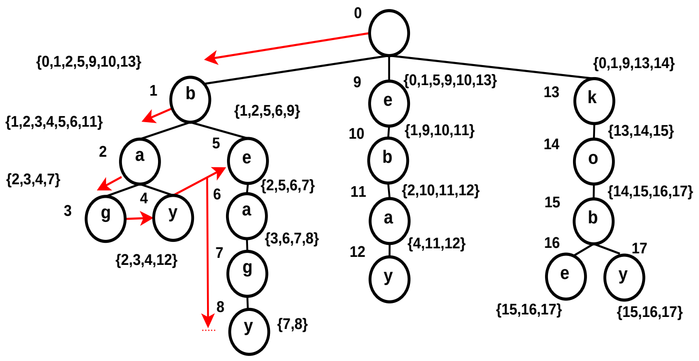
Several algorithms have been introduced in the previous studies to solve the string edit similarity join problem, such as Part-Enum [1], All-Pairs-Ed [2], ED-Join [18], Pass-Join [12], MassJoin [5] and Trie-PathStack [15]. Trie-PathStack and our approach `PreJoin` are trie-based methods, whereas the others are q-gram based methods. Next, we introduce the working principles of the trie-based string similarity join framework.

2.2 Trie-based string similarity join

In trie-based similarity join, a trie is used to index all strings in the dataset $\mathcal{R}$. The trie structure is a rooted tree, where each path from the root to a node represents a (prefix of) string in $\mathcal{R}$, and every node on the path has a label of a corresponding character in that string. Thus, every trie leaf corresponds to a string in $\mathcal{R}$. Figure

SID	strings
s_1	ba
s_2	bag
s_3	ebay
s_4	bay
s_4	kobe
s_6	koby
s_7	beagy

(a)



(b)

Fig. 1. (a) a sample string dataset; (b) a trie index. The set appearing next to each trie node is its active node set at $\tau = 1$.

1(b) shows the trie structure of the sample data set given in Figure 1(a). Numbers on nodes are node identifiers.[1] The node numbered 12, e.g., has the corresponding character "y", and "ebay" is its corresponding string.

Note that, strings with the same prefix share the same ancestor nodes on the trie. Thus, if two different prefixes are not similar, the groups of strings sharing these prefixes are not similar too. Based on this observation, a pruning technique called *dual subtrie pruning* is proposed in [15, 6]. It works as follows. Given a trie node n and an edit distance threshold τ, a trie node m is called an *active node* for n if $ed(p_n, p_m) \leq \tau$, where p_i is the prefix corresponding to the node i. Thus, if a node m is not an active node for a node n, then the strings corresponding to the descendants of m will not be similar to the strings corresponding to the descendants of n. For example, consider the trie in Figure 1 and suppose $\tau = 1$. Since the node 14 is not an active to the node 1, then all the strings with prefix "ko", i.e., strings with *sid*s s_5 and s_6 in the data set, are not similar to the strings with prefix "b", i.e., strings with *sid*s s_2, s_4 and s_7.

In [15, 6], several algorithms have been proposed to traverse the trie index to

[1] Hereafter, we use i and n_i interchangeably as a node identifier.

find similar string pairs using dual subtrie pruning. Also, different pruning techniques such as *length* pruning, *single-branch* pruning and *count* pruning have been introduced to improve performance. Trie-based similarity join works as follows. For each encountered node n in the trie traversal, its active-node set, denoted $\mathcal{A}_n$, is computed. If p_n is a data string, then for every active node $m \in \mathcal{A}_n$ if p_m is a data string too then $\langle p_n, p_m \rangle$ is a similar string pair. Active-node sets are computed incrementally using a method called ICAN [10] (Incrementally Computing Active Nodes) as follows. Initially, the root of the trie represents an empty string ϵ, and its corresponding active-node set includes all trie nodes m with depth no larger than τ. Suppose the active-node set of a given node n, $\mathcal{A}_n$, is computed. ICAN computes the active-node set of each child of n from the active-node set $\mathcal{A}_n$. Given a trie node m, the time complexity of computing $\mathcal{A}_m$ from its parent's active-node set is $O(\tau \cdot |\mathcal{A}_m|)$, since each active node only can be computed from its ancestors within τ steps.

Algorithm: *Trie-Traverse*$(\mathcal{R}, \tau)$

```
Input: $\mathcal{R}$: a collection of strings;
       $\tau$: edit-distance threshold.
Output: $P = \{(s \in \mathcal{R}, t \in \mathcal{R}) : ed(s,t) \le \tau\}$.
1.  $T$ = new trie($\mathcal{R}$);
2.  Let $r$ be the root id of the trie $T$;
3.  $\mathcal{A}_r = \{m$ : a trie node such that $|p_m| \le \tau\}$;
4.  for each child node of $r$, $n$ do
5.      $P \cup= findSimilarPair(n, r, \mathcal{A}_r)$;
6.Function $findSimilarPair(c, p, \mathcal{A}_p)$
7.  $\mathcal{A}_c = calcActiveNode(c, \mathcal{A}_p)$;
8.  Pruning($\mathcal{A}_c$);
9.  if $c$ is a leaf node then
10.     $P_c = outputSimilarPair(c, \mathcal{A}_c)$;
11.  for each child node of $c$, $d$ do
12.     $P_c \cup= findSimilarPair(d, c, \mathcal{A}_c)$;
12.Function $outputSimilarPair(n, \mathcal{A}_n)$
13.  for each leaf node $l \in \mathcal{A}_n (n \ne l)$ do
14.      $P_n = \{(n, l)\}$;
```

Fig. 2. Trie-Traverse Algorithm

Trie-Traverse is a preorder traversal method introduced in [15] as a basic trie-based similarity join algorithm. It first constructs a trie index for all strings in $\mathcal{R}$. It then traverses the trie in preorder, and computes the active-node set of a node n, $\mathcal{A}_n$, based on its parent's active-node set. Preorder traversal guarantees that, for each node, its parent's active-node set is computed before its own active-node set. The pseudo-code of Trie-Traverse is given in Figure 2. Since Trie-Traverse visits each node in the trie T, hence, the time complexity of Trie-Traverse is given as $O(\tau \cdot |\mathcal{A}_T|)$, where $|\mathcal{A}_T| = \sum_{n \in T} |\mathcal{A}_n|$. Figure 1 shows the active-node sets computed in the preorder traversal when $\tau = 1$. The arrows in the figure show the order in this traversal.

