The VLDB Journal (2012) 21:437–461
DOI 10.1007/s00778-011-0252-8

REGULAR PAPER

Trie-join: a trie-based method for efficient string similarity joins

Jianhua Feng · Jiannan Wang · Guoliang Li

Received: 24 January 2011 / Revised: 20 June 2011 / Accepted: 25 August 2011 / Published online: 4 October 2011

Abstract A string similarity join finds similar pairs between two collections of strings. Many applications, e.g., data integration and cleaning, can significantly benefit from an efficient string-similarity-join algorithm. In this paper, we study string similarity joins with edit-distance constraints. Existing methods usually employ a *filter-and-refine* framework and suffer from the following limitations: (1) They are inefficient for the data sets with short strings (the average string length is not 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 method 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 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. We conducted extensive experiments on four real data sets. Experimental results show that our algorithms outperform state-of-the-art methods by an order of magnitude on the data sets with short strings.

Keywords String similarity joins · Data integration and cleaning · Edit distance · Trie index · Subtrie pruning

J. Feng · J. Wang (✉) · G. Li
Department of Computer Science and Technology,
Tsinghua University, Beijing 100084, China
e-mail: wjn08@mails.tsinghua.edu.cn

J. Feng
e-mail: fengjh@tsinghua.edu.cn

G. Li
e-mail: liguoliang@tsinghua.edu.cn

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 [13] proposed by Microsoft has been used in the data debugger project [12].

In this paper, we study string similarity joins with edit-distance constraints, which, given two sets of strings, find all similar string pairs from the two sets, such that the edit distance between each string pair is within a given threshold. The string similarity join can be used to find near duplicated queries in query log mining and correlate two sets of data (e.g., people name, place name, address).

Existing studies to address this problem, such as Part-Enum [5], All-Pairs-Ed [7], Ed-Join [56], 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 not larger than 30 based on the experimental results in Sect. 6), since they cannot select high-quality signatures for short strings, and thus, they may generate a large number of candidate pairs that need to be further verified. Secondly, they cannot support dynamic update of data sets efficiently. For example, in order to achieve a high performance, Ed-Join and All-Pairs-Ed need to select signatures with high weights. They typically use inverse document frequency (IDF) as the weights of signatures. Obviously, the dynamic update may change the weights. Thus, the two methods need to reselect signatures, rebuild indexes, and rerun their algorithms from scratch. Thirdly, they involve large index sizes as they will generate 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 (Sect. 2). We use a trie structure to index strings, which needs much smaller space than existing methods, since the trie structure can share many common prefixes of strings. A straightforward algorithm utilizing trie structures to do similarity joins first constructs a trie structure for one data set and then for each string in the other data set checks whether the string is similar to a string in the trie structure. Note that if the string are not *similar enough* to a trie node, the string will not be similar to those strings under the trie node. Based on this observation, we propose a subtrie pruning technique. To further improve the performance, we develop a dual substring pruning technique, which guarantees that if two trie nodes are not *similar enough*, the string pairs under the two nodes will not be similar. In this way, we can prune large numbers of string pairs.

To efficiently check whether two nodes are similar enough, we propose several trie-based algorithms (Sect. 3). Our algorithms only need to access each node once, which can avoid many unnecessary computations. To avoid considering node pairs repeatedly, we propose efficient algorithms to dynamically construct the trie structures and compute the similar pairs on the fly. To further improve the performance, we also develop three effective pruning techniques based on length filtering and count filtering.

We discuss how to support dynamic update of data sets. We first set the trie nodes for the original data set as *visited* and then append the trie nodes of the update data set as *unvisited*. Finally, we find similar string pairs by only accessing the unvisited nodes (Sect. 4). We also extend our algorithms to do similarity joins on two different sets by building two trie structures.

To support large edit-distance thresholds efficiently, we propose to partition strings into two parts, the left-half part and the right-half part. We prove that if two strings are similar, their left-half parts or right-half parts must be *similar enough*. Based on this observation, we build a trie structure for the left-half parts and another trie structure for the right-half parts. We extend our trie-based algorithms to find the candidate pairs on the two trie structures with a smaller threshold (a half of the original threshold). Finally, we verify the candidate pairs to get the final results (Sect. 5).

We have constructed an extensive set of experiments on four real data sets. Experimental results on real data sets show that our approach can improve the performance by an order of magnitude (Sect. 6.2).

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, ED(`koby`, `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 a specified edit-distance threshold τ, a similarity join finds all string pairs $\langle r, s\rangle \in \mathcal{R} \times \mathcal{S}$ such that $\text{ED}(r, s) \leq \tau$, i.e., $\{\langle r, s\rangle \mid \text{ED}(r, s) \leq \tau, r \in \mathcal{R}, s \in \mathcal{S}\}$.

