Trie-Join: Efficient Trie-based String Similarity Joins with Edit-Distance Constraints

Jiannan Wang, Jianhua Feng, Guoliang Li

Department of Computer Science, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 10084, China

wjn08@mails.thu.edu.cn; fengjh@tsinghua.edu.cn; liguoliang@tsinghua.edu.cn

ABSTRACT

A string similarity join finds 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. Existing methods usually employ a filter-and-refine framework and have the following disadvantages: (1) They are inefficient for the data sets with short strings (the average string length is no larger than 30); (2) They involve large indexes; (3) They are expensive to support dynamic update of data sets. To address these problems, we propose a novel framework called trie-join, which can generate results efficiently with small indexes. We use a trie structure to index the strings and utilize the trie structure to efficiently find the similar string pairs based on subtrie pruning. We devise efficient trie-join algorithms and pruning techniques to achieve high performance. Our method can be easily extended to support dynamic update of data sets efficiently. Experimental results show that our algorithms outperform state-of-the-art methods by an order of magnitude on three real data sets with short strings.

1. INTRODUCTION

The similarity join is an essential operation in many applications, such as data integration and cleaning, near duplicate object detection and elimination, and collaborative filtering. Recently it has attracted significant attention in both academic and industrial community. For example, SSJoin [3] proposed by Microsoft has been used in the data debugger project. A similarity join between two sets of objects finds all similar object pairs from each set. For example, given two sets of strings \( \mathcal{R} = \{ \text{kobe, ebay, . . . } \} \) and \( \mathcal{S} = \{ \text{bag, koby, . . . } \} \). We want to find all similar pairs \( \langle r, s \rangle \in \mathcal{R} \times \mathcal{S} \), such as \( \langle \text{kobe, koby} \rangle \).

Many similarity functions have been proposed to quantify the similarity between two objects, such as jaccard similarity, cosine similarity, and edit distance. In this paper, we study string similarity joins with edit-distance constraints, which, given two sets of strings, find all similar string pairs from each set, such that the edit distance between each string pair is within a given threshold. The string similarity join has many real applications, such as finding near duplicated queries in query log mining and correlating two sets of data (e.g., people name, place name, address).

Existing studies, such as Part-Enum [4], All-Pairs-Ed [4], Ed-Join [8], usually employ a filter-and-refine framework. In the filter step, they generate 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. However, 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. For example, Ed-Join and All-Pairs-Ed need to select signatures with higher weights. The dynamic update may change the weights of signatures. Thus the two methods need to reselect signatures, rebuild indexes, and rerun their algorithms from scratch. Thirdly, they involve large index sizes as there could be large numbers of signatures.

To address above-mentioned problems, in this paper we propose a new trie-based framework for efficient string similarity joins with edit-distance constraints. In comparison with the filter-and-refine framework, our approach can efficiently generate all similar string pairs without the refine step. We use a trie structure to index strings which needs much smaller space than existing methods, as the trie structure can share many common prefixes of strings. To avoid repeated computation, we propose subtrie pruning and dual subtrie pruning to improve performance. We devise efficient trie-join-based algorithms and three pruning techniques to achieve high performance. Our method can be easily extended to support dynamic update of data sets.

To summarize, in this paper, we make the following contributions: (1) We propose a trie-based framework for efficient string similarity joins with edit-distance constraints. (2) We devise efficient trie-join-based algorithms and develop pruning techniques to achieve high performance. (3) We extend our method to support dynamic update of data sets efficiently. (4) Experimental results show that our method achieves high performance and outperforms existing algorithms by an order of magnitude on data sets with short strings (the average string length is no larger than 30).
The rest of this paper is organized as follows. Section 2 introduces the trie-based framework. We propose efficient trie-join algorithms in Section 3 and develop three pruning techniques in Section 4. Section 1 presents an incremental algorithm for string similarity joins and Section 2 discuss how to extend our algorithm for two different string sets. Experimental results are provided in Section 5. We review related work in Section 6 and make a conclusion in Section 7.

2. TRIE-BASED FRAMEWORK

In this section, we first formalize the problem of string similarity joins with edit-distance constraints and then introduce a trie-based framework for efficient similarity joins.

2.1 Problem Formulation

Given two sets of strings, a similarity join finds all similar string pairs from the two sets. In this paper, we use edit distance to quantify the similarity between two strings. Formally, the edit distance between two strings r and s, denoted as ED(r, s), is the minimum number of single-character edit operations (i.e., insertion, deletion, and substitution) needed to transform r to s. For example, ED(koby, ebay) = 3. In this paper two strings are similar if their edit distance is not larger than a given edit-distance threshold τ. We formalize the problem of string similarity joins as follows.

Definition 1 (String Similarity Joins). Given two sets of strings R and S, and an edit-distance threshold τ, a similarity join finds all similar string pairs (r, s) ∈ R × S such that ED(r, s) ≤ τ.

2.2 Prefix Pruning

One naive solution to address this problem is all-pair verification, which enumerates all string pairs (r, s) ∈ R × S and computes their edit distances. However, this solution is rather expensive. In fact, in most cases to check whether two strings are similar, we need not compute the edit distance between the two complete strings. Instead we can do an early termination in the dynamic-programming computation as follows [12].

Given two strings r = r1r2 . . . rn and s = s1s2 . . . sm, let D denote a matrix with n + 1 rows and m + 1 columns, and D(i, j) be the edit distance between the prefix r1r2 . . . ri and the prefix s1s2 . . . sj. We use the dynamic-programming algorithm to compute the matrix: D(0, j) = j for 0 ≤ j ≤ n, and D(i, j) = min(D(i − 1, j) + 1, D(i, j − 1) + 1, D(i − 1, j − 1) + θ) where θ = 0 if ri = sj; otherwise θ = 1. D(i, j) is called an active entry if D(i, j) ≤ τ. Figure 1 shows the matrix to compute the edit distance between “ebay” and “koby”. The shaded cells (e.g., D(1, 1)) denote active entries for τ = 1. (For all running examples in the remainder of this paper, we assume τ = 1.) To check whether r = “ebay” and s = “koby” are similar, we first compute the entries in row D(0, s) (only those entries circled by the bold lines). As D(0, 0) and D(0, 1) are active entries, we compute the entries in row D(1, s). Similarly, we compute the entries in row D(2, s). We find that D(2, 1), D(2, 2) and D(2, 3) are not active entries. Based on the dynamic-programming algorithm, the following rows D(i > 2, s) cannot have active entries, thus we can do an early termination. This pruning technique is called prefix pruning. However the method using prefix pruning for similarity joins also needs to do all-pair verification. To improve prefix pruning and increase performance, we make the following two observations.

Figure 1: Prefix pruning. Matrix for computing edit distance of two strings “ebay” and “koby”. Shaded cells denote active entries for τ = 1.

2.3 Our Observations

Observation 1 - Subtrie Pruning. As there are a large number of strings in the two sets and many strings share prefixes, we can extend prefix pruning to prune a group of strings. We use a trie structure to index all strings. Trie is a tree structure where each path from the root to a leaf represents a string in the data set and every node on the path has a label of a character in the string. For instance, Figure 2 shows a trie structure of a sample data set with six strings. String “ebay” has a trie node ID of 12 and its prefix “eb” has a trie node ID of 10. For simplicity, a node is mentioned interchangeably with its corresponding string in later text. For example, both node “ko” and string “koby” refer to node 14, and node 14 also refers to string “koby”. Given a trie node n, let |n| denote its depth (the depth of the root node is 0). For example, |“ko”| = 2.

Figure 2: Trie index of a sample data set

Note that many strings with same prefixes share the same ancestor nodes on the trie structure. Based on this property, we can extend the idea of prefix pruning to prune a group of strings. Given a trie and a string s, node n in the trie is called an active node of string s if ED(s, n) ≤ τ. If n is not an active node for every prefix of string s, then all the strings under n cannot be similar to s. The reason is the following. For any string with prefix n in the trie, say r, in the dynamic-programming algorithm, we can take r as the row and s as the column. As the row D(|n|, s) has no active entry, r cannot be similar to s based on prefix pruning. Based on this observation, we propose a new pruning technique, called subtrie pruning: Given a trie and a string s, to compute the similar strings of s on the trie, for each trie node n, if n is not an active node of every prefix of s, we need not traverse the subtrie rooted at n. The following Lemma shows the correctness of the subtrie pruning.