The major challenge facing trie-based similarity join is when τ is large. The higher the edit distance threshold τ, the larger the number of active nodes. Moreover, when $\mathcal{R}$ is large and consists of long strings, the corresponding trie becomes very large. The main objective of this paper is to scale trie-based similarity join on long and distant strings. While another traversal method called Trie-PathStack has been introduced in [15, 6] to speed up trie-based join, in this paper, we present a new method called `PreJoin`, which uses the preorder traversal combined with a novel active-nodes generation method to optimize the number of active nodes.

3 `PREJOIN` ALGORITHM

In trie-based similarity join algorithms, active-node sets are generated in a separate phase (see, e.g., line 7 in Figure 2) by the ICAN method [10]. A number of pruning heuristics are then used in a subsequent phase in the algorithm (e.g., line 8 in Figure 3) to optimize the size of each generated active-node set. The computation and space overheads caused by these two phases are the main reasons of why the current trie-based similarity join is not the best choice for processing long and distant strings. Here, we devise a novel active nodes generation method, which minimizes the number of generated active nodes by applying the pruning heuristics early in the generation process. Therefore, the pruning phase is not required. This new generation method scales trie-based join framework on long and distant strings.

Combining the preorder traversal of the trie index with the new active nodes generation method, a new algorithm, called `PreJoin`, is developed. Figure 3 outlines `PreJoin` algorithm. It works as follows. Given a dataset $\mathcal{R}$, each string $s \in \mathcal{R}$ is inserted into the trie index according to the order of stings in the dataset. Recall that each trie leaf corresponds to a data string. Data strings which are contained in other data strings are represented by intermediate trie nodes. Thus, in our model, a trie node representing a data string is identified by the logical variable EOS (stands for End Of String). `PreJoin` visits nodes in preorder as in Trie-Traverse. However, `PreJoin` differs from Trie-Traverse in that: first, in addition to constructing the active-node set for the next child node to be visited as in Trie-Traverse, it also constructs the active-node sets for all the child node siblings; thus reducing *redundant computations*. Consequently, active-node set of each node will be available when reaching that node in the traversal. Second, `PreJoin` does not follow the fixed order imposed by the trie structure during traversal, it instead *virtually reorders* the children of each processing node to identify *significant sub-tries* to be traversed next. Finally, `PreJoin` employs a novel active nodes generation method that (1) avoids adding as many active nodes as possible into the active-node sets during active nodes generation, and (2) generates active nodes of a given trie node by investigating relatively deeper subtries rooted at the parent's active-nodes.

Algorithm: `PreJoin` $(\mathcal{R}, \tau)$

```
Input: $\mathcal{R}$: a collection of strings; $\tau$: an edit-distance threshold.
Output: $P = \{(s \in \mathcal{R}, t \in \mathcal{R}) : ed(s,t) \le \tau\}$.
1.  $T$ = new Trie($\mathcal{R}$);
2.  $Pre\_Traverse(root)$;
3.Procedure $Pre\_Traverse(t)$
4.  impose an order on $t$ children;
5.  let $CN_t = \{n_1, \ldots, n_k\}$ contain children of $t$ in the given order;
5.  for each $n_i$ do
7.     if $n_i$ is $EOS$ then $Out\_Similar(n_i, CN_t, \mathcal{A}_{n_i}, i, \tau)$;
8.     if $n_i$ is a leaf then continue;
9.     $Gen\_ActiveNode(n_i, CN_t, \mathcal{A}_{n_i}, i, \tau)$;
10.    $Pre\_Traverse(n_i)$;
11.Function $Gen\_ActiveNode(n_i, CN_t, \mathcal{A}_{n_i}, i, \tau)$
12.    for each node $m \in \mathcal{A}_{n_i}$ at distance $d$ do
13.       if $n_i^c == m^c$ then $Push\_down(n_i, m, d, \tau, 1)$;
14.       else $Push\_down(n_i, m, d, \tau, 0)$;
    /* $n_j, j > i$ are also active nodes to $n_i$ with distance 1 */
15.    for each $n_j$, $j > i$ do $Push\_down(n_i, n_j, 1, \tau, 0)$;
```

Fig. 3. `PreJoin` Algorithm

3.1 Novel Active Nodes Generation Method

3.1.1 Pruning Rules

The new active nodes generation method avoids adding as many active nodes as possible into the active-node sets by enforcing the following rules during the generation process.

