PreJoin: An Efficient Trie-based String Similarity Join Algorithm

Karam Gouda
Faculty of Computers & Informatics
Benha University, Benha, Egypt
karam.gouda@fci.bu.edu.eg

Metwally Rashad
Faculty of Computers & Informatics
Benha University, Benha, Egypt
metwally.rashad@fci.bu.edu.eg

Abstract

A string similarity join finds all similar pairs between two collections of strings. It is an essential operation in many applications, such as data integration and cleaning, and has attracted significant attention recently. In this paper, we study string similarity joins with edit-distance constraints. Recently, a Trie-based similarity Join framework is proposed [3]. Existing Trie-based Join algorithms have shown that Trie Indexing is more suitable for Similarity Join on short strings. The main problem with current approaches is that they generate and maintain lots of candidate prefixes called active nodes which need to be further removed. 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 over current Trie-based Join methods. It efficiently finds all similar string pairs using a new active-node set generation method, and a dynamic preorder traversal of the Trie index. Experiments show that PreJoin 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.

1. Introduction

String data is ubiquitous, and its management has taken on particular importance in the past few years. $Similarity\ join$ has become an essential operation in many applications, such as data integration and cleaning, web page detection, and pattern recognition. Similarity join is also adopted in the industry solutions. A string similarity join between two sets of strings finds all similar string pairs from the two sets. For example, consider two sets of strings {stick, mode, ...} and {stich, make, ...}. We want to find all similar pairs, e.g., (stick, stich) and (mode, make). Many similarity functions have been proposed to quantify the similarity between two strings, such as Jaccard similarity, Cosine similarity, and edit distance. In this paper, we study string similarity joins with edit-distance constraints [5]. Edit distance measures the minimum number of edit operations (insertion, deletion, and substitution) to transform one string to another. Edit distance has two distinctive advantages over alternative distance or similarity measure: (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.

Previous algorithms for Similarity joins such as Part-Enum [1], All-Pairs-Ed [4], Ed-Join [2], usually employ a filter-and-refine framework. In the $filter$ step, they generate Q-gram signatures for each string and use the signatures to generate candidate pairs. In the $refine$ step, they verify the candidate pairs and output the final results. These approaches have the following disadvantages. $Firstly$, they are inefficient for the data sets with short strings (the average string length is no larger than 30), since they cannot select high-quality signatures for short strings, and thus they may generate a large number of candidate pairs which need to be further verified. $Secondly$, they cannot support dynamic update of data sets. The dynamic update may change the weights of signatures. Thus 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 similarity Join framework is proposed [3]. In comparison with the filter-and-refine framework, Trie-based Join can efficiently generate all similar string pairs without the refine step. As many common prefixes of strings are shared by the trie structure, the index size is minimized, and prefix pruning is used to improve performance. Nevertheless, the main problem of current Trie-based Join methods is that: At each prefix (trie node), they generate and maintain lots of candidate prefixes called active nodes which need to be further removed. With large

edit distance, the number of active nodes becomes quite large. To overcome this problem, they use many pruning techniques[1] to remove false positive prefixes in a subsequent phase, consequently more computation overhead is added. This make the existing approaches inefficient for processing large data sets with long strings, and higher edit distance threshold.

In this paper, we propose a new Trie-based similarity Join approach called PreJoin, which minimizes false candidate prefixes. PreJoin uses preorder traversal combined with an efficient way to generate the actual active nodes. The new generation method encapsulates the previous pruning techniques. Thus, the overhead of applying these pruning techniques is removed. Moreover, the tree traversal in PreJoin is dynamic, that is, at each node, the next subtrie to be processed is determined according to an effective ordering methodology. Extensive 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.

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 PreJoin algorithm. Experimental results are given in Section 4. Finally, Section 5 concludes the paper.