Lemma 1 (Subtrie Pruning). Given a trie T and a string s, if node n is not an active node for every prefix of s, then n’s descendants will not be similar to s.
For example, consider the trie in Figure 2 and suppose \( \tau = 1 \). Given a string “ebay”, since node “ko” is not an active node for every prefix of “ebay”, we can figure out that all the strings in the subtrie rooted at “ko” cannot be similar to “ebay” based on Lemma 11, and thus those strings under “ko” (e.g., “kobe” and “koby”) can be pruned.

Using subtrie pruning, we can devise a trie-search-based method for similarity joins, called Trie-Search. Trie-Search first constructs a trie structure for all strings in \( R \), and then for each string \( s \in S \), computes the active-node set \( A_s \) of \( s \) based on subtrie pruning. We can also use the incremental algorithm [7] to compute the active-node sets. For each \( r \in A_s \), if \( r \) is a leaf node (i.e., \( r \in S \)), \( \langle s, r \rangle \) is a similar string pair. For example, in Figure 2, given a string \( s = “ebay” \), \( A_{“ebay”} = \{4, 11, 12\} \). As node 4 (“bay”) is a leaf node, (“ebay”, “bay”) is a similar string pair.

**Observation 2 - Dual Subtrie Pruning:** Subtrie pruning only utilizes the trie structure to index strings in \( R \). In fact, the strings in \( S \) also share prefixes, and we can do subtrie pruning for the strings in \( S \). To this end, we construct a trie for strings in both \( R \) and \( S \), and use the trie to do subtrie pruning for strings in both of the two sets. For example, in Figure 2 based on subtrie pruning, all the nodes in the subtrie rooted at “ko” can be pruned for the string “ebay” in \( S \). In terms of the similarity-join problem, there are a collection of strings of length \( v \) in \( S \), and all the strings cannot be similar to strings with prefix “ko”.

Thus we can prune the two subtries rooted at “eb” and “ko”.

**Figure 3: Dual subtrie pruning**

Based on this observation, we propose a new pruning technique, called dual subtrie pruning: Given a trie, for any two nodes \( u \) and \( v \), if \( u \) is not an active node for every ancestor of \( v \), and \( v \) is not an active node for every ancestor of \( u \), we can prune the subtries rooted at \( u \) and \( v \). The following Lemma shows the correctness of dual subtrie pruning.

**Lemma 2 (Dual Subtrie Pruning).** Given two trie nodes \( u \) and \( v \), if \( u \) is not an active node for every ancestor of \( v \), and \( v \) is not an active node for every ancestor of \( u \), the strings under \( u \) and \( v \) cannot be similar to each other.

For example, in Figure 2, consider node “ba” and node “ko”, as node “ba” is not an active node of “φ”, “k” and “ko” and node “ko” is not an active node of “φ”, “b” and “ba”, all strings in the subtrees of the two nodes cannot be similar, e.g., “bag” and “kobe”, “bag” and “koby”, “bay” and “kobe”, “bay” and “koby”. It is not straightforward to traverse the trie structure to find similar pairs using dual subtrie pruning. This paper proposes efficient trie-based algorithms.

### 3. Trie-Based Algorithms

In this section, using dual subtrie pruning, we propose three efficient algorithms. For ease of presentation, here we focus on self-join, that is \( R = S \). Our approach can be easily extended to \( R \neq S \) (Section 1).

#### 3.1 Trie-Traverse Algorithm

Recall the trie-search-based algorithm Trie-Search (Section 2.3), it can only use subtrie pruning, and cannot use dual subtrie pruning. To address this problem and improve performance, in this section we propose a trie-traversal-based method, called Trie-Traverse.

**Algorithm Description:** Trie-Traverse first constructs a trie index for all strings in \( R \), and then traverses the trie in pre-order. For each trie node, Trie-Traverse computes its active-node set. When reaching a leaf node \( l \), for \( s \in A_l \), if \( s \) is a leaf node (i.e., \( s \in S \)), Trie-Traverse outputs \( l, s \) as a similar string pair.

**Computing Active-Node Sets:** Obviously, for the root node, its active-node set is composed of the nodes with depth smaller than \( \tau \). For example, in Figure 1 suppose \( \tau = 1 \), \( A_0 = \{0, 1, 9, 13\} \). For each of other nodes \( n \), we compute its active-node set \( A_n \), using its parent’s active-node set \( A_p \), where \( p \) denotes the parent of \( n \). That is for each node in \( A_n \), it must have an ancestor in \( A_p \) based on dual subtrie pruning. The following Lemma shows the correctness.

**Lemma 3.** Given a node \( n \), let \( p \) denote \( n \)'s parent, for each node \( n' \in A_n \), there must exist a node \( p' \in A_p \), such that \( p' \) is an ancestor of \( n' \).

For example, in Figure 1 consider \( n = “kob” \) and its parent node \( p = “ko” \). To compute \( A_{“kob”} \), we only need to verify whether the descendants of nodes in \( A_{“ko”} = \{13, 14, 15\} \) are active nodes of “kob”. For the other nodes, e.g., node 2 (“ba”), its descendants (“bag” and “bay”) cannot be similar to the descendants of “ko” (“kobe” and “koby”) based on dual subtrie pruning.

Next we discuss how to use \( A_p \) to compute \( A_n \). For each active node \( p' \in A_p \), we verify whether each of \( p' \)’s descendants is an active node of \( n \), by considering the following operations: match, substitution, deletion, and insertion [8]. For example, in Figure 1, node 0 is the parent of node 9, and we compute \( A_0 \) based on \( A_0 = \{0, 1, 9, 13\} \) as follows. Consider node 0 in \( A_0 \), as we can do a deletion operation on node 0 (deleting c), thus node 0 is an active node of node 9. In addition, we can also do an substitution for node 0, by substituting “b” for “e”, thus node 1 is an active node of node 9. Similarly node 13 is an active node. We can do a match, thus node 9 is an active node. For node 9, we can do an insertion, thus node 10 is an active node. Thus we can compute \( A_0 = \{0, 1, 5, 9, 10, 13\} \). Similarly \( A_{10} = \{1, 9, 10, 11\} \), \( A_{11} = \{2, 10, 11, 12\} \), and \( A_{12} = \{4, 11, 12\} \).

Note that in the worst case the time complexity of computing \( A_n \) from \( A_p \) is \( O(\tau |A_p|) \), since each active node only can be computed from its ancestors within \( \tau \) steps. Therefore, the time complexity of Trie-Traverse is \( O(\tau \cdot |A_p|) \) where \( |A_p| \) is the sum of the numbers of the active-node sets of all the trie nodes in the trie \( T \). When traversing the trie nodes, we need to maintain the trie and the active nodes of ancestors of the current node. Given a leaf node \( l \), let \( C(l) \) denote the sum of the active nodes of ancestors of node \( l \), and \( C_{max} \) is the maximal value of \( C(l) \) among all leaf nodes.
The space complexity is $O(|T| + C_{\text{max}})$, where $|T|$ is the size of trie $T$.