RULE I: The first rule is to exploit the *symmetry property* of the string edit distance early in the generation. This property states that for any two strings s_1 and s_2, $ed(s_1, s_2) = ed(s_2, s_1)$. Note that a similar concept is used in Trie-PathStack. But, it is used there in a subsequent pruning phase, not in the generation phase as in `PreJoin`. There are two cases where we can apply the symmetry property.
Case 1: Suppose n is a trie node at level i which is currently under processing in

the traversal. There are two sub-cases: (1) Each ancestor node m of n, which is at level $i - j, j = 1 \ldots \tau$, is an active node of n within the edit distance j, since j *deletion* operations are required to transform the corresponding string of n, p_n, into the corresponding string of m, p_m. (2) Each descendant node m of n, which is at level $i + j, j = 1 \ldots \tau$, is an active node of n within the edit distance j, since j *insertion* operations are required to transform the corresponding string of n, p_n, into the corresponding string of m, p_m. Based on the symmetry property, our generation method does not insert any ancestor or descendant active node m of n into $\mathcal{A}_n$. Nevertheless, when n is of type *EOS*, the function *Out-Similar* (Line 7, Figure 3) searches for descendants m of type *EOS* which are at distance τ from n, and then outputs the strings corresponding to n and m as similar pairs. Note that *Out-Similar* does not handle ancestor active nodes because of the symmetry property. As an example, although the trie nodes n_1, n_2, n_3, n_4 in Figure 1(b) are active to the node n_2 and included in $\mathcal{A}_{n_2}$ by Trie-Traverse algorithm, they are not inserted into $\mathcal{A}_{n_2}$ by `PreJoin` according to our generation method (Figure 4). However, the similar pairs (s_1, s_2) and (s_1, s_4) will be output by the function *Out-Similar* when n_2 is accessed.

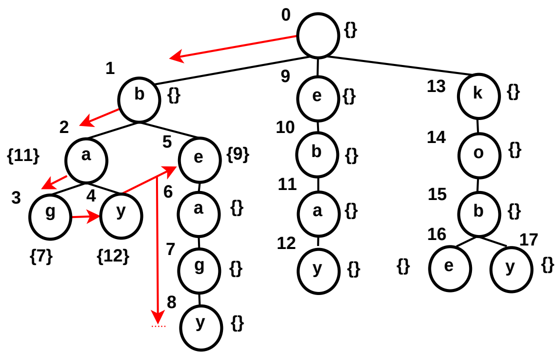


Fig. 4. Pre-order traversal plus active-node set generation in `PreJoin`, $(\tau = 1)$.

Case 2: Also based on the symmetry property, our method does not allow an already traversed node to be active for n – the current node to be processed. For example, suppose that n_{13} of Figure 1(b) is the currently processing node. The active-node set $\mathcal{A}_{n_{13}}$ does not include the nodes n_1 and n_9, since they are already processed. However, to guarantee completeness, the function *Out-Similar* will also be responsible of dealing with this case as follows. Suppose n is the trie node currently under processing in the traversal, and is of type *EOS*. For each active node $m \in \mathcal{A}_n$ at distance d from n, *Out-Similar* searches in the hight $\tau - d$ subtrie rooted at m for any node of type *EOS* to output. It also searches the hight $\tau - 1$

subtrie[2] rooted at each n's sibling for EOS nodes to output. For example, let n_2 (Figure 4) be the current processing node. Since it is of type EOS, the subtries rooted at n_5 and n_{11} are processed by *Out-Similar*.

RULE II: The second rule taken by the generation method is that: in the process of generating the active-node set $\mathcal{A}_n$ of a node n from its parent's active-node set, the siblings of n are not inserted into $\mathcal{A}_n$ even though those siblings are active nodes of n. [3] Nevertheless, later on, when n becomes the current node to be processed, the method considers those siblings as active nodes and the subtrie rooted at each sibling will be investigated (Line 15, Figure 3). As an example, using our method, the trie nodes n_9 and n_{13} in Figure 4 are not inserted into $\mathcal{A}_{n_1}$. But later on, when creating active-node sets of n_1's children, the subtries rooted at n_9 and n_{13} are investigated, taking into account they are at edit distance one of n_1.

Applying the previous pruning rules minimizes the number of generated active nodes. Figure 4 shows the active nodes generated by `PreJoin` when $\tau = 1$ of the sample dataset given in Figure 1(a). Comparing these against the ones generated by Trie-Traverse (Figure 1(b)) - a total of 4 active nodes instead of 76 active nodes with Trie-Traverse. Thus, the space used for holding active nodes in this simple example is an order of magnitude smaller than that of previous algorithms. Reducing the memory required to hold active nodes from the beginning would allow `PreJoin` to cope with larger τ. Below, we show how the new generation method computes active-node sets by investigating relatively larger subtries which enables `PreJoin` to deal with long strings. Moreover, since `PreJoin` assumes that the active-node set of a node to be available when it is visited, the generation method chooses to generate active-node sets of all children of a currently visited node at once. Thus, each subtrie is searched only once, removing most of the duplicated computations. Hence, `PreJoin` is capable of dealing with long and distant strings efficiently.

3.1.2 Active nodes Construction

First, let n^c be the character stored at the node n. The active nodes computation is outlined in `PreJoin` under the function *Gen_ActiveNode* (Figure 3, lines 11-15). Suppose n is the currently processing node in `PreJoin`. `PreJoin` assumes that $\mathcal{A}_n$ is already available. To generate the active-node sets of the children of n, the generation method works as follows. For each active node $m \in \mathcal{A}_n$, and at distance $d \leq \tau$ from n, the sub-trie rooted at m is searched. The sub-tries rooted at the unprocessed siblings of n are also searched, since they are active nodes of n at distance $d = 1$, but they are not included in $\mathcal{A}_n$ according to the second pruning rule. The trie level at which the search can reach in each sub-trie depends on the

[2] Recall, each sibling of n is at distance 1 from n.

[3] Any two siblings are within edit distance one because one substitution operation is required to transform the corresponding string of one sibling into the corresponding string of another.

characters n^c and m^c. We have two cases: (1) m^c differs from n^c. In this case, the search can reach up to the level $l_m + (\tau - d) + 2$, where l_m is the trie level of the node m; and (2) m^c matches n^c. Here, we can reach up to the level $l_m + (\tau - d) + 1$. Figure 5 illustrates these two cases. Next, we show how to search the sub-tries and generate new active nodes.

