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

Jiannan Wang Jianhua Feng Guoliang Li
Department of Computer Science and Technology, 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 [4] 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 $\mathcal{R} = \{\texttt{kobe}, \texttt{ebay}, \dots\}$ and $\mathcal{S} = \{\texttt{bag}, \texttt{koby}, \dots\}$. We want to find all similar pairs $\langle r, s\rangle \in \mathcal{R} \times \mathcal{S}$, such as $\langle \texttt{kobe}, \texttt{koby}\rangle$.

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 [1], All-Pairs-Ed [2], Ed-Join [19], 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).

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 $\text{ED}(r, s)$, is the minimum number of single-character edit operations (i.e., insertion, deletion, and substitution) needed to transform r to s. For example, $\text{ED}(\texttt{koby}, \texttt{ebay})$=3. In this paper two strings are *similar* if their edit distance is no 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 $\mathcal{R}$ and $\mathcal{S}$, and an edit-distance threshold τ, a similarity join finds all similar string pairs $\langle r, s\rangle \in \mathcal{R} \times \mathcal{S}$ such that $\text{ED}(r, s) \le \tau$.*

2.2 Prefix Pruning

One naïve solution to address this problem is *all-pair verification*, which enumerates all string pairs $\langle r, s\rangle \in \mathcal{R} \times \mathcal{S}$ 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 [15].

Given two strings $r = r_1 r_2 \ldots r_n$ and $s = s_1 s_2 \ldots s_m$, let D denote a matrix with $n+1$ rows and $m+1$ columns, and $D(i, j)$ be the edit distance between the prefix $r_1 r_2 \ldots r_i$ and the prefix $s_1 s_2 \ldots s_j$. We use the dynamic-programming algorithm to compute the matrix: $D(0, j) = j$ for $0 \le j \le n$, and $D(i, j) = \min(D(i-1, j)+1, D(i, j-1)+1, D(i-1, j-1) + \theta)$ where $\theta = 0$ if $r_i = s_j$; otherwise $\theta = 1$. $D(i, j)$ is called an *active entry* if $D(i, j) \le \tau$. 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 $\tau = 1$. (For all running examples in the remainder of this paper, we assume $\tau = 1$.) To check whether r = "ebay" and s = "koby" are similar, we first compute the entries in row $D(0, *)$ (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 $D(1, *)$. Similarly, we compute the entries in row $D(2, *)$. 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 $D(i > 2, *)$ 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.

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,

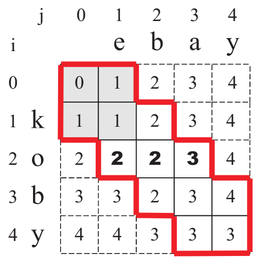
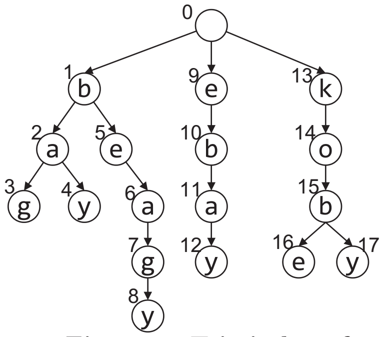


Figure 1: Prefix pruning. Matrix for computing edit distance of two strings "ebay" and "koby". Shaded cells denote *active entries* for $\tau = 1$.

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 "ko" refer to node 14, and node 14 also refers to string "ko". Given a trie node n, let $|n|$ denote its depth (the depth of the root node is 0). For example, $|\text{"ko"}| = 2$.



A sample data set

SID	String
s_1	bag
s_2	ebay
s_3	bay
s_4	kobe
s_5	koby
s_6	beagy

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 $\text{ED}(s, n) \le \tau$. 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 $D(|n|, *)$ 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 subtree rooted at "ko" cannot be similar to "ebay" based on LEMMA 1, 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 $\mathcal{R}$, and then for each string $s \in \mathcal{S}$, computes the active-node set $\mathcal{A}_s$ of s based on subtrie pruning. We can also use the incremental algorithm [9] to compute the active-node sets. For each $r \in \mathcal{A}_s$, if r is a leaf node (i.e., $r \in \mathcal{S}$), $\langle s, r\rangle$ is a similar string pair. For example, in Figure 2, given a string s = "ebay", $\mathcal{A}_{\text{"ebay"}} = \{4, 11, 12\}$. As node 4 ("bay") is a leaf node, $\langle$"ebay", "bay"$\rangle$ is a similar string pair.

Observation 2 - Dual Subtrie Pruning: Subtrie punning only utilizes the trie structure to index strings in $\mathcal{R}$. In fact, the strings in $\mathcal{S}$ also share prefixes, and we can do subtrie pruning for the strings in $\mathcal{S}$. To this end, we construct a trie for stings in both $\mathcal{R}$ and $\mathcal{S}$[1], and use the trie to do subtrie pruning for strings in both of the two sets. For example, in Figure 3, based on subtrie pruning, all the nodes in the subtrie rooted at "ko" can be pruned for the string "ebay" in $\mathcal{S}$. In terms of the similarity-join problem, there are a collection of strings with prefix "eb" in $\mathcal{S}$, and all such strings cannot be similar to strings with prefix "ko". Thus we can prune the two subtries rooted at "eb" and "ko".

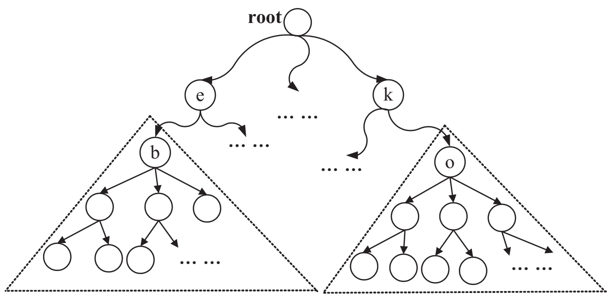


Figure 3: Dual subtrie pruning

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 subtries 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 trie 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 $\mathcal{R} = \mathcal{S}$. Our approach can be easily extended to $\mathcal{R} \neq \mathcal{S}$, and Appendix E gives the details.

3.1 Trie-Traverse Algorithm

Recall the trie-search-based algorithm TRIE-SEARCH (Section 2.3), it can only use subtrie punning, 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.

[1]Appendix E gives the details about how to construct a trie structure for two data sets.

Algorithm Description: TRIE-TRAVERSE first constructs a trie index for all strings in $\mathcal{S}$, 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 $s \in \mathcal{A}_l$, if s is a leaf node (i.e., $s \in \mathcal{S}$), TRIE-TRAVERSE outputs $\langle l, s\rangle$ as a similar string pair. Appendix A gives the pseudo-code of the TRIE-TRAVERSE algorithm.

Computing Active-Node Sets: Obviously, for the root node, its active-node set is composed of the nodes with depth smaller than τ. For example, in Figure 4, suppose $\tau = 1$, $\mathcal{A}_0 = \{0, 1, 9, 13\}$. For each of other nodes n, we compute its active-node set $\mathcal{A}_n$ using its parent's active-node set $\mathcal{A}_p$, where p denotes the parent of n. That is for each node in $\mathcal{A}_n$, it must have an ancestor in $\mathcal{A}_p$ 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 $n' \in \mathcal{A}_n$, there must exist a node $p' \in \mathcal{A}_p$, such that p' is an ancestor of n'.*

For example, in Figure 4, consider n = "kob" and its parent node p = "ko". To compute $\mathcal{A}_{\text{"kob"}}$, we only need to verify whether the descendants of nodes in $\mathcal{A}_{\text{"ko"}} = \{13, 14, 15\}$ 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.