2. Preliminaries

2.1. Problem Statement

Let Σ be a finite alphabet of symbols σ_i ($1 \leq i \leq |\Sigma|$). A string s is an ordered array of symbols drawn from Σ. We use $|s|$ to denote the length of string s, and "$s[i]$" to denote the $i\text{-}th$ character of s, and "$s[1..j]$" to denote the prefix of string from starting of string to its j-th character. Each string s is also assigned an identifier $s.id$. 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 (i.e., insertion, deletion, and substitution) needed to transform s_1 to s_2. For example, the edit distance between s_1 = $Jim\ Gray$ and s_2 = $Jim\ Grey$ is 1, since s_1 is transformed to s_2 with a substitute operation that replace the letter in position $s_1[7]$ with the new letter e.

Given two sets of strings $\mathcal{R}$ and $\mathcal{S}$, a **similarity join with edit distance threshold** τ (or edit similarity join [5]) returns pairs of strings from each set, such that their edit distance is no larger than τ , i.e., $\{< r, s >: ed(r, s) \leq \tau, r \in \mathcal{R}, s \in \mathcal{S}\}$. For example, consider the strings $s_1 = kaushik\ chakrab$, $s_2 = caushik\ chakrabar$. Suppose threshold τ = 3, $< s_1, s_2 >$ is a similar pair as their edit distance is not larger than τ. In this paper, for the ease of exposition, we will focus on the self-join case, i.e., $\mathcal{S} = \mathcal{R}$.

[1] Pruning technique such as Length Pruning, Single-branch Pruning and count Pruning

Several algorithms have been proposed in the previous studies to solve the similarity join problem such as Part-Enum [1], All-Pairs-Ed [4], ED-Join [2] and Trie-PathStack [3]. Trie-PathStack and our approach PreJoin are Trie-based methods, whereas other methods follow the filter and refine framework. Next, we introduce the working principle of the Trie-based string similarity Join.

2.2. Trie-based String Similarity Join

In this paper, a trie is used to index all strings in the data set $\mathcal{R}$. Trie is a tree structure where each path from a root to a leaf represents a string in $\mathcal{R}$, and every node on the path has a label of a corresponding character in the string. For instance, Figure 1 shows a trie structure of a sample data set. String "ebay" has a trie node id of 12, and its prefix "eb" has a trie node id of 10. Given a trie node n, let $|n|$ denote its depth, e.g., $|"ko"| = 2$ (the depth of the root node is 0).

Note that many strings with the same prefix share the same ancestor nodes on the trie. Thus, if two 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 [3]. It works as follow. Given a trie node n and an edit distance τ, another trie node m is called an *active node* for n if $ed(p_n, p_m) \leq \tau$, where p_i is the corresponding prefix of node i. Thus, if a node m is not an active node of a node n, then m's descendants will not be similar to n's descendants. For example, consider the trie in Figure 1 and suppose $\tau = 1$. The set appears next to each node is its active-node set, i.e., the set of all its active nodes. Since "ko" is not an active node of "b", then all the strings with prefix "ko", i.e., strings with id s_5 and s_6 in the data set, are not similar to the strings with prefix "b", i.e., strings with id s_2, s_4 and s_7.

In [3], J. Feng, et al. proposed several algorithms to traverse the trie structure to find similar string pairs using dual subtrie pruning. They also introduced different pruning techniques to improve performance. Trie-based similarity join works as follows. For each encountered node n in the traversal, it computes its active-node set denoted $\mathcal{A}_n$. If n is a leaf node, i.e., it represents a string $s \in \mathcal{R}$, then for every leaf node $s_i \in \mathcal{A}_n$, $<s, s_i>$ is a similar string pair. Active-node sets are computed incrementally