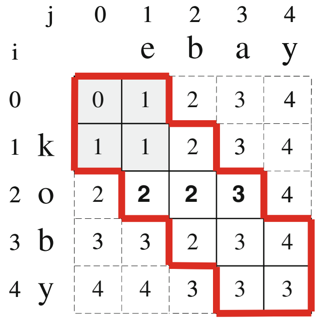
2.2 Prefix pruning

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 [46].

Given two strings $r = r_1r_2 \ldots r_n$ and $s = s_1s_2 \ldots s_m$, let D denote a matrix with $n + 1$ rows and $m + 1$ columns, and $D(i, j)$ be the edit distance between the prefix $r_1r_2 \ldots r_i$ and the prefix $s_1s_2 \ldots s_j$. We use the dynamic-programming algorithm to compute the matrix: $D(0, j) = j$ for $0 \leq j \leq n$, and

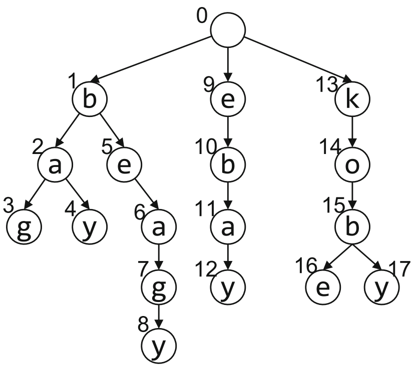
$$D(i, j) = \min\big(D(i-1, j) + 1,\\ D(i, j-1) + 1,\ D(i-1, j-1) + \theta\big), \quad (1)$$

where $\theta = 0$ if $r_i = s_j$; otherwise, $\theta = 1$. $D(i, j)$ is called an *active entry* if $D(i, j) \leq \tau$. 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 $\tau = 1$. (Unless otherwise specified, 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 enclosed 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



j		0	1	2	3	4
i			e	b	a	y
0		0	1	2	3	4
1	k	1	1	2	3	4
2	o	2	**2**	**2**	**3**	4
3	b	3	3	2	3	4
4	y	4	4	3	3	3

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



A sample data set

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

Fig. 2 Trie index of a sample data set

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: Consider a string set $\mathcal{R}$. Let $|\Sigma|$ denote the number of distinct characters in $\mathcal{R}$. There are at most $|\Sigma|^i$ possible prefixes with i characters in $\mathcal{R}$. Many strings may share common prefixes, especially when there are a large number of strings in $\mathcal{R}$. For example, suppose $|R| = 10^6$ and $|\Sigma| = 26$. On average, there are at least $\frac{|R|}{|\Sigma|} = 38462$ strings sharing one common one-character prefix. Note that this is the analysis of the worst case. In Sect. 6.1, we show that strings in real data sets can share many more common prefixes. Based on this observation, 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, Fig. 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, $|\text{"ko"}| = 2$.

Note that many strings with the 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 $\text{ED}(s,n) \leq \tau$. 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 for any 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, a string s and an edit-distance threshold τ, if node n is not an active node for any prefix of s, then n's descendants will not be similar to s.*

Proof Let u denote a child node of n. We first prove u is not an active node for any prefix of s. Consider an arbitrary prefix of s, $s' = s_1 s_2 \cdots s_{|s'|}$, and the matrix D between s' and u. Based on Eq. 1, the edit distance between s' and u is

$$\begin{aligned} & D(|s'|,|u|) \\ & \quad = \min\big(D(|s'|-1,|u|)+1,\ D(|s'|,|u|-1)+1,\ D(|s'|-1,|u|-1)+\theta\big) \\ & \quad = \min\big(D(|s'|-1,|u|)+1,\ D(|s'|,|n|)+1,\ D(|s'|-1,|n|)+\theta\big) \\ & \quad \geq \min\big(D(|s'|-1,|u|)+1,\ D(|s'|,|n|)+1,\ D(|s'|-1,|n|)\big). \qquad (2) \end{aligned}$$

To prove u is not an active node for s', i.e., $D(|s'|,|u|) > \tau$, we only need to prove the right-hand side (RHS) of Eq. 2 is larger than τ. Consider $D(|s'|-1,|n|)$ that denotes the edit distance between $s_1 s_2 \cdots s_{|s'|-1}$ and string n. One possible transformation from $s_1 s_2 \cdots s_{|s'|-1}$ to string n is to first transform $s_1 s_2 \cdots s_{|s'|-1}$ to u with $D(|s'|-1,|u|)$ edit operations and then delete the last character of u, which needs $D(|s'|-1,|u|)+1$ edit operations in total. Since $D(|s'|-1,|n|)$ denotes the minimum number of edit operations (i.e., insertion, deletion, and substitution) needed to transform $s_1 s_2 \cdots s_{|s'|-1}$ to n, then we have