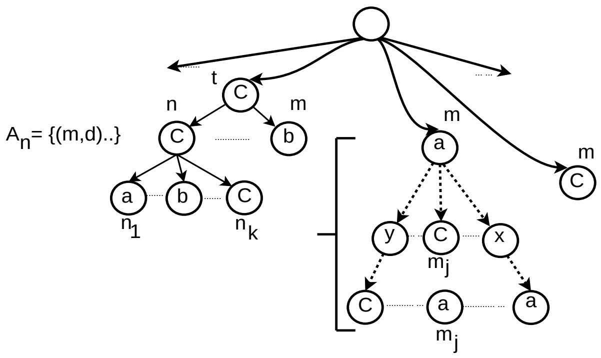


Fig. 5. Computing active-node sets of the children of n.

Considering the active node m: The relation of the active node m to the children of n is determined based on whether m^c matches n^c or not. First, suppose that m^c matches n^c. The node m will be at distance $d+1$ from each child n_i of n if either m^c matches n_i^c or not. This is because d edit operations are required to transform p_m to p_n and another deleting operation is required for n_i^c. Second, let m^c differ from n^c. The node m will be at distance $d + 1$ from each child n_i of n if m^c differs from n_i^c, since d edit operations are required to transform p_m to p_n and another deleting operation is required for n_i^c. Otherwise, if m^c matches n_i^c, the node m will remain at distance d from n_i, since the substitution operation between m^c and n^c is replaced by an insertion operation of n^c while transforming p_{n_i} into p_m.

Considering the descendants of the active node m: A descendant m_j of m positioned at level l_m+l, $l \leq \tau - d$, is at distance $d+l$ from n and from each child n_i of n if each character on the path from m to m_j does not match both n^c and n_i^c. It is because in order to first transform p_n to p_{m_j}, we need d edit operations to transform p_m to p_n and extra l insertion operations are required for the path characters. Second, to transform p_{n_i} to p_{m_j}, we need one less character insertions than in the previous case, and one more operation is required to substitute n_i^c with m_j^c. If m_j^c is the only path character that matches either n^c or n_i^c, then m_j is at distance $d+l-1$ from n or from n_i, respectively. Finally, If m^c matches n^c, the descendant m_j is at distance $d + l$ from n despite the relation between n^c and the path characters other than m^c, and will be at distance $d + l - 1$ from each child n_i of n if there exists a path character matching n_i^c; otherwise, if there is no path character matching n_i^c, m_j becomes at distance $d + l$ from n_i.

The generation method searches the sub-tries by taking the previous considerations into account while iterating through the depth value l; and when a descendant m_j of m becomes an active node to n, the sub-trie rooted at m_j needs to be further processed. Similar to ICAN, the method also keeps the minimum distance of each generated active node, that is, whenever we add a node m_j with distance d_1 to the active node set, if m_j is already there with distance d_2, we always keep the smaller distance.

Example Consider the trie in Figure 4, and let $\tau = 1$. Suppose n_1 is the currently processing node in `PreJoin`. $\mathcal{A}_{n_1}$ must be available. According to rules I and II, $\mathcal{A}_{n_1}$ is empty. Note that n_9 and n_{13} are at distance 1 but they are not included in $\mathcal{A}_{n_1}$. Thus, the subtries rooted at n_9 and n_{13} will be searched to compute $\mathcal{A}_{n_2}$ and $\mathcal{A}_{n_5}$. Considering n_9: n_9 is at distances 1 and 2 from n_5 and n_2, respectively, since n_9^c matches n_5^c and differs from n_2^c. Since $\tau = 1$, n_9 is included in $\mathcal{A}_{n_5}$ but not in $\mathcal{A}_{n_2}$. Considering descendants of n_9: n_{10} is at distance 2 from n_5 and n_2, since n_{10}^c differs from both n_2^c and n_5^c. Since $\tau = 1$, n_{10} is not included in $\mathcal{A}_{n_2}$ and $\mathcal{A}_{n_5}$. Since n_{10}^c matches n_1^c, n_{10} is an active node of n_1 with distance 1. Then the subtrie rooted at n_{10} must be processed. n_{11} is at distance 1 and 2 from n_2 and n_5, respectively, since n_{11}^c matches n_2^c and differs from n_5^c. Since $\tau = 1$, n_{11} is included in $\mathcal{A}_{n_2}$ but not in $\mathcal{A}_{n_5}$. Similarly the subtrie rooted at n_{13} can be searched.

4 SCALING `PREJOIN`

To improve the performance of `PreJoin`, it is scaled in two different ways. First, a dynamic re-ordering of the trie index is used to accelerate the search for similar string pairs. Second, a partitioning method of string space is used to improve performance on large edit distance thresholds.

4.1 Dynamic re-ordering

In this subsection we discuss the various ways to dynamically choose nodes to visit in the preorder traversal of the trie index in order to boost the search for similar string pairs. Our approach is not to follow the rigid structure of the trie, which is susceptible to the order of strings in the dataset. Alternatively, in our approach, at each traversing trie node, the next node to be visited, or subtrie to be processed, is determined according to one of the following three ordering strategies. Please note that this ordering is virtual, that is, it does not affect the original structure of the trie.

The *first ordering method* is based on the size of the subtrie rooted at each child of an already traversing node, where subtrie size is taken as the number of strings that belong to it. Here, we explore subtries in the ascending order of their size. To do so, we maintain a variable at each trie node to count how many data strings go through it, that is, the number of strings that this node is a prefix to. The *second ordering method* is based on the fan-out value of each child of a traversing node. We explore subtries in the ascending order of this value, that is, a child with the smallest fan-out will be visited first. The *third ordering method* is based on the depth of the subtrie rooted at each child of a traversing node, where subtrie depth is the length of the maximum string in the subtrie. Here, we explore subtries in the ascending order of their depth. To do so, we maintain a variable at each trie node to hold the length of the longest string passing through this node. We have applied the above ordering strategies in our approach, and carried out many experiments to asses the effect of each ordering method on the overall performance of `PreJoin`. Experiments carried out in Section 5 show that the third ordering method is more suitable than others.