**Algorithm 1: Trie-Traverse($S, \tau$)**

**Input:** $S$: a collection of strings  
$\tau$: a given edit-distance threshold  

**Output:** $P = \{(s \in S, t \in S) \mid \text{ED}(s, t) \leq \tau\}$

1. begin
2. $T =$ new Trie($S$);
3. Let $r$ denote the root of Trie $T$.
4. $A_r = \{n \mid \text{for each trie node } n, \text{s.t., } |n| \leq \tau\}$;
5. for each child node of $r$, $c$ do
   1. $P_c \cup = \text{findSimilarPair}(c, r, A_r)$;
7. end

**Function findSimilarPair($c, p, A_p$)**

**Input:** $c$: a trie node  
$p$: the parent of $c$  
$A_p$: the active node set of $p$

**Output:** $P_c = \{(s \in S, t \in S) \mid \text{ED}(s, t) \leq \tau \text{ and } s \text{ is a leaf descendant of } c\}$

1. begin
2. $A_c = \text{calcActiveNode}(c, A_p)$;
3. if $c$ is a leaf node then
   1. $P_c \cup = \text{outputSimilarPair}(c, A_c)$;
5. for each child node of $c$, $d$ do
   6. $P_c \cup = \text{findSimilarPair}(d, c, A_c)$;
7. end

**Function outputSimilarPair($n, A_n$)**

**Input:** $n$: a trie node  
$A_n$: $n$'s active-node set

**Output:** $P_n = \{(n, s \in A_n) \mid \text{ED}(n, s) \leq \tau\}$

1. begin
2. for each leaf node $l \in A_n$ ($n \neq l$) do
3. $P_n \cup = \{(n, l)\}$;
4. end

**Figure 4: Trie-Traverse algorithm**

Figure 4 gives the pseudo-code of the Trie-Traverse algorithm. It first constructs a trie index for all strings (Line 2), computes the active-node set of the root node (Line 3), and then calls its subroutine findSimilarPair to find all similar string pairs recursively (Lines 24-25). findSimilarPair first calculates the active-node set $A_c$ of node $c$ based on its parent’s active-note set $A_p$ (Line 3), using the incremental algorithm mentioned above. If $c$ is a leaf node, it calls a subroutine outputSimilarPair to output all the similar string pairs of $c$ (Line 24). Finally findSimilarPair calls itself to compute the similar string pairs of $c$’s descendants (Lines 22-25).

Example 1. Consider the string set and the corresponding trie structure in Figure 4. Initially, we construct a trie index for all strings. We compute the active-node set of the root node $A_0 = \{0, 1, 9, 13\}$, which is composed of the nodes with depths within $\tau = 1$, since their edit distances to the root node (an empty string) are within $\tau$. Then we compute active-node sets of every node using preorder traversal (following the dashed lines). This traversal can guarantee that, for each node we always compute its parent’s active-node set before its own active-node set. Consider node 2, we use its parent’s active-node set $A_1$ to compute its active-node set $A_2$. Similarly, we compute $A_3$ using $A_2$. As node 3 is a leaf node, and node 4 is a leaf node in $A_3 = \{2, 3, 4, 7\}$, thus we output the similar pair $(3, 4)$.

**Theorem 1.** Given a set of strings $S$ and an edit-distance threshold $\tau$, Trie-Traverse can compute all similar string pairs ($s \in S, t \in S$) such that $\text{ED}(s, t) \leq \tau$.

**Proof.** Let $T$ be a trie index constructed from $S$ with nodes $V(T)$ and edges $E(T)$. For a node $v$ in $T$, let $P(v)$ denote $v$’s pre-order number and $A_v = \{u \in V(T) \mid \text{ED}(u, v) \leq \tau\}$ denote the active-node set of $v$.

We first prove that after visiting a node $v$ in pre-order traversal, Trie-Traverse can compute active-node sets of $v$ and $v$’s ancestors correctly. We prove this claim by induction. The base case for the root node clearly holds since after visiting the root node $r$, it is obvious that $A_r = \{u \in V(T) \mid ||u|| \leq \tau\}$. Assume this claim holds for a node $v$. We want to prove that this claim also holds for the next node $v'$ (i.e. $P(v') = P(v) + 1$). Based on the pre-order traversal, $v'$ is either a child of $v$ or a child of some ancestor of $v$, thus active-node sets of $v$’s ancestors are correctly computed. Since the active-node set of $v$’s parent is correctly computed, Trie-Traverse can correctly compute $v'$’s active-node set. Then active-node sets of $v$ and $v$’s ancestors are also correctly computed. Therefore, the claim is proved.

Obviously for each string pair $(s \in S, t \in S)$, $\text{ED}(s, t) \leq \tau$ if and only if both the nodes $s$ and $t$ are leaf nodes of $T$, and $s$ is in $A_t$. Based on the proved claim, after visiting the node $t$, we can correctly compute the active-node set $A_t$. Also note that for each string $s$ in $S$, its corresponding node in $T$ must be a leaf node. Therefore, the theorem is proved.

**3.2 Trie-Dynamic Algorithm**

Trie-Dynamic has to compute the active-node sets for every trie node. However, we need not compute all of them. For instance, in Figure 1 consider node 3, as it is an active node of node 2 (i.e. $3 \in A_2$). Based on the symmetry property of active nodes: if $u$ is an active node of $v$, then $v$ must be an active node of $u$, node 2 must be in the active-node set of node 3 (i.e. $2 \in A_3$). Thus, we can avoid unnecessary computation when computing the active-node set of node 3.
Algorithm 2: Trie-Dynamic ($S, \tau$)

Input: $S$: a collection of strings  
$\tau$: a given edit-distance threshold

Output: $P = \{(s \in S, t \in S) \mid ED(s, t) \leq \tau\}$

1 begin
2 $T = \text{new Trie}();$
3 for each $s = s_1s_2\ldots s_m$ in $S$ do
4     Find trie node $t = s_1s_2\ldots s_i$ which is the longest prefix of $s$;
5     for $j = i+1$ to $m$ do
6         Append a new child node $c$ to node $t$;
7         $A_c = \text{calcActiveNode}(c, A_i)$;
8         /* update active nodes */
9         for each node $a \in A_c$  
10            $a \notin \text{add} a$ to $A_i$;
11         end for
12         if $t$ is a leaf node /*
13     end for
14 end for

Figure 6: Trie-Dynamic algorithm

Based on this observation, we design a new algorithm, called Trie-Dynamic, which avoids the redundant active-node computation introduced by Trie-Traversal. Figure 6 gives the pseudo-code. Initially, Trie-Dynamic constructs an empty trie with only a root node (Line 2), and then incrementally inserts strings into the trie. At each step, Trie-Dynamic maintains a trie index of all previously inserted strings. For a new string $s = s_1s_2\ldots s_m$, Trie-Dynamic inserts into the trie structure as follows (Lines 3-13). First Trie-Dynamic finds the trie node $t = s_1s_2\ldots s_i$ which is the longest prefix of $s$. Then Trie-Dynamic updates the trie by adding some new nodes under node $t$ (Line 6) and computing their corresponding active-node sets (Line 9). As the active-node set of an existing node may be affected by a newly added node, Trie-Dynamic updates all such active-node sets based on the symmetry property (Line 5).

As Trie-Dynamic utilizes the symmetry property of active nodes, its time complexity is reduced to $O(|T| + |A_T|)$. As it needs to keep active nodes of all trie nodes, its space complexity increases to $O(|T| + |A_T|)$. Example 4 shows how the Trie-Dynamic algorithm works.

Example 2. Consider the string set in Figure 8. Figure 9 shows how to dynamically construct the trie structure by adding a new string. Each node in the trie is associated with an ID and its active-node set. In Figure 8(a), we initialize a trie index with only a root node 0 and its active-node set $A_0 = \{0\}$. To insert a new string "bag", as every prefix of "bag" is not the trie node, we first insert node 0 with label "b" as a child of node 0 and compute its active-node set $A_1 = \{0, 1\}$. If $A_0 = \{0, 1\}$, then insert node 0 with label "a" as a child of node 0 and compute its active-node set $A_2 = \{1, 2\}$ using $A_1 = \{0, 1\}$, and update $A_1$ by inserting node 2, i.e., $A_1 = \{0, 1, 2\}$; finally insert node 3 with label "g" as a child of node 2 and compute its active-node set $A_3 = \{2, 3\}$ using $A_2 = \{1, 2\}$, and update $A_2$ by inserting node 3, i.e., $A_2 = \{1, 2, 3\}$. Figure 8(b) gives the detailed steps.