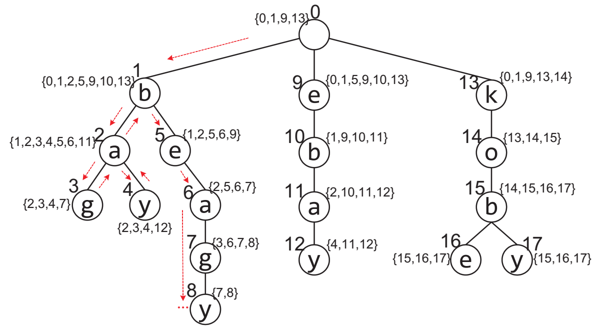


Figure 4: An example to use Trie-Traverse algorithm to find all similar pairs ($\tau = 1$)

Next we discuss how to use $\mathcal{A}_p$ to compute $\mathcal{A}_n$. For each active node p' in $\mathcal{A}_p$, we verify whether each of p''s descendants is an active node of n, by considering the following operations: match, substitution, deletion, and insertion [9]. For example, in Figure 4, node 0 is the parent of node 9, and we compute $\mathcal{A}_9$ based on $\mathcal{A}_0 = \{0, 1, 9, 13\}$ as follows. Consider node 0 in $\mathcal{A}_0$, as we can do a deletion operation on node 0 (deleting e), 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 $\mathcal{A}_9$ = {0, 1, 5, 9, 10, 13}. Similarly $\mathcal{A}_{10}$ = {1, 9, 10, 11}, $\mathcal{A}_{11}$ = {2, 10, 11, 12}, and $\mathcal{A}_{12}$ = {4, 11, 12}.

Note that in the worst case the time complexity of computing $\mathcal{A}_n$ from $\mathcal{A}_p$ is $\mathcal{O}(\tau \cdot |\mathcal{A}_n|)$, since each active node only can be computed from its ancestors within τ steps. Therefore, the time complexity of TRIE-TRAVERSE is $\mathcal{O}(\tau \cdot |\mathcal{A}_T|)$

where $|\mathcal{A}_T|$ 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 $\mathcal{O}(|T|+C_{max})$, where $|T|$ is the size of trie T. Example 1 shows how TRIE-TRAVERSE works.

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 $\mathcal{A}_0 = \{0, 1, 9, 13\}$, which is composed of the nodes with depths within $\tau = 1$, since their edit distances to the root node (an empty string) are within τ. 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 $\mathcal{A}_1$ to compute its active-node set $\mathcal{A}_2$. Similarly, we compute $\mathcal{A}_3$ using $\mathcal{A}_2$. As node 3 is a leaf node, and node 4 is a leaf node in $\mathcal{A}_3 = \{2, 3, 4, 7\}$, thus we output the similar pair $\langle 3, 4\rangle$.*

3.2 Trie-Dynamic Algorithm

TRIE-TRAVERSE has to compute the active-node sets for every trie node. However, we need not compute all of them. For instance, in Figure 4, consider node 3, as it is an active node of node 2 (i.e. $3 \in \mathcal{A}_2$). 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. $2 \in \mathcal{A}_3$). Thus, we can avoid unnecessary computation when computing the active-node set of node 3.

Based on this observation, we design a new algorithm, called TRIE-DYNAMIC, which avoids the redundant active-node computation introduced by TRIE-TRAVERSE. TRIE-DYNAMIC dynamically constructs the trie structure. Initially, TRIE-DYNAMIC constructs an empty trie with only a root node (for empty string), and then incrementally inserts strings into the trie. Given a new string s, for each prefix of s, if the prefix is not in the trie, TRIE-DYNAMIC inserts a new node for the prefix and computes its active-node set on the current trie. Suppose node n is a newly inserted node. For each node $v \in \mathcal{A}_n$, TRIE-DYNAMIC updates $\mathcal{A}_v$ by inserting n into $\mathcal{A}_v$ based on the symmetry property. Finally, as s is a leaf node, for each node $r \in \mathcal{A}_s$, TRIE-DYNAMIC outputs the similar string pair $\langle r, s\rangle$. Appendix B gives the pseudo-code of the TRIE-DYNAMIC algorithm.

As TRIE-DYNAMIC utilizes the symmetry property of active nodes, its time complexity is reduced to $\mathcal{O}(\frac{\tau}{2} \cdot |\mathcal{A}_T|)$. As it needs to keep active nodes of all trie nodes, its space complexity increases to $\mathcal{O}(|T| + |\mathcal{A}_T|)$. Example 2 shows how the TRIE-DYNAMIC algorithm works.

EXAMPLE 2. *Consider the string set in Figure 2, Figure 5 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 5(a), we initialize a trie index with only a root node 0 and its active-node set $\mathcal{A}_0 = \{0\}$. To insert a new string "*`bag`*", as every prefix of "*`bag`*" is not in the trie, we first insert node 1 with label "*`b`*" as a child of node 0 and compute its active-node set $\mathcal{A}_1 = \{0, 1\}$ using $\mathcal{A}_0 = \{0\}$, and update $\mathcal{A}_0$ by inserting node 1 based on the symmetry property of active nodes, i.e,*

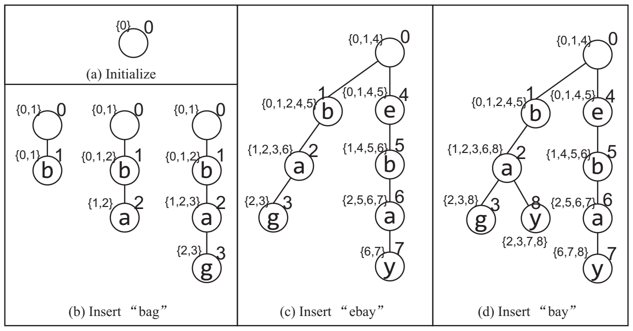


Figure 5: An example to use Trie-Dynamic algorithm to find all similar pairs ($\tau = 1$)

$\mathcal{A}_0 = \{0, 1\}$; then insert node 2 with label "`a`*" as a child of node 1 and compute its active-node set $\mathcal{A}_2 = \{1, 2\}$ using $\mathcal{A}_1 = \{0, 1\}$, and update $\mathcal{A}_1$ by inserting node 2, i.e, $\mathcal{A}_1 = \{0, 1, 2\}$; finally insert node 3 with label "*`g`*" as a child of node 2 and compute its active-node set $\mathcal{A}_3 = \{2, 3\}$ using $\mathcal{A}_2 = \{1, 2\}$, and update $\mathcal{A}_2$ by inserting node 3, i.e, $\mathcal{A}_2 = \{1, 2, 3\}$. Figure 5(b) gives the detailed steps.*

Similarly, we can insert "`ebay`*" (Figure 5(c)). In Figure 5(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 5(d) with Figure 5(c), we find that $\mathcal{A}_2$, $\mathcal{A}_3$, $\mathcal{A}_7$ are different. Because after we insert node 8 and compute $\mathcal{A}_8 = \{2, 3, 7, 8\}$, we update the active-node sets of nodes in $\mathcal{A}_8$(nodes 2, 3, 7). For each node n in $\mathcal{A}_8$, we add node 8 to n's active-node set based on the symmetry property.*

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 5(d), when inserting a string "`bay`", TRIE-DYNAMIC generates a new node 8 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 8. 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.[2] 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. Firstly, 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-

[2]If we first sort the strings and then dynamically insert them into the trie, TRIE-DYNAMIC need not maintain all active-node sets. However it has two problems: 1) it involves an additional sorting step; 2) it is still expensive to update the active-node sets (the symmetry property).