4.2 Partitioning string space

When the edit distance threshold τ gets large, it becomes expensive for `PreJoin` to get the similar string pairs, since the size of the active-node set of each trie node increases. To address this problem, we improve `PreJoin`, and call the new version `PreJoin-Plus`. `PreJoin-Plus` is based on an idea from [3]. Consider a string $r = r_1 r_2 \ldots r_{|r|}$. We use $L(r) = r_1 r_2 \ldots r_{\frac{|r|}{2}}$ to denote the left half of r and $R(r) = r_{\frac{|r|}{2}+1} \ldots r_{|r|}$ to denote the right half of r. It is observed that if a string r is similar to a string s within τ, then at least one of the following conditions holds: (1) $L(r)$ is similar to a prefix of s within $\lfloor \frac{\tau}{2} \rfloor$; (2) $R(r)$ is similar to a suffix of s within $\lfloor \frac{\tau}{2} \rfloor$. For example, consider a string r = "avataresha", its left half $L(r)$ = "avata" and its right half $R(r)$ = "resha". Given a string s = "vatarts", as r is similar to s within τ=4, we can see the first condition holds, that is, $L(r)$ is similar to the prefix "vat" of s within $\lfloor \frac{4}{2} \rfloor = 2$.

`PreJoin-Plus` is illustrated as follows. Given a string collection S and edit threshold τ, it first constructs a new string set $L(S)$ that consists of the left half of each string in S. Then, we run `PreJoin` on $L(S)$ and S with edit distance threshold $\lfloor \frac{\tau}{2} \rfloor$. For a string $L(r)$ in $L(S)$, to find all the strings in S whose prefix is similar to $L(r)$, we traverse the descendants of each active node of node $L(r)$ and find the leaf nodes in S. Clearly, these leaf nodes have a prefix that is similar to $L(r)$. Similarly, if we reverse the strings in S, we can get all the string pairs $\langle r, s \rangle \in S \times S$ such that the right half $R(r)$ is similar to a suffix of s within $\lfloor \frac{\tau}{2} \rfloor$. We verify the candidate pairs generated from the two cases and obtain final results.

5 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of `PreJoin` on real data sets. `PreJoin` is implemented in standard C++ with STL library support and compiled with GNU GCC. Experiments were run on a PC with Intel(R) Core(TM) 2 Duo 2.66GHz CPU and 4GB memory running Linux.

Datasets: Three real datasets are used in experiments. (1) DBLP Author[4]. Only author names are extracted from DBLP dataset. (2) DBLP Author+Title. Each string is a concatenation of author names and title of a publication. (3) AOL Query Log[5]. Table 1 gives the detailed information of each data set and its abbreviated name that will be used afterwards. It shows the average, max and min lengths of strings in the data sets. DBLP Author is an example of data set with short strings, DBLP Author+Title is a data set with long strings, and the Query Log is a set of query logs. Note that these datasets are the same as those used in [15].

Table 1. Datasets statistics

Data sets	$avg-len$	$max-len$	$min-len$	$\|\sum\|$
DBLP Author (author)	12.82	46	4	37
AOL Query Log (query)	20.94	500	1	37
DBLP Authors+title (dblp)	104.78	1.743	10	37

5.1 Evaluation of dynamic re-ordering

In this set of experiments we evaluate the dynamic re-ordering technique associated with `PreJoin` algorithm. Figure 6 shows the comparison of running time between `PreJoin` with ordering [6] and `PreJoin` without ordering on different datasets and at edit distance threshold $\tau = 3$.

These experiments were performed on data subsets of size up to 200k strings. Figure 6 reveals that `PreJoin` with ordering beats `PreJoin` without ordering. For instance, at dataset size 200k, on author dataset, Figure 6(a) shows that `PreJoin` without ordering takes 70.707 seconds, while after ordering it takes 60.259 seconds. On query dataset, Figure 6(b) shows that `PreJoin` without ordering spends 98.817 seconds, while after ordering it spends 80.651 seconds. Also on dblp dataset, Figure 6(c) shows that `PreJoin` without ordering spends 47.556 seconds, while after ordering finishes within 35.876 seconds. We conclude that for small-size data subsets, reordering can save up to 25% of time.

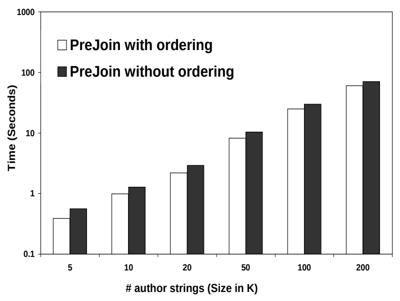
5.2 Comparison with trie-based join algorithms

Here, we compare `PreJoin` algorithm with Trie-Travese algorithm and the state-of-the-art algorithm Trie-PathStack at different τ. The executables for Trie-Travese and Trie-PathStack were obtained from their author [15].

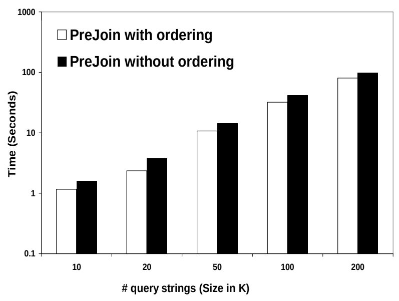
[4] http://www.informatik.unitrier.de/ ley/db

[5] http://www.gregsadetsky.com/aol-data/

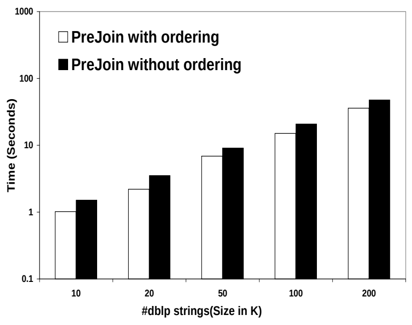
[6] We use the third ordering method (see Section 4.2) in these experiments since it shows the best performance among other ordering methods.