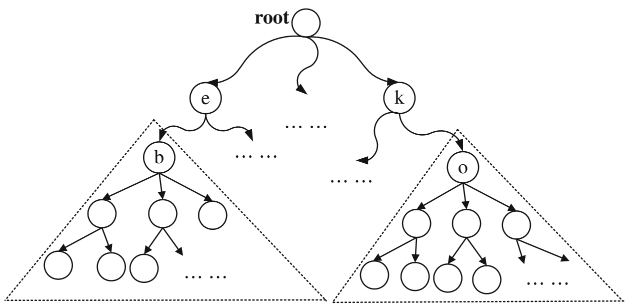


Fig. 3 Dual subtrie pruning

$$D(|s'|-1, |u|) + 1 \geq D(|s'|-1, |n|). \tag{3}$$

Since n is not an active node for every prefix of s, then

$$D(|s'|, |n|) > \tau \quad \text{and} \quad D(|s'|-1, |n|) > \tau. \tag{4}$$

Based on Eqs. 3 and 4, we deduce that the RHS of Eq. 2 is larger than τ, thus $D(|s'|, |u|) > \tau$. As s' could be any prefix of s, u is not an active node for every prefix of s. Using the same idea, we can also prove that child nodes of u are not active nodes for every prefix of s. Accordingly, n's descendants are not active nodes for every prefix of s. Therefore, n's descendants are not similar to s. □

For example, consider the trie in Fig. 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.

Observation 2—dual subtrie pruning: Subtrie pruning 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 the two sets. For example, in Fig. 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”.

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, and an edit-distance threshold τ, 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.*

Proof Consider any two strings s_u and s_v under nodes u and v, respectively. Based on Lemma 1, since u is not an active node for every ancestor of v, u's descendants will not be similar to v. Thus, node v is not an active node for every node in the path from node u to node s_u. Similarly, since v is not an active node for every ancestor of u, v is not an active node for every ancestor of s_u. Based on Lemma 1, v's descendants will not be similar to s_u. As s_v is a descendant of v, s_v is not similar to s_u. Therefore, the strings under u and v cannot be similar to each other. □

For example, in Fig. 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

We first introduce a straightforward trie-search-based method for similarity joins, called Trie-Search, which only utilizes subtrie pruning. Given two string sets $\mathcal{R}$ and $\mathcal{S}$, Trie-Search first constructs a trie structure for all strings in $\mathcal{R}$, and then for each string $s \in \mathcal{S}$, computes its active-node set $\mathcal{A}_s$. For each $r \in \mathcal{A}_s$, if r is a leaf node (i.e., $r \in \mathcal{R}$), $\langle s, r \rangle$ is a similar string pair. For example, in Fig. 2, given a string $s =$ “ebay$''$, $\mathcal{A}_{\text{“}\texttt{ebay}''} = \{4, 11, 12\}$. As node 4 (“bay”) is a leaf node, ⟨“ebay$''$, “bay$''$⟩ is a similar string pair.

We can use the incremental algorithm proposed by Choudri and Kaushik, Ji et al. [14,28] to compute the active-node sets. For a string $s = s_1 s_2 \ldots s_m$, the algorithm first initializes the active-node set of an empty string, which is composed of the nodes with depths no larger than τ in the trie. Then, it computes the active-node set of each prefix of s incrementally. That is the active-node set of $s_1 s_2 \ldots s_i$ is computed using the active-node set of $s_1 s_2 \ldots s_{i-1} (i \in [1, m])$.

However, Trie-Search has a limitation that it neglects strings in $\mathcal{S}$ also share common prefixes; thus, it cannot benefit from dual subtrie pruning. For example, consider two strings ànd ìn $\mathcal{S}$ with a common prefix ˎ To obtain $\mathcal{A}_{\text{“}\texttt{kobe}''}$ and $\mathcal{A}_{\text{“}\texttt{koby}''}$, Trie-Search will suffer from redundant computation since it needs to compute $\mathcal{A}_{\text{“}\texttt{k}''}$, $\mathcal{A}_{\text{“}\texttt{ko}''}$ and $\mathcal{A}_{\text{“}\texttt{kob}''}$ twice. In this section, using dual subtrie pruning, we propose three efficient algorithms and pruning techniques to improve the performance.