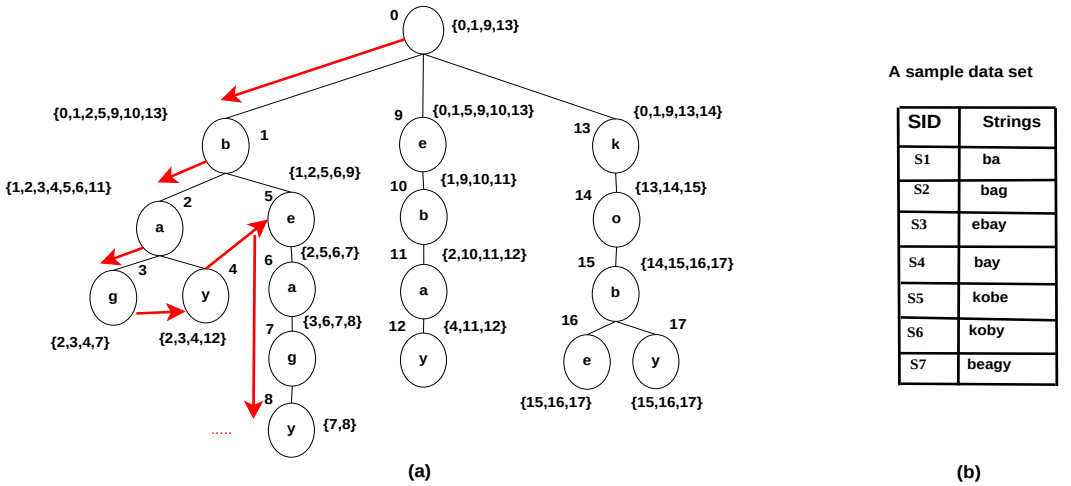


Figure 1. A sample data set and its trie-based Index.

using a method called ICAN [6] (Incrementally Computing Active Nodes) as follows. Initially, the trie root r represents an empty string ϵ, and its corresponding active-node set $\mathcal{A}_r$ includes all trie nodes m with depths no larger than τ. Suppose the active-node set $\mathcal{A}_n$ of a given node n is computed. The algorithm computes the active-node set of each n's child from the active-node set $\mathcal{A}_n$. The time complexity of computing $\mathcal{A}_c$ from its parent's active-node set $\mathcal{A}_p$ is $O(\tau.|\mathcal{A}_c|)$, since each active node only can be computed from its ancestors within τ steps.

Trie-Traverse is a preorder traversal method introduced in [3] 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 compute active-node set $\mathcal{A}_c$ of a node c based on its parent's active-node set $\mathcal{A}_p$. 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, hence, the time complexity of Trie-Traverse is given as $O(\tau.|\mathcal{A}_\mathcal{T}|)$, where $|\mathcal{A}_\mathcal{T}| = \sum_{c\in\mathcal{T}}|\mathcal{A}_c|$. Figure 1 shows the active-node sets computed in the preorder traversal. The arrows in the figure show the order of 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 sizes of active-node sets. Moreover, when $\mathcal{R}$ is large and consists of long strings, the corresponding trie becomes very large. The main objective of our research is to scale up Trie-based similarity Join on long strings. While a different traversal method called Trie-PathStack has been introduced in [3] to speed up Trie-based Join, here we present a new method called PreJoin, which uses the pre-

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

Input: $\mathcal{R}$: a collection of strings;
τ: edit-distance threshold.
Output: $P = \{(s \in \mathcal{R}, t \in \mathcal{R}) : ed(s,t) \leq \tau\}$.
1. $\mathcal{T}$ = new Trie($\mathcal{R}$);
2. Let r denote the root of Trie $\mathcal{T}$;
3. $\mathcal{A}_r$= $\{n$: for each trie node n, s.t., $|n| \leq \tau\}$;
4. **for** each child node of r, c **do**
5. $P \cup$= $findSimilarPair(c, r, \mathcal{A}_r)$;
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 $l \in \mathcal{A}_n (n \neq l)$ **do**
14. $P_n = \{(n, l)\}$;

Figure 2. Trie-Traverse algorithm

order traversal combined with a new active-node generation method to optimize the active-node sets' sizes.

3. PreJoin Algorithm

In Trie-based similarity Join approaches, the active node generation method ICAN has to generate active nodes in a separate phase (e.g., line 7 in Figure 2). Many active

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

Input: $\mathcal{R}$: a collection of strings;
τ: edit-distance threshold.
Output: $P = \{(s \in \mathcal{R}, t \in \mathcal{R}) : ed(s,t) \leq \tau\}$.