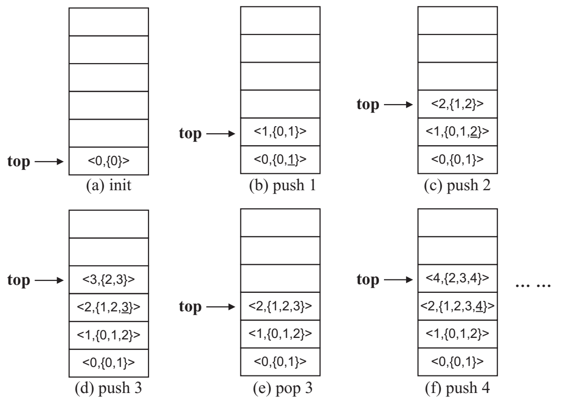


Figure 6: An example to use Trie-PathStack algorithm to find all similar pairs ($\tau = 1$)

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 τ elements (i.e., n's ancestors within τ 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 τ nodes are active nodes of n. Experimental results shows that TRIE-PATHSTACK can avoid a lot of unnecessary update.

Based on the two ideas, we devise the TRIE-PATHSTACK algorithm. TRIE-PATHSTACK first constructs a trie for all strings, and then traverses the trie nodes in preorder. TRIE-PATHSTACK uses a runtime stack to maintain active-node sets of nodes from the root to the current node. When visiting a new node n, TRIE-PATHSTACK first computes its active-node set using the virtual partial trie based on its parent's active-node set (the top element in the runtime stack), and pushes n into the stack. Then TRIE-PATHSTACK updates the active-node sets of n's ancestors within τ steps away from n (the topmost τ elements in the stack). If n is a leaf node, TRIE-PATHSTACK outputs corresponding similar string pairs and pops n from the stack. Appendix C gives the pseudo-code of TRIE-PATHSTACK. Obviously its time complexity is $\mathcal{O}(\frac{\tau}{2} \cdot |\mathcal{A}_T|)$ and space complexity is $\mathcal{O}(|T| + C_{max})$. Example 3 shows how TRIE-PATHSTACK works.

EXAMPLE 3. *Consider the string set and the corresponding trie structure in Figure 2, Figure 6 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 $\mathcal{A}_0 = \{0\}$ into the stack, and get its first child, node 1 (Figure 6(a)). In Figure 6(b), we compute $\mathcal{A}_1 = \{0, 1\}$ using $\mathcal{A}_0 = \{0\}$. Though node 2 is also an active node of node 1, we ignore it since it is unvisited in preorder traversal. We then update the active-node sets of its ancestors by adding node 1 to $\mathcal{A}_0$ (the underlined number). We repeat these steps until visiting node 3 which has no children. We pop node 3 (Figure 6(e)) from the stack and push its sibling node 4 into the stack (Figure 6(f)). 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.*

Appendix D proposes a partition-based method to improve TRIE-PATHSTACK for large edit-distance thresholds.

THEOREM 1. *Given a set of strings $\mathcal{S}$ and an edit-distance threshold τ,* TRIE-TRAVERSE, TRIE-DYNAMIC, *and* TRIE-PATHSTACK *can compute all similar string pairs $\langle s \in \mathcal{S}, t \in \mathcal{S}\rangle$ such that $\text{ED}(s,t) \le \tau$.*

PROOF. Due to space constraints, we omit the proof. Interested readers are referred to [18] for details. □

4. PRUNING TECHNIQUES

This section proposes three techniques to improve performance which can also reduce the sizes of active-node sets.

Length Pruning: Consider two strings r and s, if their length difference is larger than τ, their edit distance cannot be within τ [6]. We exploit this property for pruning in our framework. In Figure 7, in the left box, for each node, we maintain a range of lengths of strings in the subtrie, $[l_s, l_l]$, where l_s is the length of the shortest string in the subtrie and l_l 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 subtries have at least two differences (larger than $\tau = 1$), node v can be pruned from $\mathcal{A}_u$ through length pruning, although node v is an active node of node u.

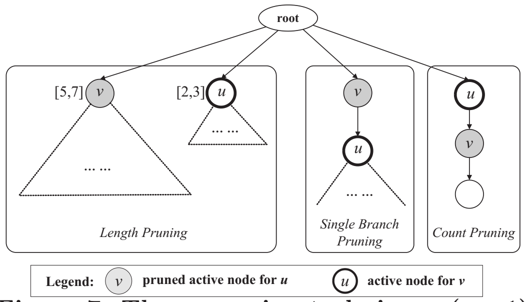


Figure 7: Three pruning techniques ($\tau = 1$)

Single-branch Pruning: If node v is an ancestor node of node u and their subtries have the same leaf nodes, then node v can be pruned from $\mathcal{A}_u$, 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 $\mathcal{A}_u$ to compute the active-node sets of u's children, v will not generate new leaf active nodes, thus we can remove v from $\mathcal{A}_u$. We call this pruning technique *single-branch pruning*. For instance, in the center box of Figure 7, as node v and node u have the same leaf nodes, based on single-branch pruning, v can be pruned from $\mathcal{A}_u$.

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 $\mathcal{A}_u$ because we cannot find *two* strings in their subtries. As an example in the right box of Figure 7, v can be excluded from $\mathcal{A}_u$ 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 $\mathcal{A}_6 = \{2, 5, 6, 7\}$. Using length pruning, we have $\mathcal{A}_6 = \{5, 6, 7\}$. Using single-branch pruning, we have $\mathcal{A}_6 = \{6, 7\}$. Using count pruning, we have $\mathcal{A}_6 = \{\}$. 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. Suppose we have gotten the self-join results of a string set $\mathcal{S}$, and then $\mathcal{S}$ is updated by adding another string set $\Delta\mathcal{S}$, 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 $\mathcal{S}$, a new string set $\Delta\mathcal{S}$, and an edit-distance threshold τ, an incremental similarity join finds all similar string pairs $(r \in \Delta\mathcal{S}, s \in \mathcal{S} \cup \Delta\mathcal{S})$ such that $\text{ED}(r, s) \leq \tau$.*

Due to space constraints, here we only show how to extend TRIE-PATHSTACK algorithm to support incremental similarity joins, and we can easily extend other algorithms to support update. Consider the trie index $\mathcal{T}$ constructed from $\mathcal{S}$. Given a new string set $\Delta\mathcal{S}$, we update the original trie $\mathcal{T}$ to $\mathcal{T}'$ by inserting the strings in $\Delta\mathcal{S}$. In the updated trie $\mathcal{T}'$, let $\Delta\mathcal{T}$ denote the partial trie for strings $\Delta\mathcal{S}$. Then we extend TRIE-PATHSTACK to find similar string pairs for trie nodes in $\Delta\mathcal{T}$ as follows. When reaching a trie node n, different from TRIE-PATHSTACK which computes n's active-node set $\mathcal{A}_n$ from visited nodes, the incremental similarity-join algorithm computes $\mathcal{A}_n$ from the nodes in $\mathcal{T}'$. Appendix F gives the pseudo-code of the incremental similarity-join algorithm. Example 4 shows how the algorithm works.