Similarly, we can insert "ebay" (Figure 8(c)). In Figure 8(d), we insert "bay" into the trie. As the prefix "ba" of "bay" is in the trie, we only need to create node 8 with label "y" and append node 8 as a child of node 2. Compared Figure 8(d) with Figure 8(c), we find that $A_2$, $A_4$, $A_7$ are different. Because after we insert node 8 and compute $A_8$ = \{2, 3, 7, 8\}, we update the active-node sets of nodes in $A_8$ (nodes 2, 3, 7). For each node $n$ in $A_8$, we add node 8 to $n$’s active-node set based on the symmetry property.

![Diagram](https://via.placeholder.com/150)

**Figure 7:** An example to use Trie-Dynamic algorithm to find all similar pairs ($\tau = 1$)

**THEOREM 2.** Given a set of strings $S$ and an edit-distance threshold $\tau$, Trie-Dynamic can compute all similar string pairs $(s \in S, t \in S)$ such that $ED(s, t) \leq \tau$.

**PROOF.** As Trie-Dynamic constructs the trie structure dynamically, let $T_\tau$ denote the trie index constructed from all the nodes inserted before $v$ (including itself). For a node $v$ in $T_\tau$, $P(v)$ denote the number of nodes in $T_\tau$. $T_\tau$’s node set $V(T_\tau) = \{v \in V(T) \mid ED(u, v) \leq \tau\}$ and its edge set $E(T_\tau) = \{(u, v) \in E(T) \mid ED(u, v) \leq \tau\}$. In addition let $A_v = \{a \in A(T) \mid ED(u, v) \leq \tau\}$ denote the active-node set of $u$ w.r.t. $T_\tau$.

We first prove that after adding a new node $v$ in the trie, for each node $u$ of $T_\tau$, Trie-Dynamic can compute $A_v$ correctly. We prove this claim by induction. The base case for the root node clearly holds since after adding a root node $r$, $T_r$ has only one node and $A_r = \{r\}$ is correct. Assume this claim holds for a node $v$. We want to prove that it also holds for the next added node $v'$ (i.e. $P(v') = P(v) + 1$). For each node $u$ in $T_{v'}$, if $u = v'$, then the active-node set of $v'$’s parent w.r.t. $T_\tau$ is correctly computed (as $u$’s parent is in $T_\tau$). Trie-Dynamic can correctly compute $A_{v'}$ (i.e. $A_{v'}$); if $u \neq v'$, then $u$ must be a node of $T_\tau$, so $A_u$ is correctly computed. Trie-Dynamic uses $A_u$ to compute $A_{v'}$. If $u$ is in $A_{v'}$, then $ED(u, v') \leq \tau$, thus Trie-Dynamic can obtain $A_v$ correctly by setting it as $A_{v'}$. Therefore, for each node $u$ of $T_{v'}$, Trie-Dynamic can compute $A_u$ correctly. Thus our claim is true.

Obviously for each string pair $(s \in S, t \in S)$ (without loss of generality, suppose $P(s) \leq P(t)$), $ED(s, t) \leq \tau$ if and only if both the nodes $s$ and $t$ are leaf nodes of $T_\tau$ and $s$ is in $A_t$. Based on the proved claim, after adding the node $t$, $A_t$ is correctly computed. Also note that for each string $s$ in $S$ and $P(s) \leq P(t)$, its corresponding node in $T_\tau$ must be a leaf node. Therefore, the theorem is proved. 

\[ \square \]
3.3 Trie-PathStack Algorithm

When inserting a new string, Trie-Dynamic may generate some new nodes and append them as children of any existing node. Thus Trie-Dynamic may use active-node sets of any existing node to compute the active-node sets of newly added nodes. For example, in Figure 1(d), when inserting a string “bay”, Trie-Dynamic generates a new node $S$ and appends it as a child of existing node $2$, and uses the active-node set of node 2 to compute the active-node set of the newly inserted node $S$. Thus although Trie-Dynamic avoids unnecessary active-node computation introduced by Trie-Traverse, Trie-Dynamic involve large memory space to maintain the active-node sets of all trie nodes.

Recall Trie-Traverse, it first constructs a trie index for all strings, and then gets similar string pairs by traversing the trie in pre-order. Throughout the algorithm, the maximal number of active-node sets that Trie-Traverse needs to maintain is the same as the maximal depth of trie leaf nodes. To summarize, Trie-Traverse uses little memory space but involves unnecessary active-node computation; on the contrary, Trie-Dynamic avoids such repeated computation but involves large memory space.

To address this problem, we propose a new algorithm, called Trie-PathStack, which not only requires little memory space but also achieves much higher performance. The basic idea behind Trie-PathStack is the following. First, when traversing the trie nodes, we maintain a “virtual partial” subtrie to keep the visited nodes. For each unvisited node, we first set it visited and then compute its active-node set in the virtual partial trie. For subsequent unvisited nodes, when computing their active nodes, we only consider the visited nodes. Thus we can avoid the redundant computation. Secondly, we traverse the trie nodes in preorder and use a stack to maintain the nodes that need to be updated. Throughout the preorder traversal, we use a stack to maintain the nodes from the root to the current node (with corresponding active-node sets). When visiting a node $n$, as its parent node must be the top element in the stack, we can use the active-node set of the top element to compute $n$’s active-node set. After computing $n$’s active-node set, we only need to update the active-node sets of the topmost $\tau$ elements (i.e., $n$’s ancestors within $\tau$ steps away from $n$) in the stack. Because we can guarantee that any unvisited node’s parent will be pushed into the stack, and only the topmost $\tau$ nodes are active nodes of $n$. Experimental results shows that Trie-PathStack can avoid a lot of unnecessary update.

Figure 8 shows the pseudo-code. Initially, Trie-PathStack constructs a trie structure $T$ for all strings (Line 1). To avoid repeated active-node computation, we logically maintain a virtual partial trie index consisting of the nodes marked by “visited”. In the beginning, we only set the root “visited” (Line 2). Accordingly, in this partial trie we define the active-node set of a node $u$ as $A_u = \{ v | v \in A_u \text{ } v \text{ has been visited} \}$ and we can get $A_u = \{ \} \text{ (Line 12)}$. Throughout the Trie-PathStack, we use a stack $\mathcal{S}$ to maintain active-node sets of nodes from the root node to the current node. When pushing a new node $c$ into the stack, we first compute $c$’s active-node set $A_c$ based on its parent’s active-node set $A_{\text{parent}(c)}$. For subsequent unvisited nodes, we only need to update the active-node sets of the topmost $\tau$ nodes, so we will not use their active-node sets. Therefore Trie-PathStack significantly decreases the number of updated active-node sets from $|A_u|$ to $\tau$ and performs more efficient than Trie-Dynamic, although they both take into account the symmetry property of active nodes. We can see the time complexity of Trie-PathStack is $O(\tau \cdot |A_T|)$ and the space complexity is $O(|T| + C_{\text{max}})$. In comparison with Trie-Traverse and Trie-Dynamic, Trie-PathStack can achieve high performance with less memory. Example 3 shows how Trie-PathStack works.