1. $\mathcal{T}$ = new Trie($\mathcal{R}$);
2. $Pre\_Traverse(root)$;
3.**Procedure** $Pre\_Traverse(t)$
4. impose an order on t children;
5. let $CN_t = \{n_1, \ldots, n_k\}$ denote children of t;
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** $Push\_down(n_i, n_j, 1, \tau, 0)$;

Figure 3. PreJoin Algorithm

nodes are false candidates and need to be removed in a subsequent pruning phase (e.g., line 8 in Figure 2). The computation overhead caused by these two phases is the main reason of why the current Trie-based similarity Join is not the best choice on long strings. Here, we devise a new generation method different from ICAN, which produces the actual active nodes, and therefore we do not need the pruning phase. This will scale Trie-based Join framework on long strings.

Combining the trie preorder traversal with the new generation method, a new algorithm called PreJoin is developed. Figure 3 outlines PreJoin Algorithm. It is illustrated as follows. Given a data set $\mathcal{R}$, each string $s \in \mathcal{R}$ is inserted into the trie according to the data set order. Recall that each trie leaf represents a string in the data set. Intermediate nodes may also represent data strings which are contained by other strings. These nodes are identified by the logical variable *EOS* (stand for End Of String). PreJoin visits nodes in preorder as Trie-Traverse. However, PreJoin differs from Trie-Traverse in that: First, in addition to constructing the active-node set for the next child to be visited as in Trie-Traverse, it also constructs the active-node sets for all its siblings. Thus, active-node set of each sibling will be available when reaching that sibling in the traversal. Second, PreJoin does not follow the order imposed by the trie structure during traversal, it instead re-orders the siblings virtually, to decide which subtrie to be traversed next. Finally, PreJoin employs a new active-node set generation method that first avoids adding false positive into the active-node sets, and generates active nodes by investigating relatively larger subtries rooted at parent's active nodes.

3.1. Novel Active Nodes Generation Method

The new generation method avoids adding false candidates by enforcing the following **rules** during active-nodes generation.

RULE I: The first rule is to apply *symmetry property* of edit distance early in the generation: Given 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 the trie node at level i that is currently processed in the traversal. There are two sub-cases: (1) Each n's ancestor node m at depth $j = i-1, i-2, \ldots, i-\tau$, is an active node of n within distance j, since j deletion operations are required to transform the corresponding string of n into the corresponding string of m. (2) Each n's descendant node m at depth $j = i+1, i+2, \ldots, i+\tau$, is an active node of n within distance j, since j insertion operations are required to transform the corresponding string of n into the corresponding string of m. Based on the symmetry property, our generation method does not include all ancestors and descendants active nodes m 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* and within distance τ from n, and then outputs the strings corresponding to n and m. As an example, according to our generation method, although the nodes n_1, n_2, n_3, n_4 in Figure 1 are actives to the node n_2 according to Trie-Traverse algorithm, they are not included into $\mathcal{A}_{n_2}$ by PreJoin (see 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.
Case 2: Also based on the symmetry property, our method does not allow any node already traversed to be an active node of n. For example, suppose that n_{13} of Figure 1 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 follow. Suppose n is the trie node currently under processing in the traversal. For each active node $m \in \mathcal{A}_n$ with distance d, *Out-Similar* searches in the hight $\tau - d$ subtrie rooted at m for any node of type EOS to be

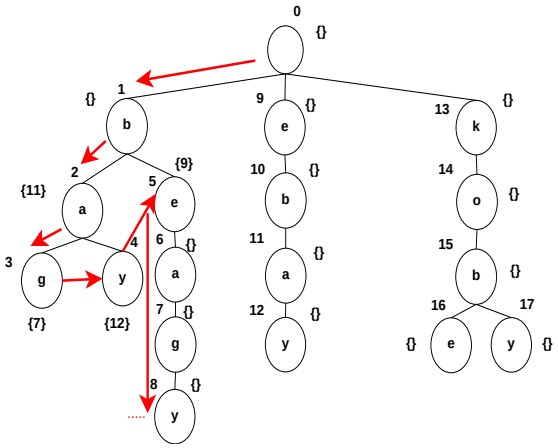


Figure 4. Preorder Traversal plus Active-node set generation in PreJoin ($\tau = 1$).

output. It also searches the hight $\tau - 1$ subtrie rooted at each n's sibling for *EOS* nodes to output. For example, let node n_2 in Figure 4 be under processing. Since it is of type EOS, the subtries rooted at n_2 and n_{11} are processed by *Out-Similar*.