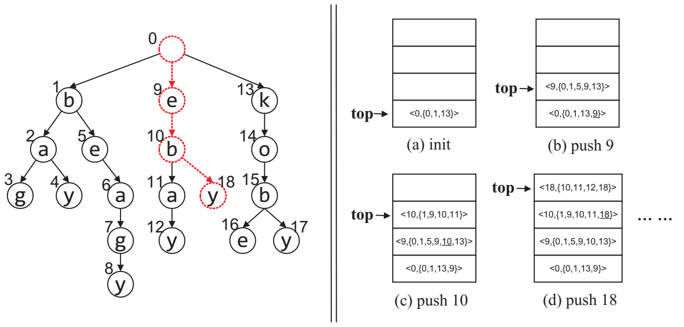


Figure 8: Incremental similarity joins on sample data set in Figure 2 ($\Delta\mathcal{S}$={"eby"}, $\tau = 1$)

EXAMPLE 4. *Consider the trie structure $\mathcal{T}$ in Figure 2. Suppose $\Delta\mathcal{S}$={"*`eby`*"}. Based on our incremental trie-join algorithm, we update the original $\mathcal{T}$ to $\mathcal{T}'$ by inserting "*`eby`*" (Figure 8) and get the partial trie $\Delta\mathcal{T}$ marked by the dot lines. Then we traverse the trie $\Delta\mathcal{T}$ 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 $\mathcal{T}'$. Next we push nodes 9, 10 and 18 into the stack. When reaching leaf node 18 (Figure 8(d)), we output similar string pair (18,12). The algorithm stops when the stack is empty.*

6. 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. We compared our algorithms with state-of-the-art methods, Part-Enum [1], All-Pairs-Ed [2], Ed-Join [19](Appendix G.1). 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. Appendix G.1 gives detailed data-set descriptions and experimental settings.

6.1 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 9(a)-9(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-TRAVERSE does 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 $|\mathcal{A}_n|$ active-node sets, while TRIE-PATHSTACK only updates τ ($\ll |\mathcal{A}_n|$) active-node sets. Throughout the algorithm, TRIE-PATHSTACK only maintains a small portion of active nodes. Appendix G.2 gives the numbers of maintained active nodes for each algorithm.

6.2 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. Table 1 shows the results. In the table, "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 about 25% useless active nodes for the edit-distance threshold $\tau = 2$ and count pruning nearly prunes 50% useless active nodes for $\tau = 1$. In addition, we also compared the running time of employing different pruning techniques. The three pruning techniques can improve the performance beyond TRIE-PATHSTACK by 24.7% when $\tau = 1$ and 14.7% when $\tau = 2$.

Table 1: Numbers of active nodes ($*10^6$) of Trie-PathStack with different pruning techniques (AOL)

τ	No Pruning	Length	Single Branch	Count	All Pruning
1	42.3	39.5	33.2	23.7	20.6
2	230.5	175.1	212.8	203.8	147.7

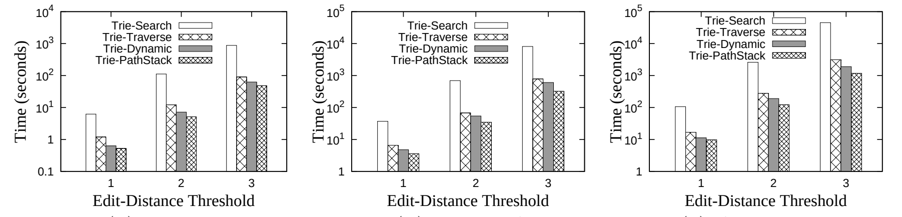
6.3 Comparison with Existing Methods

Index sizes: We compared index sizes with the state-of-the-art methods on three data sets, as shown in Table 2. 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 we used a trie index to share the common prefixes of strings.

Table 2: Index sizes (MB)

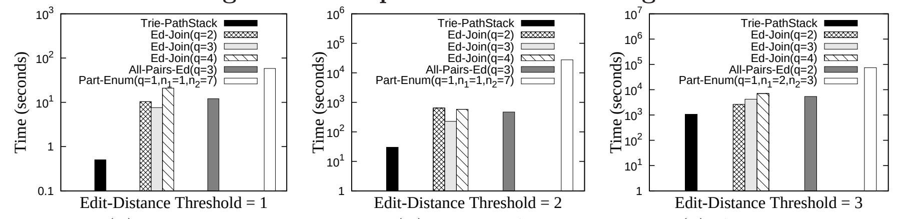
Data Sets	TRIE-PATHSTACK	Part-Enum	All-Pairs-Ed	Ed-Join
Dict	2	16	30	10
DBLP	16	54	155	65
AOL	29	120	305	160

Efficiency: We compared efficiency of the four algorithms on the three data sets. As the performance of state-of-the-art methods highly depends on parameters settings, it took considerable time for tuning parameters to optimize their runtime for each experiment. Figure 10 depicts the results.



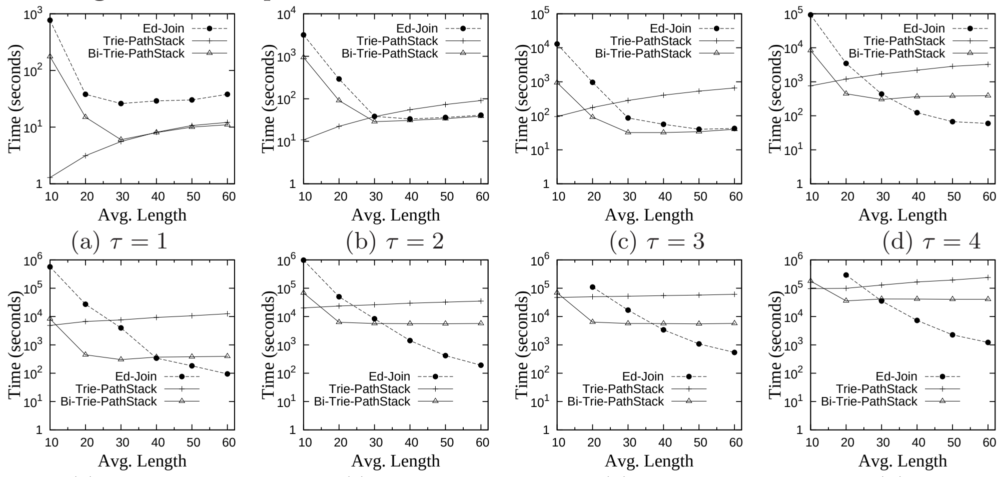
(a) English Dict (b) DBLP Author (c) AOL Query Log

Figure 9: Comparison of the four algorithms



(a) English Dict (b) DBLP Author (c) AOL Query Log

Figure 10: Comparison with state-of-the-art methods on three datasets



(a) $\tau = 1$ (b) $\tau = 2$ (c) $\tau = 3$ (d) $\tau = 4$

(e) $\tau = 5$ (f) $\tau = 6$ (g) $\tau = 7$ (h) $\tau = 8$

Figure 11: Comparison of 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.)