(a) author dataset



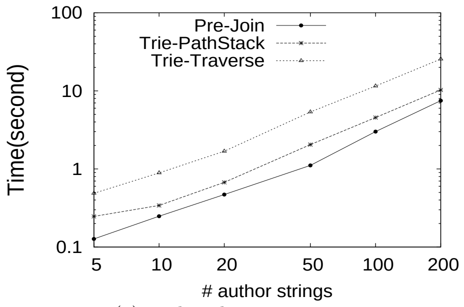
(b) query dataset



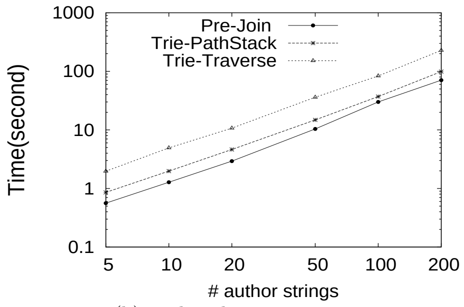
(c) dblp dataset

Fig. 6. Comparison of running time of `PreJoin` with and without ordering at $\tau = 3$.

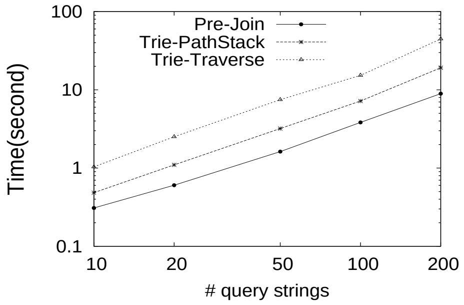
Figure 7 shows the results for the three datasets at different τ = 2-3, respectively. Note that we have two sub-figures for each dataset arranged based on τ. Each sub-figure plots the performance results with fixed τ and different subsets of the original dataset. Different subsets are used to show the scalability on the dataset size, whereas a sub-figure is used for each τ to show the scalability when τ increases. On the author dataset, a dataset characterized by short strings, Figure 7 shows that `PreJoin` performs the best. It outperforms Trie-PathStack, and the performance gap increases with larger τ and larger subsets. The performance gap between `PreJoin` and Trie-Traverse is relatively large, especially at $\tau \geq 2$. On the dblp dataset, a dataset characterized by long strings, `PreJoin` significantly outperforms Trie-Traverse by more than one order of magnitude, especially at $\tau \geq 2$. Also, the performance gap between `PreJoin` and Trie-PathStack is monotonically increasing with both τ and dataset size; it outperforms Trie-PathStack by factor five when τ=3 and the subset size is large.



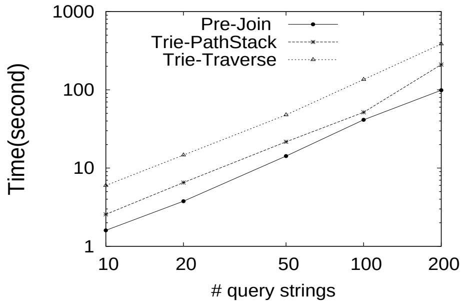
(a) author dataset: $\tau = 2$



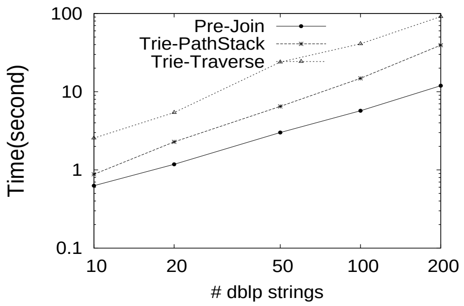
(b) author dataset: $\tau = 3$



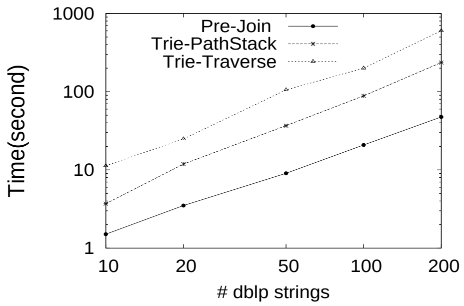
(c) query dataset: $\tau = 2$



(d) query dataset: $\tau = 3$



(e) dblp dataset: $\tau = 2$

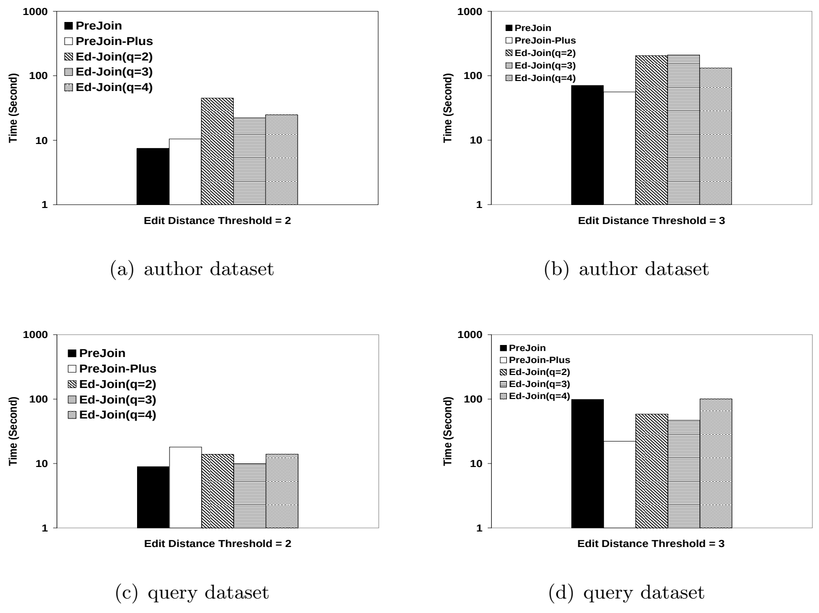


(f) dblp dataset: $\tau = 3$

Fig. 7. Comparative Performance: PreJoin, Trie-PathStack and Trie-Traverse on different dataset sizes (#strings in K).