RULE II: The second rule taken by the generation method is that, while the method generates active-node set $\mathcal{A}_c$ from its parent's active-node set, it does not insert the remaining siblings as active nodes, though those siblings are active nodes to each other because any two siblings are within edit distance one. Note that one substitution operation is required to transform the corresponding string of one sibling into the corresponding string of another. Nevertheless, latter on when a sibling becomes the current node to be processed, the method considers the unprocessed 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 latter 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 within edit distance one.

Figure 4 shows the active-node sets generated by PreJoin. Comparing these sets against the ones generated by Trie-Traverse, we find that the active-node sets produced by PreJoin are minimized.

Below, we show how our generation method computes an active-node set of a given node by deeply investigating a subtrie rooted at each active node of its parent. The involved deep search guarantees that larger subtries are considered relative to ICAN method. Moreover, since PreJoin requires 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, saving most of the duplicated computations. Hence, PreJoin is capable of dealing with long strings efficiently. Note also that the pruning rules considered will make PreJoin able to cope with larger τ. Before proceeding, let n^c denote the character at node n.

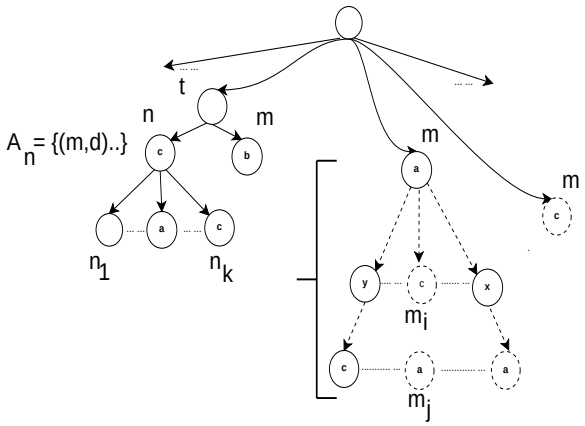


Figure 5. Computing Active-node sets of n children.

The active nodes computation is outlined in PreJoin under the function *Gen_ActiveNode* (Line 11-15, Figure 3). It works as follows. Suppose n is the currently visited node in PreJoin. PreJoin supposes that $\mathcal{A}_n$ is already available. For each active node m in $\mathcal{A}_n$ at distance $d \leq \tau$, only the subtrie rooted at m is examined. Note also that, the unprocessed siblings are active nodes at distance one, but they are not included in $\mathcal{A}_n$ according to the above second rule. The trie level at which the search can reach in a subtrie depends on n^c and m^c, as illustrated in Figure 5. 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 node m, and (2) m^c matches n^c. Here, we can reach up to the level $l_m + (\tau - d) + 1$. Next, we show how to search the subtrie.

Considering m: m will be an active node of each child n_i of n at distance $d + 1$, if m^c differs from n_i^c, since m could be transformed to n with d edit operations and then one deleting operation is required for n_i^c. Otherwise, if m^c matches n_i^c, m is an active node of n_i at distance d, since the previous substitution operation between m^c and n^c is replaced by an insertion operation of n^c.

Considering m's descendants: Each descendant m_i of

m at level $l_m + l$, $l \leq \tau - d$ is an active node of each child n_i of n with distance $d + l$, if each node character on the path from m to m_i does not match both n^c and n_i^c, since m could be transformed to n with d edit operations, and $l - 1$ insertion operations are required for the path characters, and one substitution operation is required for n_i^c and m_i^c. Otherwise, if m_i^c is the only path character that matches n_i^c, then m_i and n_i are within distance $k+l-1$.