In Figure 10, q is a parameter of gram based methods (the length of a gram), and n_1 and n_2 are two additional parameters for Part-Enum, which denote the numbers of partition and enumeration respectively. Figure 10(a) shows that Trie-PathStack is about 15 times faster than the best existing method Ed-Join ($q = 3$) on the English Dict data set with $\tau = 1$. Figure 10(b) shows that Trie-PathStack outperforms the best existing method Ed-Join ($q = 3$), by an order of magnitude on the DBLP Author data set with $\tau = 2$. In Figure 10(c), on the AOL data set with $\tau = 3$, the best q for Ed-Join was 2 instead of 3 (for the other two data sets). Trie-PathStack took 1000 seconds while Ed-Join ($q = 2$) involved 2600 seconds. Trie-PathStack is always better than Ed-Join on the three datasets with short strings (the average string length is no larger than 30). This is because that Ed-Join can get effective pruning power with large q values. But for short strings, it cannot choose large q values, since such values will destroy the majority or all of grams with only one edit error; while small q values will increase inverted-list sizes and generate many more candidates that need to be further verified.

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 [19], 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 11, 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 Appendix D, called Bi-Trie-PathStack) by varying the edit-distance thresholds from 1 to 8. From Figures 11(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 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 11(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 $\lfloor \frac{\tau}{2} \rfloor \leq 1$, and Trie-PathStack is very efficient for smaller edit-distance thresholds. In addition, Bi-Trie-PathStack generates a smaller number of candidates than Ed-Join and thus achieves higher efficiency.

Figures 11(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

Table 3: Algorithm selection

Avg. Length	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau \in [4, 8]$
(0,20]	TP	TP	TP/Bi-TP	TP/Bi-TP
(20,30]	TP	TP/Bi-TP	Bi-TP	Bi-TP
(30,40]	TP	Bi-TP	Bi-TP	Bi-TP/EJ
(40,60]	Bi-TP	Bi-TP	Bi-TP	EJ

than Bi-Trie-PathStack in some cases. This is because for short strings, both Bi-Trie-PathStack and Ed-Join will verify a large number of candidates, but Trie-PathStack can directly generate all results. For larger thresholds ($\tau \geq 4$) and longer strings (the average string length is larger than 30), as shown in Figures 11(d)-(h), Ed-Join is more efficient than our algorithms since in these cases since 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 3 illustrates how to select a good algorithm based on the results from Figure 11, where TP, Bi-TP, and EJ respectively denote Trie-PathStack, Bi-Trie-PathStack and Ed-Join. We have the following observations. Firstly, for $\tau \leq 3$, our methods outperform Ed-Join. Secondly, for $\tau \in [4, 8]$, both Bi-Trie-PathStack and Ed-Join are effective for the data sets with the average string length within (30,40]. Thirdly, for $\tau \in [4, 8]$, Ed-Join is more effective for the data sets with the average string length within (40,60].

6.4 Additional Experiments

In Appendix G.3, we evaluate our incremental algorithm for update of data sets. We evaluate our algorithms for $\mathcal{R} \neq \mathcal{S}$ in Appendix G.4 and test the scalability in Appendix G.5.

7. RELATED WORK

String similarity joins have been extensively studied [6, 1, 2, 4, 16, 19, 20]. Gravano et al. [6] proposed to use SQL statements for similarity joins in relational databases. Chaudhuri et al. [4] proposed a primitive operator for effective similarity joins. Arasu et al. [1] developed a signature scheme which can be used as a filter for effective similarity joins. Bayardo et al. [2] proposed all-pair similarity joins, a prefix-filtering based algorithm. Xiao et al. [20] proposed ppjoin to improve all-pair algorithm by introducing positional filtering and suffix filtering.

The other related studies are approximate string searching [7, 12, 3] (See Appendix H) 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. 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 [9, 5], 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 $\mathcal{R}$ and $\mathcal{S}$, for each string in $\mathcal{S}$, we find its similar strings from set $\mathcal{R}$. As discussed in Section 2, this method is inefficient as they cannot utilize the fact that the strings in $\mathcal{S}$ also share common prefixes. We propose new effective algorithms and pruning techniques.

8. 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).

9. ACKNOWLEDGEMENT

This work is partly supported by the National Natural Science Foundation of China under Grant No. 60873065, the National High Technology Development 863 Program of China under Grant No. 2009AA011906, and the National Grand Fundamental Research 973 Program of China under Grant No. 2006CB303103.

10. REFERENCES

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.
[2] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.
[3] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, pages 313–324, 2003.
[4] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.
[5] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD Conference*, pages 707–718, 2009.
[6] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.
[7] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, pages 267–276, 2008.
[8] M. Hadjieleftheriou, N. Koudas, and D. Srivastava. Incremental maintenance of length normalized indexes for approximate string matching. In *SIGMOD Conference*, pages 429–440, 2009.
[9] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 371–380, 2009.
[10] M.-S. Kim, K.-Y. Whang, J.-G. Lee, and M.-J. Lee. n-Gram/2L: A space and time efficient two-level n-gram inverted index structure. In *VLDB*, pages 325–336, 2005.
[11] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, 2008.
[12] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.
[13] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.
[14] S. C. Sahinalp, M. Tasan, J. Macker, and Z. M. Özsoyoglu. Distance based indexing for string proximity search. In *ICDE*, pages 125–, 2003.
[15] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. pages 159–165, 1990.
[16] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.
[17] K. U. Schulz and S. Mihov. Fast string correction with levenshtein automata. *IJDAR*, 5(1):67–85, 2002.
[18] J. Wang, J. Feng, and G. Li. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. Technical report, Tsinghua University, 2010. http://dbgroup.cs.tsinghua.edu.cn/technicalreports/triejoin.pdf.
[19] C. Xiao, W. W. 0011, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.
[20] C. Xiao, W. W. 0011, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, 2008.

APPENDIX

A. TRIE-TRAVERSE ALGORITHM

Figure 12 gives the pseudo-code of the TRIE-TRAVERSE algorithm. Different from TRIE-SEARCH, TRIE-TRAVERSE uses dual subtrie pruning to find similar string pairs. It first constructs a trie index for all strings (Line 2), computes the active-node set of the root node (Line 4), and then calls its subroutine `findSimilarPair` to find all similar string pairs recursively (Lines 5-6). `findSimilarPair` first calculates the active-node set $\mathcal{A}_c$ of node c based on its parent's active-note set $\mathcal{A}_p$ (Line 2), using the incremental algorithm introduced in Section 3.1, whose time complexity is $O(\tau \cdot |\mathcal{A}_c|)$. If c is a leaf node, it calls a subroutine `outputSimilarPair` to output all the similar string pairs of c (Line 3). Finally `findSimilarPair` calls itself to compute the similar string pairs of c's descendants (Lines 5-6).

```
Algorithm 1: TRIE-TRAVERSE($\mathcal{S}, \tau$)
  Input: $\mathcal{S}$: a collection of strings
         $\tau$: a given edit-distance threshold
  Output: $\mathcal{P} = \{(s \in \mathcal{S}, t \in \mathcal{S}) \mid \textsc{ed}(s,t) \le \tau\}$
1 begin
2   $\mathcal{T}$ = new Trie($\mathcal{S}$);
3   Let $r$ denote the root of Trie $\mathcal{T}$.
4   $\mathcal{A}_r$= $\{n \mid$ for each trie node $n$, s.t., $|n| \le \tau\}$;
5   for each child node of $r$, $c$ do
6     $\mathcal{P} \cup$ = findSimilarPair($c, r, \mathcal{A}_r$);
7 end
```

```
Function findSimilarPair($c, p, \mathcal{A}_p$)
  Input: $c$: a trie node
         $p$: the parent of $c$
         $\mathcal{A}_p$: the active node set of $p$
  Output: $\mathcal{P}_c = \{(s \in \mathcal{S}, t \in \mathcal{S}) \mid \textsc{ed}(s,t) \le \tau$ and $s$ is a
          leaf descendant of $c\}$
1 begin
2   $\mathcal{A}_c$ = calcActiveNode($c, \mathcal{A}_p$);
3   if $c$ is a leaf node then
4     $\mathcal{P}_c \cup$ = outputSimilarPair($c, \mathcal{A}_c$);
5   for each child node of $c$, $d$ do
6     $\mathcal{P}_c \cup$ = findSimilarPair($d, c, \mathcal{A}_c$);
7 end
```

```
Function outputSimilarPair($n, \mathcal{A}_n$)
  Input: $n$: a trie node
         $\mathcal{A}_n$: $n$'s active-node set
  Output: $\mathcal{P}_n$ =$\{(n, s \in \mathcal{A}_n) \mid \textsc{ed}(n,s) \le \tau\}$
1 begin
2   for each leaf node $l \in \mathcal{A}_n$ $(n \ne l)$ do
3     $\mathcal{P}_n \cup = \{(n,l)\}$;
4 end
```