Example 3. Consider the string set and the corresponding trie structure in Figure 4. Figure 8 shows how to use Trie-PathStack to compute similar pairs. In the initial step, besides constructing a trie index, we also create a stack from the root node to the current node. We first push node 0 and its active-node set $A_0 = \{0\}$ into the stack, and get its first child, node 1 (Figure 4(a)). In Figure 4(b), we compute $A_1 = \{0,1\}$ using $A_0 = \{0\}$. Though node 2 is also an

![Algorithm 3: Trie-PathStack ($\mathcal{S}, \tau$)](image)

Input: $\mathcal{S}$: a collection of strings $\tau$: a given edit-distance threshold Output: $\mathcal{P} = \{(s \in \mathcal{S}, t \in \mathcal{S}) | \text{ED}(s,t) \leq \tau\}$

1. $\mathcal{T} = \text{new Trie}(\mathcal{S})$;
2. $\mathcal{S} = \text{new Stack}();$
3. Let $r$ denote the root of Trie $\mathcal{T}$ and set $r$ visited.
4. $A_r = \{\} ;$
5. $\mathcal{S}.\text{push}(\langle r, A_r \rangle);$ 
6. $c = r.\text{firstchild}$;
7. while not $\mathcal{S}.\text{empty}()$ do 
   8. while $c$ is not null do 
      9. $\langle p, A_p \rangle = \mathcal{S}.\text{top}();$
      10. Set $c$ visited;
      11. $A_r = \text{calcActiveNode}'(c, A_r);$ 
      12. /* update active nodes */ 
      13. for each ancestor node of $c, t$ do 
          14. if $|c| - |t| \leq \tau$ then 
              15. $\mathcal{P} \cup = \text{outputSimilarPair}(c, A_c);$ 
          16. if $c$ is a leaf node then 
              17. $\mathcal{S}.\text{push}(c, A_c);$ 
              18. $c = c.\text{firstchild};$
              19. $\langle p, A_p \rangle = \mathcal{S}.\text{pop}();$
              20. $c = p.\text{nextSibling};$
   21. end
active node of node 1, we ignore it since it is unvisited in pre-order traversal. We then update the active-node sets of its ancestors by adding node 1 to \( A_u \) (the underlined number). We repeat these steps until visiting node 3 which has no children. We pop node 3 (Figure 9(a)) from the stack and push its sibling node 4 into the stack (Figure 9(b)). We continue to push the first child of node 5 (if any). When visiting a leaf node, i.e., nodes 3 and 4, we output the similar string pairs. We repeat above steps until the stack is empty.

**Theorem 3.** Given a set of strings \( S \) and an edit-distance threshold \( \tau \), Trie-Traversal can compute all similar string pairs \( \langle r, s \rangle \in S \times S \) such that \( ED(r, s) \leq \tau \).

**Proof.** Let \( T \) be a trie index constructed from \( S \) with nodes \( V(T) \) and edges \( E(T) \). For a node \( v \) in \( T \), let \( P(v) \) denote \( v \)'s pre-order number and \( A_v = \{ u \in V(T) \mid ED(u, v) \leq \tau \} \) denote the active-node set of \( v \).

We first prove that after visiting a node \( v \) in pre-order traversal, Trie-Traversal can compute active-node sets of \( v \) and \( v \)'s ancestors correctly. We prove this claim by induction. The base case for the root node clearly holds since after visiting the root node \( r \), it is obvious that \( A_r = \{ u \in V(T) \mid |u| \leq \tau \} \). Assume this claim holds for a node \( v \). We want to prove that this claim also holds for the next node \( v' \) (i.e., \( P(v') = P(v) + 1 \)). Based on the pre-order traversal, \( v' \) is either a child of \( v \) or a child of some ancestor of \( v \), thus active-node sets of \( v \)'s ancestors are correctly computed. Since the active-node set of \( v' \)'s parent is correctly computed, Trie-Traversal can correctly compute \( v' \)'s active-node set. Then active-node sets of \( v' \) and \( v' \)'s ancestors are also correctly computed. Therefore, the claim is proved.

Obviously for each string pair \( \langle s \in S, t \in S \rangle \), \( ED(s, t) \leq \tau \) if and only if both the nodes \( s \) and \( t \) are leaf nodes of \( T \), and \( s \in A_r \). Assume for the proved claim, after visiting the node \( t \), we can correctly compute the active-node set \( A_t \). Also note that for each string \( s \) in \( S \), its corresponding node in \( T \) must be a leaf node. Therefore, the theorem is proved.

### 3.4 Improving Trie-PathStack on Large Edit-Distance Thresholds

In this section, we improve Trie-PathStack to support large edit-distance thresholds. Consider a string \( r = r_1r_2\ldots r_n \). We divide \( r \) into two parts \( r_1r_2\ldots r_\lfloor \frac{n}{2} \rfloor \) and \( r_{\lceil \frac{n}{2} \rceil + 1}\ldots r_n \). Note that for a string \( s \), if \( r \) is similar to \( s \) within distance threshold \( \tau (\tau \leq n) \), then at least one of the following condition is correct: 1) the first part of \( r \), \( r_1r_2\ldots r_{\lfloor \frac{n}{2} \rfloor} \) is similar to a prefix of \( s \) within \( \lfloor \frac{n}{2} \rfloor \); 2) the second part of \( r \), \( r_{\lceil \frac{n}{2} \rceil + 1}\ldots r_n \) is similar to a suffix of \( s \) within \( \lfloor \frac{n}{2} \rfloor \). For example, given a string \( r = "arnold schwarzeneger" \), if string \( s \) is similar to \( r \) within \( \tau = 5 \), then either the edit distance between a prefix of \( s \) and "arnold sch" is within 2 or the edit distance between a suffix of \( s \) and "warzeneger" is within 2. We use this property to improve Trie-PathStack.

Given a string set \( S \) and a threshold \( \tau \), we first discuss how to use Trie-PathStack to find all the string pairs \( \langle r, s \rangle \in S \times S \) such that the first part of \( r \) is similar to a prefix of \( s \) within \( \lfloor \frac{n}{2} \rfloor \). We can construct a new string set \( S^1 \) that consists of the first part of each string in \( S \). Then we run the Trie-PathStack on \( S^1 \) and \( S \) with edit-distance threshold \( \lfloor \frac{n}{2} \rfloor \). For a string \( c \in S^1 \), to find all the strings in \( S \) whose prefix is similar to \( c \), we traverse the subtrie rooted at the active nodes in \( A_n \) and get the leaf nodes. Clearly, these leaf nodes have a prefix that is similar to \( c \). 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 second part of \( r \) is similar to a suffix of \( s \) within \( \lfloor \frac{n}{2} \rfloor \). Based on the candidates generated from the two cases, we verify them to generate final results.

### 4. PRUNING TECHNIQUES

In this section, we propose three pruning techniques to reduce the sizes of active-node sets which can further improve performance of our proposed algorithms.

**Length Pruning:** Consider two strings \( r \) and \( s \), if their length difference is larger than \( \tau \), their edit distance cannot be within \( \tau \). We exploit this property for pruning in our framework. In Figure 10(a) in the left box, for each node, we maintain a range of lengths of strings in the subtrie, \([l_u, l_t] \), where \( l_u \) is the length of the shortest string in the subtrie and \( l_t \) is the length of the longest string in the subtrie. For instance, the length range of strings in subtrie of \( v \) is \([5, 7] \) and that of \( u \) is \([2, 3] \). As the lengths of strings from the two subtrees have at least two differences (larger than \( \tau = 1 \)), node \( v \) can be pruned from \( A_u \) through length pruning, although node \( v \) is an active node of node \( u \).

**Single-branch Pruning:** If node \( v \) is an ancestor node of node \( u \) and their subtrees have the same leaf nodes, then node \( v \) can be pruned from \( A_n \), even if node \( v \) is an active node of node \( u \). Intuitively, as there is only a single branch from node \( v \) to node \( u \), when we use \( A_u \) to compute the active-node sets of \( u \)'s children, \( v \) will not generate new leaf active nodes, thus we can remove \( v \) from \( A_u \). We call this pruning technique single-branch pruning. For instance, in the center box of Figure 10(b), as node \( v \) and node \( u \) have the...
same leaf nodes, based on single-branch pruning, v can be pruned from $A_v$.

**Count Pruning:** Given two nodes $v$ and $u$, if there is only one string that have both nodes $v$ and $u$ as prefixes, node $v$ can be safely pruned from $A_v$ because we cannot find two strings in their subtries. As an example in the right box of Figure 10, $v$ can be excluded from $A_v$, since we cannot find a similar string pair in both of their subtries.