5.3 Comparison with q-gram based algorithms

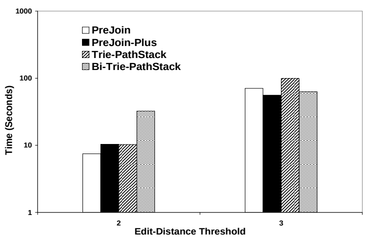
In this set of experiments, we compare `PreJoin` and `PreJoin-Plus` against the state-of-the-art q-gram based algorithm *Ed-Join* [8] on author and query datasets at different τ. Ed-Join generates $(|s| \cdot q + 1)$ q-grams for each string s and selects the first $(q \cdot \tau + 1)$ grams as gram prefix according to a predefined ordering on all grams. Those string pairs that do not share any gram in the constituted prefix length will be filtered and the survived string pairs will be verified by the edit distance computation. Ed-Join also uses *Location*-based and *content*-based mismatching q-gram for efficient filtering. Location-based filtering decreases the number of grams required in the prefix of each string, and content-based filtering reduces the amount of edit distance computation. As the performance of gram-based algorithms is highly dependent on the q parameter, we ran Ed-Join with different values of q. Figure 8 depicts the results.



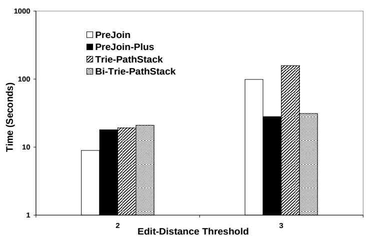
(a) author dataset
(b) author dataset
(c) query dataset
(d) query dataset

Fig. 8. Comparison of running time with q-gram based algorithms at different τ.

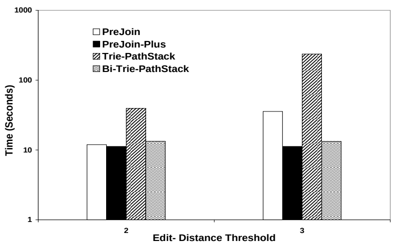
Figure 8 shows that `PreJoin` outperforms Ed-Join algorithm on author dataset. It is about 6 times faster than Ed-Join($q = 2$) at $\tau = 2$ (Figure 8 (a)), and is about 3 times faster than Ed-Join($q = 3$) at $\tau = 3$ (Figure 8 (b)). On query dataset, `PreJoin` also outperforms Ed-Join algorithm at $\tau = 2$ (Figure 8 (c)). At $\tau = 3$, (Figure 8 (d)), we find that Ed-Join is better than `PreJoin`. `PreJoin-Plus`, on the other hand, still outperforms Ed-Join for this threshold $\tau = 3$. `PreJoin-Plus` takes 22.106 seconds while Ed-Join($q = 4$) spends 100.556 seconds. In summary, we can say that our algorithms are always better than Ed-Join algorithm on author and query datasets at different τ.



(a) author dataset



(b) query dataset



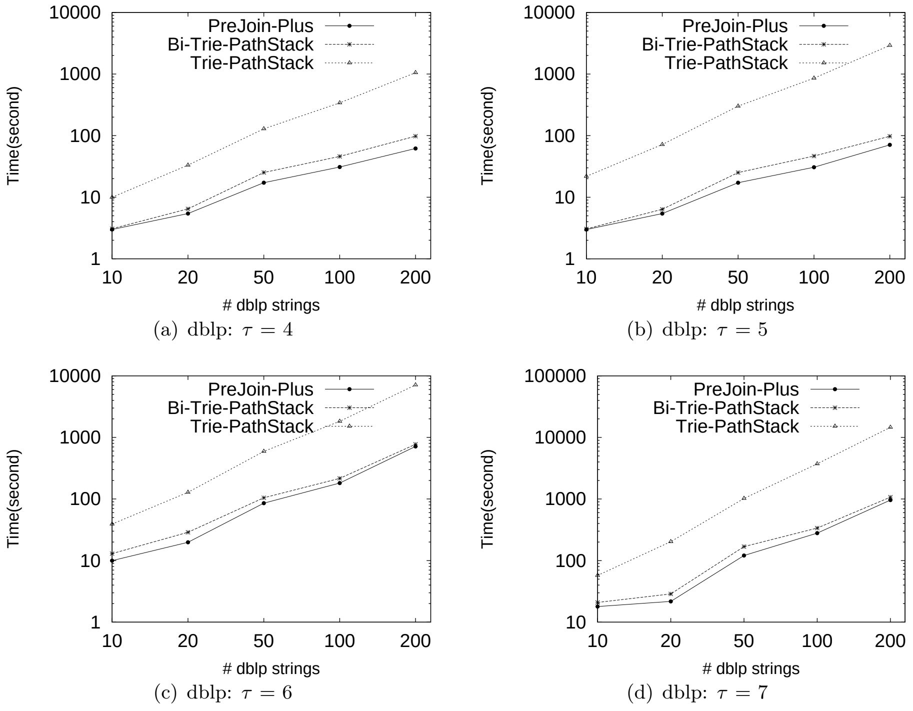
(c) dblp dataset

Fig. 9. Comparative Performance: `PreJoin`, `PreJoin-Plus`, Trie-PathStack and Bi-Trie-PathStack at $\tau = 2 - -3$.