Figure 12: Trie-Traverse algorithm

B. TRIE-DYNAMIC ALGORITHM

Figure 13 gives the pseudo-code of the TRIE-DYNAMIC algorithm. Different from TRIE-TRAVERSE, TRIE-DYNAMIC avoids unnecessary computation of active-node sets by utilizing the symmetry property of active nodes. 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 $s = s_1 s_2 \ldots s_m$, TRIE-DYNAMIC inserts it into the trie structure as follows (Lines 4-10). First TRIE-DYNAMIC finds the trie node $t = s_1 s_2 \ldots s_i$ which is the longest prefix of s. Then TRIE-DYNAMIC updates the trie by adding some new nodes under node t (Line 6-7) and computing their corresponding active-node sets (Line 8). 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 9-10). Finally, as t is a leaf node (i.e., s), TRIE-DYNAMIC outputs the similar pairs (Line 12).

```
Algorithm 2: TRIE-DYNAMIC ($\mathcal{S}, \tau$)
   Input: $\mathcal{S}$: a collection of strings
          $\tau$: a given edit-distance threshold
   Output: $\mathcal{P} = \{(s \in \mathcal{S}, t \in \mathcal{S}) \mid \textsc{ed}(s,t) \le \tau\}$
 1 begin
 2   $T$ = new $Trie$();
 3   for each $s = s_1 s_2 \ldots s_m$ in $\mathcal{S}$ do
 4     Find trie node $t = s_1 s_2 ... s_i$ which is the longest
       prefix of $s$ ;
 5     for $j$=$i$+1 to $m$ do
 6       $c$ = new $Node(s_j)$;
 7       Append a new child node $c$ to node $t$;
 8       $\mathcal{A}_c$ = calcActiveNode($c, \mathcal{A}_t$);
         /* update active nodes */
 9       for each node $a \in \mathcal{A}_c$ $(c \ne a)$ do
10         add $c$ to $\mathcal{A}_a$;
11       $t = c$;
12     $\mathcal{P} \cup$ = outputSimilarPair($t, \mathcal{A}_t$); /* $t$ is a leaf
       node */
13 end
```

Figure 13: Trie-Dynamic algorithm

C. TRIE-PATHSTACK ALGORITHM

Figure 14 shows the pseudo-code of the TRIE-PATHSTACK algorithm. Different from TRIE-TRAVERSE and TRIE-DYNAMIC, TRIE-PATHSTACK can achieve high performance with less memory. Initially, TRIE-PATHSTACK constructs a trie structure $\mathcal{T}$ for all strings (Line 2). 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 4). Accordingly, in this partial trie we define the active-node set of a node u as $\mathcal{A}_u^{'} = \{v | v \in \mathcal{A}_u,\ v$ has been visited$\}$ and we can get $\mathcal{A}_r^{'} = \{r\}$ (Line 5). Throughout the TRIE-PATHSTACK, we use a stack $\mathbb{S}$ 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 $\mathcal{A}_c{}'$ based on its parent's active-node set $\mathcal{A}_p{}'$ by calling subroutine `calcActiveNode`$'$[3] (Line 12), and then update active-node sets affected by c (Line 13-15). In TRIE-DYNAMIC, for each active node a in $\mathcal{A}_c^{'}$, we update $\mathcal{A}_a^{'}$ by adding the node c . It needs to update $|\mathcal{A}_c^{'}|$ active-node sets. But in TRIE-PATHSTACK, we only need to update the

[3]Note that `calcActiveNode`$'$ only returns visited nodes.

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

```
Input: $\mathcal{S}$: a collection of strings
       $\tau$: a given edit-distance threshold
Output: $\mathcal{P} = \{(s \in \mathcal{S}, t \in \mathcal{S}) \mid \text{ED}(s,t) \le \tau\}$
begin
    $\mathcal{T}$ = new Trie($\mathcal{S}$);
    $\mathbb{S}$ = new Stack();
    Let $r$ denote the root of Trie $\mathcal{T}$ and set $r$ visited.
    $\mathcal{A}_r^{'} = \{r\}$;
    $\mathbb{S}$.push($\langle r, \mathcal{A}_r{'} \rangle$);
    $c = r$.firstchild;
    while not $\mathbb{S}$.empty() do
        while $c$ is not null do
            $\langle p, \mathcal{A}_p{'} \rangle = \mathbb{S}$.top();
            Set $c$ visited.
            $\mathcal{A}_c{'}$ = calcActiveNode$'(c, \mathcal{A}_p{'})$;
            /* update active nodes */
            for each ancestor node of $c$, $t$ do
                if $|c| - |t| \le \tau$ then
                    add $c$ to $\mathcal{A}_t$;
            if $c$ is a leaf node then
                $\mathcal{P} \cup$= outputSimilarPair($c, \mathcal{A}_c{'}$);
            $\mathbb{S}$.push($\langle c, \mathcal{A}_c{'} \rangle$);
            $c = c$.firstchild;
        $\langle p, \mathcal{A}_p{'} \rangle = \mathbb{S}$.pop();
        $c = p$.nextsibling;
end
```

Figure 14: Trie-PathStack algorithm

active-node sets of n's ancestors within τ steps away from n, which is the topmost τ elements in the stack. This is because for other active nodes (preceding nodes but not ancestors of c), they cannot be parent nodes for subsequent nodes, so we will not use their active-node sets. Therefore Trie-PathStack significantly decreases the number of updated active-node sets from $|\mathcal{A}_c^{'}|$ to τ and performs more efficient than Trie-Dynamic, although they both take into account the symmetry property of active nodes.