We give an example to illustrate how to use the three techniques for pruning. In Figure 2 consider computing the active-node set of node 6, we have $A_6 = \{2, 5, 6, 7\}$. Using length pruning, we have $A_6 = \{5, 6, 7\}$. Using single-branch pruning, we have $A_6 = \{6, 7\}$. Using count pruning, we have $A_6 = \{\}$. Using the three pruning techniques, we can significantly reduce the number of active nodes.

### 5. INCREMENTAL SIMILARITY JOINS

In this section, we discuss how to extend our method to support dynamic update of data sets efficiently. For ease of presentation, we choose Trie-PathStack algorithm as an example to explain our idea. Suppose we have gotten the self-join results of a string set $S$, and then $S$ is updated by adding another string set $\Delta S$, it is challenging to do the similarity join incrementally. We formalize the incremental similarity-join problem as follows.

**Definition 2 (Incremental Similarity Joins).** Given a set of strings $S$, a new string set $\Delta S$, and an edit-distance threshold $\tau$, an incremental similarity join finds all similar string pairs $(r, s) \in \Delta S \times S$ such that $ED(r, s) \leq \tau$.

**Algorithm 4: IncrementalTrieJoin($T, \Delta S, \tau$)**

**Input:** $T$: a trie index of original collection of strings $\Delta S$: a new added collection of strings $\tau$: a given edit-distance threshold **Output:** $P = \{(s, t) \in \Delta S \times S \mid ED(s, t) \leq \tau\}$

1. Change Line 2 in Trie-PathStack algorithm to “$A_v = \{\text{null}\}$ for each visited trie node $v$.”
2. Change Line 2 in Trie-PathStack algorithm to “while $c$ is not null and $c$ is not visited”;
3. Change Line 2 in Trie-PathStack algorithm to “if $c$ is not null then $c = c.nxtSibling$;”
4. Change Line 2 in Trie-PathStack algorithm to “else $(p, A_p) = S.pop()$; $c = p.nxtSibling$;”

**Figure 11:** Incremental trie-join algorithm

We use Trie-PathStack algorithm as an example to show our idea that how to extend trie-join algorithms to support incremental similarity joins, and we can easily extend other algorithms to support update. Consider the trie index $T$ constructed from $S$. Given a new string set $\Delta S$, we update the original trie $T$ to $T'$ by inserting the strings in $\Delta S$. In the updated trie $T'$, let $\Delta T$ denote the partial trie for strings $\Delta S$. Then we extend Trie-PathStack to find similar string pairs for trie nodes in $\Delta T$ as follows. When reaching a trie node $n$, different from Trie-PathStack which computes $n$’s active-node set $A_n$ from visited nodes, the incremental similarity-join algorithm computes $A_n$ from the nodes in $T'$. Figure 12 shows the pseudo-code. Comparing to Trie-PathStack in Figure 2. Firstly, we update the original trie $T$ by inserting all strings in $\Delta S$ and set the nodes in $\Delta T$ “unvisited”. Secondly, we initialize $A_n$ as $r$’s visited active nodes. Thirdly, we change the condition of pushing an element into the stack. Fourthly, we need push all unvisited elements into the stack.

**Example 3** shows how the algorithm works.

**Figure 12:** Incremental similarity joins on sample data set in Figure 2 ($\Delta S = \{\text{"beagy"}\}$, $\tau = 1$)

**Example 4.** Consider the trie structure $T$ in Figure 2. Suppose $\Delta S = \{\text{"beagy"}\}$. Based on our incremental trie-join algorithm, we update the original $T$ to $T'$ by inserting “beagy” (Figure 13) and get the partial trie $\Delta T$ marked by the dot lines. Then we traverse the trie $\Delta T$ to find similar string pairs. Initially, we push node 0 and its active-node set $\{0, 1, 13\}$ into the stack. Nodes 1 and 13 are the active nodes in $T'$. Next we push nodes 9, 10 and 18 into the stack. When reaching leaf node 18 (Figure 13(d)), we output similar string pair (18, 12). The algorithm stops when the stack is empty.

### 6. SIMILARITY JOINS BETWEEN TWO DIFFERENT SETS

In this section, we discuss how to extend our algorithm to support similarity joins between two different sets $R$ and $S$. We choose Trie-PathStack algorithm as an example to explain our idea, and our techniques can be easily adapted to other algorithms with minor modifications. For ease of presentation, we first introduce a concept.

**Definition 3.** Given a trie $T$, a trie node $n$, and a string set $S$, node $n$ belongs to $S$ if there exists a string $s$ in $S$ with a prefix $n$.

For example, in the left of Figure 14, given the trie index and $S = \{\text{bag, beagy}\}$, node “be” belongs to $S$, since there exists a string $s = \"beagy\"$ with a prefix “be”.

We take Trie-PathStack as an example to introduce our idea and propose an algorithm, called Trie-PathStack$^+$ as shown in Figure 13. Different from Trie-PathStack algorithm, Trie-PathStack$^+$ builds a trie index on strings in $R \cup S$ (Line 4) and for each node belonging to $R$, computes its active-node set composed of nodes belonging to $S$. The active-node set of node $r$ is defined as $A_r^+=\{n\}$ for each trie node $n$, such that $|n| \leq \tau$ and $n \in S$ (Line 2), and calcActiveNode$^+$ returns those active nodes that belong to
Procedure 5: Trie-PathStack\(^+\) (R, S, \tau)

Input: R, S: two collections of strings
\(\tau\): a given edit-distance threshold

Output: \(P = \{ (s, t) \in R \cup S | ED(s, t) \leq \tau \}\)

1. \(T = \text{new Trie}(R \cup S);\)
2. \(S = \text{new Stack}();\)
3. Let \(r\) denote the root of Trie \(T\).
4. \(A_r'' = \{n\}\) for each trie node \(n\), s.t., \(|n| \leq \tau\) and \(n \in S\).
5. \(S.\text{push}((r, A_r''));\)
6. \(c = r.\text{firstchild}();\)
7. \(\text{while not } S.\text{empty}()\) do
   - \(\text{while } c \text{ is not null and } c \in R\) do
     - \((p, A_p'') = S.\text{top}();\)
     - \(A_c'' = \text{calcActiveNode}''(c, A_p'');\)
     - \(c = c.\text{nextsibling}();\)
   - \(\text{if } c \text{ is not null then}\)
     - \(c = c.\text{firstchild}();\)
   - \(\text{else}\)
     - \((p, A_p'') = S.\text{pop}();\)
     - \(c = p.\text{nextsibling}();\)
8. end

Figure 13: Trie-PathStack\(^+\): a similarity-join algorithm for two different sets

\(S\). We restrict that only nodes \(u \in R\) can be pushed into the stack (Line 4).

Example 4 shows how the algorithm works.

![Diagram of Trie-PathStack\(^+\)](image)

Figure 14: Similarity joins between \(R = \{\text{bay, ebay}\}\) and \(S = \{\text{bag, beagy}\}\) (\(\tau = 1\)). We push nodes in \(R\) into the stack and find their active nodes in \(S\).

Example 5. In Figure 14, we illustrate an example to join two different string sets. On the left, it is the trie index for strings in \(R = \{\text{bay, ebay}\}\) and \(S = \{\text{bag, beagy}\}\). Each node is marked by belonging to which set, such as \(R, S\) or \(R \cup S\). In Figure 14(a), the stack is initialized with node 0 and \(A_0'' = \{0\}\). Though node 9 is similar to node 0, it is excluded from the set since node 9 \(\notin S\). After pushing node 2 into the stack (Figure 14(c)), we then push node 4 into the stack, but will not push node 3 as node 3 \(\notin R\). In Figure 14(d), as node 4 is a leaf node, we output similar string pair (4, 3) by finding the leaf node in \(A_r'' = \{2, 3\}\).

We continue these steps until the stack is empty.

### 7. EXPERIMENTS

We have implemented our method and conducted an extensive set of experimental studies on three real data sets: English Dict, DBLP Author, and AOL Query Log.