If m_i^c match n^c, then m_i is an active node of n with distance $d + l - 1$. Otherwise, if m_i^c does not match n^c, then m_i is an active node of n with distance $d + l$, if each node character on the path from m to m_i does not match n^c. In these two cases, the subtrie rooted at m_i needs to be further processed. When n and m are at distance τ. First, m_i is active node of n_i with distance τ, if m_i^c matches n_i^c. Second, m_i is active node of n with distance τ, if n^c matches m_i^c, thus each child of m_i is at distance τ with the child n_i having the same character.

Keeping minimum distances: During the computation of the active-node set $\mathcal{A}_{n_i}$, whenever we add a node m_i with distance d_1 to the set, if m_i is already there with distance d_2, we always keep the smaller distance.

Example 3.1 *Consider the trie in Figure 4, and let $\tau = 1$. Suppose n_1 is the currently processed 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. 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 4G memory running Linux.

Datasets: Three real datasets are used in experiments:

Data sets	*avg-len*	*max-len*	*min-len*	$\lvert\Sigma\rvert$
DBLP Author	12,82	46	4	37
AOL Query Log	20,94	500	1	37
DBLP Authors+title	104,78	1,743	10	37

Table 1. Data set statistics.

DBLP Author[2], DBLP Author+Title, and AOL Query Log[3]. Table 1 illustrates statistical detailed information of each data set. It shows the average, max and min lengths of strings in the data sets. DBLP Author is a 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 that used in [3].

4.1. Comparison with Trie-based Join algorithms

Here, we compare our algorithm PreJoin against Trie-Travese algorithm and the state-of-the-art algorithm Trie-PathStack for different τ. The executables for Trie-Travese and Trie-PathStack were obtained from their author [3]. Figure 6 shows the result for the datasets on different $\tau = 1 - 3$. Note that we have three sub-figures for each dataset. Each sub-figure plots the performance result 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 with short strings, Figure 6 shows that PreJoin perform the best. It outperformed Trie-Pathstack, and the performance gap increases with larger τ and large subsets. The performance gap between PreJoin and Trie-Traverse is relatively large, especially for $\tau > 1$.

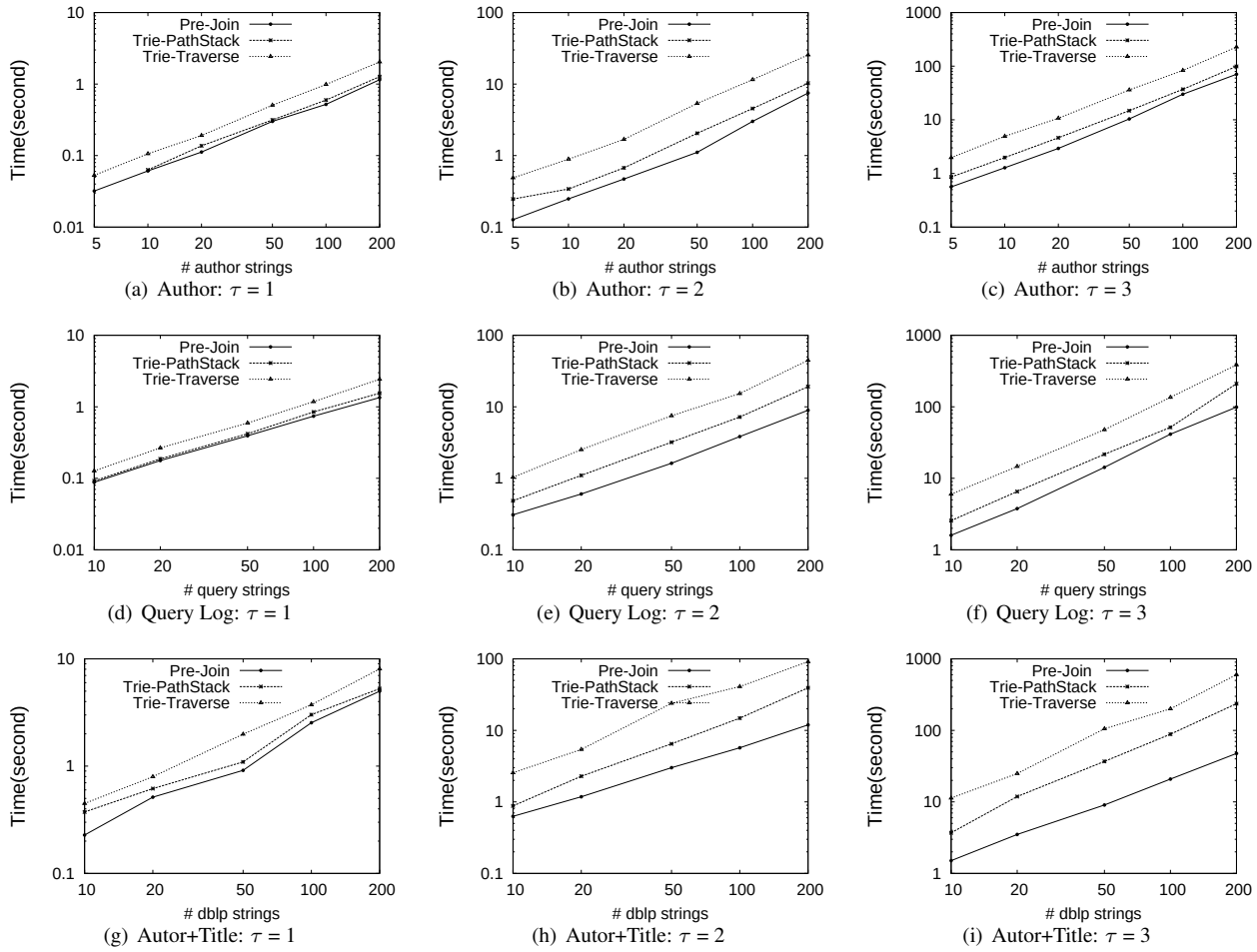
On the Author+Title dataset with long strings, PreJoin significantly outperformed Trie-Traverse by more than one order of magnitude, especially for $\tau > 1$. Also the performance gap between PreJoin and Trie-PathStack increases more than before; it outperforms Trie-PathStack by factor five when $\tau = 3$ and the subset is large.