5.4 Evaluation of `PreJoin-Plus` algorithm

In these experiments, we compare the efficiency of `PreJoin` and `PreJoin-Plus` with Trie-PathStack and Bi-Trie-PathStack. We ran all algorithms on author, query and dblp datasets by varying τ. Figure 9 shows the results at τ = 2-3. We can see that on author dataset, Figure 9 (a), `PreJoin` is faster than `PreJoin-Plus` at τ = 2. For instance, it takes 7.4 seconds while `PreJoin-Plus` spends 10.36 seconds. It also performs better than Trie-PathStack, and is about 4 times faster than Bi-Trie-PathStack. At $\tau = 3$, `PreJoin-Plus` outperforms `PreJoin`, Trie-PathStack and Bi-Trie-PathStack. For example, Figure 9 (b) shows that `PreJoin-Plus` is about 3 times faster than `PreJoin` on query dataset. For instance, it takes 28.106 seconds while `PreJoin` takes 98.817 seconds; it also performs better than Bi-Trie-PathStack and is about 6 times faster than Trie-PathStack. Figure 9 (c) shows the efficiency of `PreJoin-Plus` on dblp dataset. It shows the performance at τ = 3. In this subfigure, `PreJoin-Plus` is about 3 times faster than `PreJoin`, and is also better than Bi-Tri-PathStack. It is about 21 times faster than Trie-PathStack.

When τ gets larger, e.g., $\tau > 3$, `PreJoin-Plus` increases efficiency compared to Trie-PathStack and Bi-Trie-PathStack. We compared the running time of the three algorithms on dblp dataset, by increasing τ = 4-7, respectively. Figure 10 (a) shows that `PreJoin-Plus` is better than Bi-Trie-PathStack, and is about 17 times faster than Trie-PathStack at $\tau = 4$. Figure 10 (d) also shows that `PreJoin-Plus` is more efficient. It takes 959.645 seconds while Bi-Trie-PathStack spends 1070 seconds at 200K strings, when $\tau = 7$. `PreJoin-Plus` is also about 15 times faster than Trie-PathStack. In summary, `PreJoin-Plus` is always better than Bi-Trie-PathStack and Trie-PathStack when edit distance becomes large on datasets with long strings.



(a) dblp: $\tau = 4$ (b) dblp: $\tau = 5$

(c) dblp: $\tau = 6$ (d) dblp: $\tau = 7$

Fig. 10. Comparative Performance: `PreJoin-Plus`, Trie-PathStack and Bi-Trie-PathStack on dblp datasets for τ = 4-7(# String in K).

6 CONCLUSION

In this paper, we studied the problem of trie-based string similarity join with edit distance constraints. We proposed a new trie-based join algorithm called `PreJoin`, which improves upon current trie-based join methods. It efficiently finds all similar string pairs using a new active-node set generation method. To support large edit distance thresholds, `PreJoin` algorithm is improved by dynamically reordering the search space and partitioning of the string space. Experiments show that our approach outperforms state-of-the-art methods on datasets with short as well as long strings, even with large edit distance thresholds.

REFERENCES

[1] Arasu, A.— Ganti, V.— Kaushik, R.: Efficient exact set similarity joins. In: VLDB, 2006, pp. 918–929.

[2] Bayardo, R.— Ma, Y.— Srikant, R.: Scaling up all pairs similarity search. In: WWW, 2007, pp. 131–140.

[3] Chaudhuri, S.— Ganti, V.— Kaushik, R.: A primitive operator for similarity joins in data cleaning. In: ICDE, 2006, pp. 5–16.

[4] Dong, X.— Halevy, Y.A.— Yu, C.: Data integration with uncertainty. In: VLDB, 2007, pp. 687–698.

[5] Feng, J.— Li, G.— Hao, S.— Wang, J.— Feng, J.— Li, W.S.: Massjoin: A mapreduce-based method for scalable string similarity joins. In: ICDE, 2014, pp. 340–351.

[6] Feng, J.— Wang, J.— Li, G.: Trie-join: a trie-based method for efficient string similarity joins. VLDB J., Vol. 21, 2012, No. 4, pp. 437–461.

[7] Gouda, K.— Rashad, M.: Prejoin: An efficient trie-based string similarity join algorithm. In: INFOS (2012).

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

[9] Hernandez, M.— Stolfo, S.: Real-world data is dirty: data cleansing and the merge/purge problem. Data Mining and Knowledge Discovery, Vol. 4, 1998, No. 1, pp. 9–37.

[10] Ji, S., Li, G., Li, C., Feng, J.: Efficient interactive fuzzy keyword search. In: WWW, 2009, pp. 433–439.

[11] Jordan, M.— Kleinberg, J.— Scholkopf, B.: Pattern recognition and machine learning (information science and statistics). In Springer-Verlag New York, Inc. (2006)

[12] Li, G.— Deng, D.— Feng, J.: A partition-based method for string similarity joins with edit-distance constraints. ACM Trans. Database Syst. Vol. 38, 2013, No. 2, pp. 437–461.

[13] Ukkonen, E.: Algorithms for approximate string matching. Information and Control, Vol. 64, 1985, No. 1-3, pp. 100–118.

[14] Wagner, R.— Fischer, M.: The string-to-string correction problem. J. ACM, Vol. 21, 1974, No. 1, pp. 168–173.

[15] Wang, J.— Li, G.— Feng, J.: Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. PVLDB, Vol. 3, 2010, No. 1, pp. 1219–1230.

[16] Winkler, W.: The state of record linkage and current research problems. Technical report, Statistical Research Division, U.S. Census Bureau

[17] Xiao, C.— Qin, J.— Wang, W.— Ishikawa, Y.— Tsuda, K.— Sadakane, K.: Efficient error-tolerant query autocopletion. PVLDB, Vol. 6, 2013, No. 6, pp. 373–384.

[18] Xiao, C.— Wang, W.— Lin, X.: Ed-join: an efficient algorithm for similarity joins with edit distance constraints. In: VLDB, 2008, pp. 933–944.