#### Data sets

1. English Dict. It was composed of English words from the Aspell spellchecker for Cygwin.
2. DBLP Author. We extracted author names from DBLP dataset.
3. AOL Query Log. We randomly chose one million distinct queries.

Table 1 illustrates detailed statistical information of the three data sets. Figures 15(a)-15(c) show their length distribution respectively.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>min</th>
<th>avg_len</th>
<th>max_len</th>
<th>min_len</th>
<th>max_len</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>English Dict</td>
<td>748</td>
<td>8.77</td>
<td>27</td>
<td>4</td>
<td>46</td>
<td>4</td>
</tr>
<tr>
<td>DBLP Author</td>
<td>613,542</td>
<td>12.82</td>
<td>46</td>
<td>4</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>AOL Query Log</td>
<td>1,000,000</td>
<td>20.94</td>
<td>200</td>
<td>1,000,000</td>
<td>37</td>
<td></td>
</tr>
</tbody>
</table>

Experimental settings: We compared our algorithms with state-of-the-art methods, Part-Enum [I], All-Pairs-Ed [4], Ed-Join [5] and Trie-PathStack\(^+\).

All-Pairs-Ed [2] is a q-gram-based algorithm. It generates \(|s| - q + 1\) q-grams for each string \(s\), and selects the first \(q + 1\) grams as gram prefix according to the pre-defined ordering on all grams. Those string pairs that do not share any gram will be filtered and the survived string pairs will be verified by the edit-distance calculation.

Ed-Join [5] improves All-Pairs-Ed with both location-based and content-based mismatch filtering. Location-based filtering decreases the number of grams in the prefix of each string and content-based filtering reduces the amount of edit distance verification.

Part-Enum [I] takes the q-gram set of a string as a feature vector. For two strings, if their edit distance is within \(\tau\), then the hamming distance between their feature vectors is smaller than \(q\tau\). They use this property for filtering. Part-Enum includes two steps: 1) Partitioning. They divide every feature vector into \(n_1\) partitions; 2) Enumeration. For each partition, they further divide it into \(n_2\) sub-partitions and generate several signatures. Finally, those string pairs that share no signatures will be filtered.

For All-Pairs-Ed and Ed-Join, we downloaded their binary codes from “Similarity Joints” project site [4]. For Part-Enum, we modified the implementation in Flamingo Project [5] to support string similarity joins with edit-distance constraints. All the algorithms were implemented in C++ and compiled using GCC 4.2.3 with -O3 flag. All the experiments were run on a Ubuntu machine with an Intel Core 2 Quad X5450 3.00GHz processor and 4 GB memory.

7.1 Comparison of Four Trie-Based Algorithms

In this section, we evaluate our trie-join algorithms and compare them with the baseline algorithm Trie-Search on the three data sets. Figures 16(a)-16(c) illustrate their performance by varying different edit-distance constraints. Our three trie-join algorithms outperform Trie-Search, even by 1-2 orders of magnitude on the AOL data set. Trie-Traverse is approximately two times slower than Trie-Dynamic and Trie-PathStack, as Trie-Traversal does

1. [http://www.informatik.uni-trier.de/~ley/db]
2. [http://www.gregsdetsky.com/aol-data/]
3. [http://www.cse.unsw.edu.au/~weiw/project/simjoin.html]
4. [http://flamingo.ics.uci.edu/]


Dynamic needs to update because after inserting (visiting) a new trie node \( n \) implemented and incorporated them into 7.2 Evaluation of Pruning Techniques Trie-Dynamic Trie-PathStack
tive nodes and involves a lot of unnecessary computation.

Table 2: Maximal #active notes on AOL Query Log

<table>
<thead>
<tr>
<th>τ</th>
<th>Trie-Search</th>
<th>Trie-Traversal</th>
<th>Trie-Dynamic</th>
<th>Trie-PathStack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2444</td>
<td>42546799</td>
<td>2172</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>31374</td>
<td>23051829</td>
<td>18477</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>257896</td>
<td>244928000</td>
<td>201825</td>
<td></td>
</tr>
</tbody>
</table>

not take into account the symmetry property of two active nodes and involves a lot of unnecessary computation. Trie-PathStack also outperforms Trie-Dynamic. This is because after inserting (visiting) a new trie node \( n \), Trie-Dynamic needs to update \( |A_n| \) active-node sets, while Trie-PathStack only updates \( \tau (\leq |A_n|) \) active-node sets. Table 4 illustrates the maximal number of active nodes that four algorithms need to store. We can see that Trie-Dynamic keeps a rather large number of active nodes, since it needs to maintain the active-node sets of all trie nodes. For the other algorithms, the maximal number of active-node sets is the same as the maximal depth of trie leaf nodes. The number of active nodes for Trie-PathStack is smaller than that of Trie-Search and Trie-Traversal, since Trie-PathStack utilizes the symmetry property of two active nodes.

7.2 Evaluation of Pruning Techniques

To evaluate the effect of the three pruning techniques, we implemented and incorporated them into Trie-PathStack, and compared them with Trie-PathStack without pruning on the AOL data set. We used the number of pruned active nodes to test the pruning power. Figure 14 shows the results. In the figure, "No Pruning", "Length", "Single Branch", "Count", and "All Pruning" respectively denote Trie-PathStack without any pruning technique, with length pruning, with single-branch pruning, with count pruning, and with all three pruning techniques. We can see that the three pruning techniques indeed can prune useless active nodes. For example, length pruning can prune 25% useless active nodes for the edit-distance threshold \( \tau = 2 \) and count pruning nearly prunes 50% useless active nodes for \( \tau = 1 \).

Table 3: Performance improvement of three pruning techniques on AOL Query Log

<table>
<thead>
<tr>
<th>τ</th>
<th>No Pruning(Second)</th>
<th>Pruning(Second)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.76</td>
<td>7.35</td>
<td>24.7%</td>
</tr>
<tr>
<td>2</td>
<td>122.48</td>
<td>104.48</td>
<td>14.7%</td>
</tr>
<tr>
<td>3</td>
<td>1174.84</td>
<td>1006.12</td>
<td>9.3%</td>
</tr>
</tbody>
</table>

Figure 19: Numbers of active nodes of Trie-PathStack with different pruning techniques on the AOL data set

In addition, we also compared the running time of various algorithms and Table 4 shows the results. We can
see that the three pruning techniques can improve the performance beyond Trie-PathStack by 24.7% when $\tau = 1$, 14.7% when $\tau = 2$, and 9.3% when $\tau = 3$. These results confirm that our proposed pruning techniques can improve the performance. As Trie-PathStack with all pruning techniques achieves the best performance, we use it to compare with existing algorithms in the remainder of this paper.

### 7.3 Comparison with Existing Methods

#### Index sizes

We compared index sizes with the state-of-the-art methods on three data sets, as shown in Table 4. We tuned their parameters and compared with their best performance. We can observe that existing methods involve much more memory than our method. For example, their index sizes for the AOL data set are larger than 100MB, while our method only has 29MB. The reason is that they indexed a large number of signatures for the data set, but our method only has 29MB.

#### Efficiency

We compared efficiency of the four algorithms for Ed-Join, Trie-PathStack, and Bi-Trie-PathStack on DBLP Authors + Title data set (Note that Ed-Join did not finish in $10^6$ seconds for $\tau = 7, 8$ and the string length 10.)

Figure 18: Comparison of Ed-Join, Trie-PathStack, and Bi-Trie-PathStack on DBLP Authors + Title data set

#### Table 4: Index sizes (MB)

<table>
<thead>
<tr>
<th>Data Sets</th>
<th>Trie-PathStack</th>
<th>Part-Enum</th>
<th>All-Pairs-Ed</th>
<th>Ed-Join</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dict</td>
<td>2</td>
<td>16</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>DBLP</td>
<td>16</td>
<td>94</td>
<td>135</td>
<td>65</td>
</tr>
<tr>
<td>AOL</td>
<td>29</td>
<td>120</td>
<td>300</td>
<td>155</td>
</tr>
</tbody>
</table>

#### Algorithm Selection

To help users select a good algorithm, we conducted an experiment to suggest which algorithms should be used for different data sets.