5. Conclusion

In this paper, we have studied the problem of Trie-based string similarity Joins with edit-distance constraints. We proposed a new Trie-based Join algorithm called PreJoin,

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

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



(a) Author: $\tau = 1$ (b) Author: $\tau = 2$ (c) Author: $\tau = 3$

(d) Query Log: $\tau = 1$ (e) Query Log: $\tau = 2$ (f) Query Log: $\tau = 3$

(g) Autor+Title: $\tau = 1$ (h) Autor+Title: $\tau = 2$ (i) Autor+Title: $\tau = 3$

Figure 6. Comparative Performance: PreJoin, Trie-PathStack and Trie-Traverse on different dataset sizes (#strings in K).

which improves over current Trie-based Join methods. It efficiently finds all similar string pairs using a new active-node set generation method, and a dynamic preorder traversal of the Trie index. Experiments show that PreJoin scales the Trie-based Join to be used on datasets with long as well as short strings, even with large edit distance threshold.

References

[1] A.Arasu, V.Ganti, and R.Kaushik. Efficient exact set-similarity joins. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 918–929. VLDB Endowment, 2006.

[2] C.Xiao, W.Wang, and X.Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proceedings of the 34nd international conference on Very large data bases*. VLDB '08', pages 933–944. VLDB Endowment, 2008.

[3] J.Wang, J.Feng, and G.Li. Trie-join : Efficient trie-based string similarity joins with edit-distance constraints. In *Proceedings of the 36nd international conference on Very large data bases*, VLDB '10, pages 1219–1230. VLDB Endowment, 2010.

[4] R.J.Bayardo, Y.Ma, and R.Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 131–140. ACM, 2007.

[5] S.Chaudhuri, V.Ganti, and R.Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06. IEEE Computer Society, 2006.

[6] S.Ji, G.Li, C.Li, and J.Feng. Efficient interactive fuzzy keyword search. In *Proceedings of the 18th international conference on World Wide Web*, WWW '09, pages 371–380. ACM, 2009.