D. IMPROVING TRIE-PATHSTACK ON LARGE EDIT-DISTANCE THRESHOLDS

In this section, we improve Trie-PathStack to support large edit-distance thresholds. Consider a string $r = r_1 r_2 \dots r_n$. We divide r into two parts $r_1 r_2 \dots r_{\lfloor \frac{n}{2} \rfloor}$ and $r_{\lfloor \frac{n}{2}+1 \rfloor} \dots r_n$. Note that for a string s, if r is similar to s within edit-distance threshold τ ($\tau \le n$), then at least one of the following condition is correct: 1) the first part of r, $r_1 r_2 \dots r_{\lfloor \frac{n}{2} \rfloor}$ is similar to a prefix of s within $\lfloor \frac{\tau}{2} \rfloor$; 2) the second part of r, $r_{\lfloor \frac{n}{2}+1 \rfloor} \dots r_n$ is similar to a suffix of s within $\lfloor \frac{\tau}{2} \rfloor$. 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 $\mathcal{S}$ and a threshold τ, we first discuss how to use Trie-PathStack to find all the string pairs $\langle r, s \rangle \in \mathcal{S} \times \mathcal{S}$ such that the first part of r is similar to a prefix of s within $\lfloor \frac{\tau}{2} \rfloor$. We can construct a new string set $\mathcal{S}^1$ that consists of the first part of each string in $\mathcal{S}$. Then we run the Trie-PathStack on $\mathcal{S}^1$ and $\mathcal{S}$ with edit-distance threshold $\lfloor \frac{\tau}{2} \rfloor$. For a string c in $\mathcal{S}^1$, to find all the strings in $\mathcal{S}$ whose prefix is similar to c, we traverse the subtrie rooted at the active nodes in $\mathcal{A}_c$ and get the leaf nodes. Clearly, these leaf nodes have a prefix that is similar to c. Similarly, if we reverse the strings in $\mathcal{S}$, we can get all the string pairs $\langle r, s \rangle \in \mathcal{S} \times \mathcal{S}$ such that the second part of r is similar to a suffix of s within $\lfloor \frac{\tau}{2} \rfloor$. Based on the candidates generated from the two cases, we verify them to generate final results.

E. SIMILARITY JOINS BETWEEN TWO DIFFERENT SETS

In this section, we discuss how to extend our algorithm to support similarity joins between two different sets $\mathcal{R}$ and $\mathcal{S}$. For ease of presentation, we first introduce a concept.

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

For example, in the left of Figure 16, given the trie index and $\mathcal{S}$ = {`bag`, `beagy`}, node "`be`" belongs to $\mathcal{S}$, since there exists a string s = "`beagy`" with a prefix "`be`".

Algorithm 4: Trie-PathStack$^+$ $(\mathcal{R}, \mathcal{S}, \tau)$

```
Input: $\mathcal{R}$, $\mathcal{S}$: two collections of strings
       $\tau$: a given edit-distance threshold
Output: $\mathcal{P} = \{ (s \in \Delta\mathcal{S}, t \in \mathcal{S} \cup \Delta\mathcal{S}) \mid \text{ED}(s,t) \le \tau\}$
begin
    $\mathcal{T}$ = new Trie($\mathcal{R} \cup \mathcal{S}$);
    $\mathbb{S}$ = new Stack();
    Let $r$ denote the root of Trie $\mathcal{T}$.
    $\mathcal{A}_r{''} = \{n \mid$ for each trie node $n$, s.t., $|n| \le \tau$ and
    $n \in \mathcal{S}\}$;
    $\mathbb{S}$.push($\langle r, \mathcal{A}_r{''} \rangle$);
    $c = r$.firstchild;
    while not $\mathbb{S}$.empty() do
        while $c$ is not null and c $\in \mathcal{R}$ do
            $\langle p, \mathcal{A}_p{''} \rangle = \mathbb{S}$.top();
            $\mathcal{A}_c{''}$ = calcActiveNode$''(c, \mathcal{A}_p{''})$;
            if $c$ is a leaf node then
                $\mathcal{P} \cup$= outputSimilarPair($c, \mathcal{A}_c{''}$);
            $\mathbb{S}$.push($\langle c, \mathcal{A}_c{''} \rangle$);
            $c = c$.firstchild;
        if c is not null then
            $c = c$.nextsibling;
        else
            $\langle p, \mathcal{A}_p{''} \rangle = \mathbb{S}$.pop();
            $c = p$.nextsibling;
end
```

Figure 15: Trie-PathStack$^+$: a similarity-join algorithm for two different sets

We take Trie-PathStack as an example to introduce our idea and propose an algorithm, called Trie-PathStack$^+$ as shown in Figure 15. Different from Trie-PathStack algorithm, Trie-PathStack$^+$ builds a trie index on stings in $\mathcal{R} \cup \mathcal{S}$ (Line 2) and for each node belonging to $\mathcal{R}$, computes its active-node set composed of nodes belonging to $\mathcal{S}$. The active-node set of node r is defined as $\mathcal{A}_r{''} = \{n|$ for each trie node n, such that $|n| \leq \tau$ and $n \in \mathcal{S}\}$ (Line 5), and `calcActiveNode''` returns those active nodes that belong to $\mathcal{S}$. We restrict that only nodes $u \in \mathcal{R}$ can be pushed into the stack (Line 9).

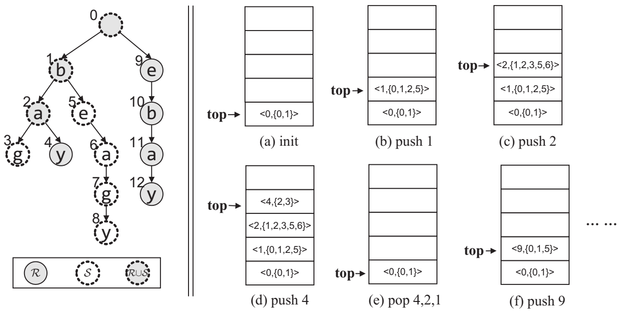


Figure 16: Similarity joins between $\mathcal{R} = \{\texttt{bay}, \texttt{ebay}\}$ and $\mathcal{S} = \{\texttt{bag}, \texttt{beagy}\}$ ($\tau = 1$). We push nodes in $\mathcal{R}$ into the stack and find their active nodes in $\mathcal{S}$.

EXAMPLE 5. *In Figure 16, we illustrate an example to join two different string sets. On the left, it is the trie index for strings in $\mathcal{R} = \{\texttt{bay}, \texttt{ebay}\}$ and $\mathcal{S} = \{\texttt{bag}, \texttt{beagy}\}$. Each node is marked by belonging to which set, such as $\mathcal{R}$, $\mathcal{S}$ or $\mathcal{R} \cup \mathcal{S}$. In Figure 16(a), the stack is initialized with node 0 and $\mathcal{A}_0'' = \{0, 1\}$. Though node 9 is similar to node 0, it is excluded from the set since node $9 \notin \mathcal{S}$. After pushing node 2 into the stack (Figure 16(c)), we then push node 4 into the stack, but will not push node 3 as node $3 \notin \mathcal{R}$. In Figure 16(d), as node 4 is a leaf node, we output similar string pair $(4, 3)$ by finding the leaf node in $\mathcal{A}_4'' = \{2, 3\}$. We continue these steps until the stack is empty.*

F. INCREMENTAL SIMILARITY JOINS

We extend TRIE-PATHSTACK to support incremental similarity joins and Figure 17 shows the pseudo-code. Firstly, we update the original trie $\mathcal{T}$ by inserting all strings in $\Delta\mathcal{S}$ and set the nodes in $\Delta\mathcal{T}$ "unvisited". Secondly, we initialize $\mathcal{A}_r'$ 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.

```
Algorithm 5: INCREMENTALTRIEJOIN($\mathcal{T}, \Delta\mathcal{S}, \tau$)
  Input: $\mathcal{T}$: a trie index of original collection of strings
         $\Delta\mathcal{S}$: a new added collection of strings
         $\tau$: a given edit-distance threshold
  Output: $\mathcal{P} = \{(s \in \Delta\mathcal{S}, t \in \mathcal{S} \cup \Delta\mathcal{S}) \mid \textsc{ed}(s,t) \leq \tau\}$
1 Change Line 2 in TRIE-PATHSTACK algorithm to
  "$\mathcal{T}$.update($\Delta\mathcal{S}$)"(Insert new added strings into the
  trie);
2 Change Line 5 in TRIE-PATHSTACK algorithm to
  "$\mathcal{A}_r'$= {$n$| for each visited trie node $n$, s.t., $|n| \leq \tau$}";
3 Change Line 9 in TRIE-PATHSTACK algorithm to
  "while $c$ is not null and $c$ is not visited";
4 Change Line 20- 21 in TRIE-PATHSTACK algorithm to
  "if $c$ is not null then
      $c = c$.nextsibling;
  else
      $\langle p, \mathcal{A}_p' \rangle = \mathbb{S}$.pop();
      $c = p$.nextsibling;"
```

Figure 17: Incremental trie-join algorithm

G. ADDITIONAL EXPERIMENTS

G.1 Experiment Setup

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[4]. 3) AOL Query Log[5]. We randomly chose one million distinct queries. Table 4 illustrates detailed statistical information of the three data sets. Figures 18(a)-18(c) show their length distribution respectively.