We used the DBLP Authors + Title data set in [18], in which each string is a concatenation of author names and the title of a publication. We truncated the prefix of each string with lengths of 10, 20, 30, 40, 50, and 60, and accordingly generated 6 data sets with different length distributions. In Figure 13, we compared the running time of three algorithms (Ed-Join, Trie-PathStack, and our improved Trie-PathStack using bidirectional filtering for both prefixes and suffixes as discussed in Section 7.1, called Bi-Trie-PathStack) by varying the edit-distance thresholds from 1 to 8. Note that for Ed-Join we adjusted the $q$ values to achieve the best performance and used its best results. From Figures 13(a)-(h), we can see that when the average string length is no larger than 30, Bi-Trie-PathStack is always superior to Ed-Join. This is because for these short strings, it is hard to select high-quality $q$-grams, and thus Ed-Join has low pruning power and will result in a large number of candidates which need to be further verified.

Even when the average string length is larger than 30, for small thresholds ($\tau \leq 3$ in Figures 13(a)-(c)), Bi-Trie-PathStack is still better than Ed-Join. This is because when $\tau \leq 3$, Bi-Trie-PathStack only needs to run Trie-PathStack twice for threshold $|q| \leq 1$, and Trie-PathStack is very efficient for smaller edit-distance thresholds. In addition, Bi-Trie-PathStack generates a smaller number of
7.4 Evaluation of Update

In this section, we evaluate the performance of our incremental similarity-join algorithm on AOL Query Log. Initially, we selected 500K strings (S), and for each time, we updated it by inserting 100K strings (∆S). We compared the running time between incremental Trie-Join and Trie-PathStack. We used speed-up to evaluate the benefit of our method, which is the ratio between the running time of two algorithms. Figure 20 shows the results. We can see that, with the increase of data sets, the speed-up of incremental Trie-Join against Trie-PathStack (from scratch) tends to be larger. For example, in Figure 20(a), the speed-up for |S| = 500K is 4 and that for |S| = 900K is 5. This result shows the superiority of our incremental algorithms.

7.5 Evaluation of Joining Two Data Sets

To evaluate the similarity join between two different data sets, we selected 200K and 400K strings from DBLP Author and tested the running time of joining them and the experimental results are shown in Figure 21. Suppose we push candidates than Ed-Join and thus achieves higher efficiency.

Figures 20(a)-(c) also show that when τ is small, Trie-PathStack has a good performance for short strings (the average string length is no larger than 30). It is even faster than Bi-Trie-Path Stack in some cases. This is because for short strings, both Bi-Trie-PathStack and Ed-Join will generate a large number of candidates which need to be further verified, but Trie-PathStack can directly generate all results. For larger thresholds (τ ≥ 4) and longer strings (the average string length is larger than 30), as shown in Figures 20(d)-(h), Ed-Join is more efficient than our algorithms since in these cases, both Trie-PathStack and Bi-Trie-PathStack are expensive to compute active nodes while Ed-Join can select high-quality q-grams with low frequency and has high pruning power.

Table 5: Algorithm selection

<table>
<thead>
<tr>
<th>Avg. Length</th>
<th>τ = 1</th>
<th>τ = 2</th>
<th>τ = 3</th>
<th>τ ∈ [4, 8]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(20, 30)</td>
<td>TP</td>
<td>TP</td>
<td>TP/TP</td>
<td>TP/TP</td>
</tr>
<tr>
<td>(30, 40)</td>
<td>TP</td>
<td>Bi-TP</td>
<td>Bi-TP</td>
<td>Bi-TP</td>
</tr>
<tr>
<td>(40, 60)</td>
<td>Bi-TP</td>
<td>Bi-TP</td>
<td>Bi-TP</td>
<td>Bi-TP/Ed-J</td>
</tr>
</tbody>
</table>

7.6 Scalability

We evaluated the scalability of Trie-PathStack and incremental trie-join algorithm on AOL Query Log. Initially, we used the DBLP Author+Title dataset with average string length of 50 (Section 5.3). We set τ = 8. Bi-Trie-PathStack took 29 MB for keeping the trie and 11 MB nodes in $R$ into the stack and traverse the trie to find active nodes in $S$. We can see that it is better to assign $R$ as the set with smaller size. This is because the smaller number of nodes pushed into the stack, we need less time to traverse the trie to find active nodes.

Figure 22: Scalability on AOL Query Log (e.g. 6+1 denotes |S| = 600K, |∆S| = 100K)

We also evaluated Bi-Trie-PathStack (Section 5.3) for the case that the active nodes cannot fit in main memory. We used the DBLP Author+Title dataset with average string length 50 (Section 5.3). We set τ = 8. Bi-Trie-PathStack took 29 MB for keeping the trie and 11 MB
for maintaining the active nodes (Bi-Trie-PathStack only needs to maintain the active nodes of the trie nodes from the root to a leaf node). To evaluate the I/O behavior, we set the available main memory buffer was 10% of the maximum memory. As it needs to read/write disk, the running time of Bi-Trie-PathStack increased to 6.3 * 10^4 second from 4 * 10^4 second for in-memory setting.

8. RELATED WORK

String similarity joins have been extensively studied [6, 11, 12, 13, 14, 15]. Gravano et al. [14] proposed to use SQL statements for similarity joins in relational databases. Chaudhuri et al. [6] proposed a primitive operator for effective similarity joins. Arasu et al. [12] developed a signature scheme which can be used as a filter for effective similarity joins. Bayardo et al. [13] proposed all-pair similarity joins, a prefix-filtering based algorithm. Xiao et al. [15] proposed ppjoin to improve all-pair algorithms by introducing positional filtering and suffix filtering. Xiao et al. [13] also studied top-k similarity joins, which can directly find the top-k string pairs without a given threshold.

In addition, there have been many studies on approximate string search [6, 11, 12, 13] and approximate string matching [13]. Given a collection of data strings and a query string, approximate string searching finds all the strings in the collection similar to the query string. Schulz and Milhow [13] proposed an automaton-based approach to address this problem. Sahinalp et al. [13] proposed an index structure called “VP-tree” for answering NN queries in terms of an edit-distance function. Kim et al. [12] proposed a novel technique called “n-Gram/2L” to improve the space and time complexity for q-gram index structures. Li et al. [12] proposed a new technique called VGRAM to judiciously select high-quality grams with variable lengths from a collection of strings for supporting approximate string queries efficiently. Li et al. [12] developed several list-merging algorithms to improve search efficiency by skipping elements on q-gram inverted lists. Navarro [13] studies approximate string matching, which given a query string and a text string, finds all substrings of the text string that are similar to the query string. These studies can be used to look for common gene expressions. Note that these two problems are different from our similarity-join problem, which given two sets of strings, finds all similar string pairs.

The trie-based approach to deal with string similarity for edit distance has been proposed in [6, 11, 12], but they focus on a different problem, fuzzy type-ahead search which returns answers as users type in keywords letter by letter. They emphasized an incremental algorithm to answer a query based on the query’s prefixes. They are not designed for the similarity-join problem. A straightforward method to extend their methods to support similarity joins is as follows. Given two string sets $R$ and $S$, for each string in $R$, we find its similar strings from set $S$. As discussed in Section 4, this method is inefficient as they cannot utilize the fact that the strings in $S$ also share common prefixes. We propose new effective algorithms and pruning techniques.

9. CONCLUSION

In this paper we have studied the problem of string similarity joins with edit-distance constraints. We proposed a new trie-based similarity-join framework which can efficiently find all similar string pairs with small indexes. We used a trie structure to index strings and devised three trie-join algorithms based on dual subtrie pruning to achieve high performance. We developed several optimization techniques to enhance performance. We also extended our method to efficiently support dynamic update of data sets. We have implemented our algorithms and our approach outperforms state-of-the-art methods on data sets with short strings (the average string length is no larger than 30).

10. REFERENCES


13