Table 4: Dataset statistics

Data Sets	Sizes	*avg_len*	*max_len*	*min_len*	$\lvert\Sigma\rvert$
English Dict	146,033	8.77	30	1	27
DBLP Author	613,542	12.82	46	4	37
AOL Query Log	1,000,000	20.94	500	1	37

Implementation of existing algorithms:

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 $q\tau + 1$ 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 [19] 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 [1] takes the q-gram set of a string as a feature vector. For two strings, if their edit distance is within τ, 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 Joins" project site[6]. For Part-Enum, we modified the implementation in Flamingo Project[7] to support string similarity joins with edit-distance constrains.

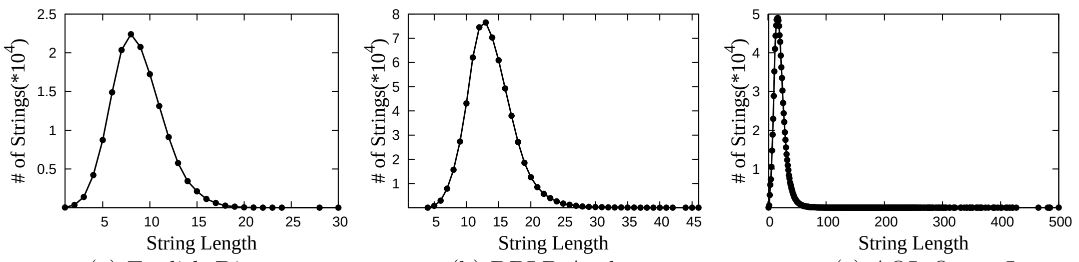
G.2 Comparison of Numbers of Maintained Active Nodes

Table 5: Maximal number of active nodes on AOL

τ	TRIE-SEARCH,TRIE-TRAVERSE	TRIE-DYNAMIC	TRIE-PATHSTACK
1	2444	42346799	**2172**
2	31374	230511829	**18477**
3	257896	2444928000	**201825**

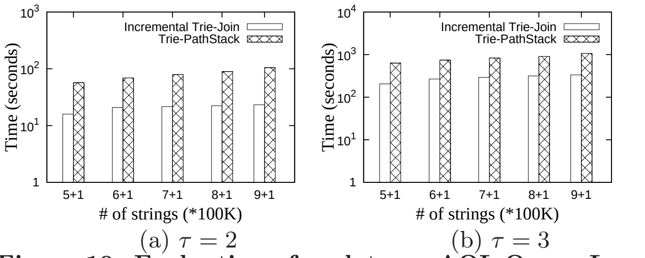
Table 5 illustrates the maximal number of active nodes that four algorithms need to store. We can see that TRIE-DYNAMIC keeps a rather large number of active nodes, since it needs to maintain the active-node sets of all trie nodes. For the other algorithms, the maximal number of active-node sets is the 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-TRAVERSE, since TRIE-PATHSTACK utilizes the symmetry property of two active nodes.

[4]http://www.informatik.uni-trier.de/~ley/db
[5]http://www.gregsadetsky.com/aol-data/
[6]http://www.cse.unsw.edu.au/~weiw/project/simjoin.html
[7]http://flamingo.ics.uci.edu/



(a) English Dict (b) DBLP Author (c) AOL Query Log

Figure 18: String length distribution



(a) $\tau = 2$ (b) $\tau = 3$

Figure 19: Evaluation of update on AOL Query Log (e.g. 6+1 denotes $|\mathcal{S}| = 600K$, $|\Delta\mathcal{S}| = 100K$)

G.3 Evaluation of Update

In this section, we evaluate updates on AOL data set. Initially, we selected 500K strings ($\mathcal{S}$), and for each time, we updated it by inserting 100K strings ($\Delta\mathcal{S}$). 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 19 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 19(a), the speed-up for $|\mathcal{S}|$ = 500K is 3.5 and that for $|\mathcal{S}|$ = 900K is 4.5.

G.4 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 20. Suppose we push nodes in $\mathcal{R}$ into the stack and traverse the trie to find active nodes in $\mathcal{S}$. We can see that it is better to assign $\mathcal{R}$ 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.

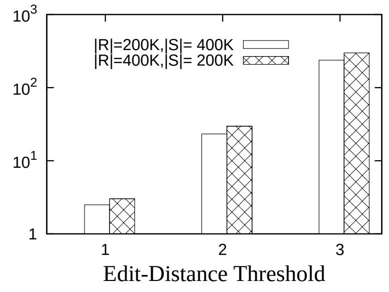


Figure 20: Evaluation of joining two different data sets on DBLP Author

G.5 Scalability

We evaluated the scalability of TRIE-PATHSTACK and incremental trie-join algorithm on AOL Query Log. Initially, the data set was empty, and we inserted 100K strings at each time. We compared the running time of the two algorithms and Figure 21 shows the experimental results with the increase of data sets. We observe that our incremental algorithm scales better than TRIE-PATHSTACK. For example, for 100K strings, both TRIE-PATHSTACK and incremental trie-join algorithm took 6.88s ($\tau = 2$); For 1 million strings, TRIE-PATHSTACK increased to 104.65s while incremental trie-join algorithm only took 23s ($\tau = 2$).

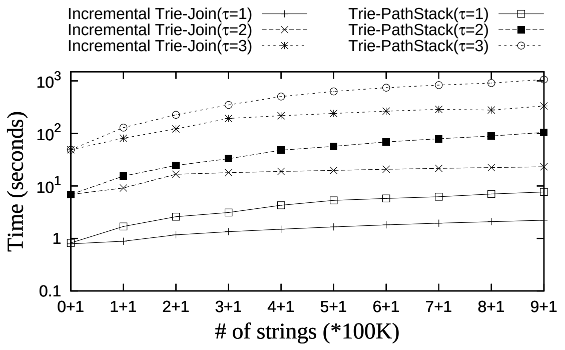


Figure 21: Scalability on AOL Query Log(e.g. 6+1 denotes $|\mathcal{S}| = 600K$, $|\Delta\mathcal{S}| = 100K$)

We also evaluated BI-TRIE-PATHSTACK (Appendix D) 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 6.3). We set $\tau = 8$. 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.

H. RELATED WORK

There have been many studies on approximate string search [7, 12, 8, 3]. Schulz and Mihov [17] proposed an automaton-based approach to address this problem. Sahinalp et al. [14] proposed an index structure called "VP-tree" for answering NN queries in terms of an edit-distance function. Kim et al. [10] 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. [11] developed several list-merging algorithms to improve search efficiency by skipping elements on q-gram inverted lists.