[1] Section 4.2 gives the details about how to construct a trie structure for two data sets.

```
Algorithm 1: TRIE-TRAVERSE($\mathcal{S}, \tau$)
   Input: $\mathcal{S}$: a collection of strings
          $\tau$: a given edit-distance threshold
   Output: $\mathcal{P} = \{(s \in \mathcal{S}, t \in \mathcal{S}) \mid \text{ED}(s, t) \le \tau\}$
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: $\mathcal{P}_c = \{(s \in \mathcal{S}, t \in \mathcal{S}) \mid \text{ED}(s, t) \le \tau$ and $s$ is a
           leaf descendant of $c\}$
1  begin
2      $\mathcal{A}_c$ = calcActiveNode($c, \mathcal{A}_p$);
3      Pruning($\mathcal{A}_c$);                    /* See Section 3.4 */
4      if $c$ is a leaf node then
5          $\mathcal{P}_c \cup =$ outputSimilarPair($c, \mathcal{A}_c$);
6      for each child node of $c$, $d$ do
7          $\mathcal{P}_c \cup =$ findSimilarPair($d, c, \mathcal{A}_c$);
8  end
```

```
Function outputSimilarPair($n, \mathcal{A}_n$)
   Input: $n$: a trie node
          $\mathcal{A}_n$: $n$'s active-node set
   Output: $\mathcal{P}_n = \{(n, s \in \mathcal{A}_n) \mid \text{ED}(n, s) \le \tau\}$
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
```

Fig. 4 TRIE-TRAVERSE algorithm

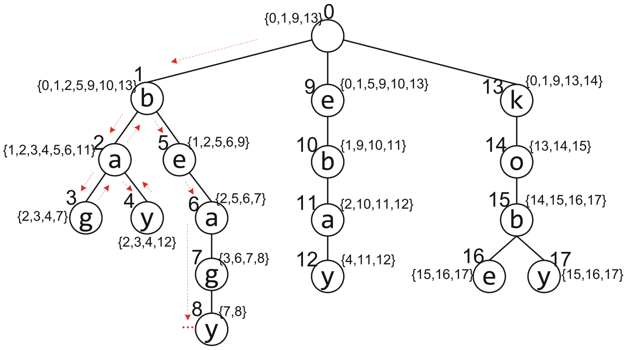


Fig. 5 An example to use TRIE-TRAVERSE algorithm to find all similar pairs ($\tau = 1$)

3.1 Trie-traverse algorithm

In this section, we propose a trie-traversal-based method, called TRIE-TRAVERSE, to improve the performance of TRIE-SEARCH. Intuitively, TRIE-TRAVERSE utilizes dual subtrie pruning to avoid duplicated computation in TRIE-SEARCH. TRIE-TRAVERSE constructs a trie for the strings in both $\mathcal{R}$ and $\mathcal{S}$ and computes the active-node set for a node in the trie exactly once even though the node is a prefix of a potentially large number of strings.

For ease of presentation, in the following, we focus on self-join, that is $\mathcal{R} = \mathcal{S}$. Our approach can be easily extended to $\mathcal{R} \ne \mathcal{S}$ (Sect. 4.2). TRIE-TRAVERSE first constructs a trie index for all strings in $\mathcal{S}$ and then traverses the trie in preorder. 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. 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 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 above-mentioned incremental algorithm. 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 6–7).

In the worst case, the time complexity of computing $\mathcal{A}_c$ from its parent's active-note set $\mathcal{A}_p$ is $\mathcal{O}(\tau \cdot |\mathcal{A}_c|)$, 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 Fig. 5. 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 $\langle 3, 4\rangle$. □

Theorem 1 *Given a set of strings $\mathcal{S}$ and an edit-distance threshold τ,* Trie-Traverse *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 Let $\mathcal{T}$ be a trie index constructed from $\mathcal{S}$ with nodes $V(\mathcal{T})$ and edges $E(\mathcal{T})$. For a node v in $\mathcal{T}$, let $P(v)$ denote v's preorder number and $\mathcal{A}_v = \{u \in V(\mathcal{T}) \mid \text{ED}(u,v) \le \tau\}$ denote the active-node set of v.

We first prove that after visiting a node v in preorder 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 $\mathcal{A}_r = \{u \in V(\mathcal{T}) \mid |u| \le \tau\}$. Assume this claim holds for a node v. We prove that this claim also holds for the next node v' (i.e., $P(v') = P(v)+1$). Based on the preorder 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 $\langle s \in \mathcal{S}, t \in \mathcal{S}\rangle$, $\text{ED}(s,t) \le \tau$ if and only if both the nodes s and t are leaf nodes of $\mathcal{T}$, and s is in $\mathcal{A}_t$. Based on the proved claim, after visiting the node t, we can correctly compute the active-node set $\mathcal{A}_t$. Also note that for each string s in $\mathcal{S}$, its corresponding node in $\mathcal{T}$ must be a leaf node. Therefore, the theorem is proved. □

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 Fig. 5, 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. Intuitively, Trie-Dynamic avoids the redundant active-node computation introduced by Trie-Traverse using symmetric property. 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 $s = s_1 s_2 \ldots s_m$, 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 (Lines 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 (Lines 9–10). Finally, as t is a leaf node (i.e., s), Trie-Dynamic outputs the similar pairs (Line 12).

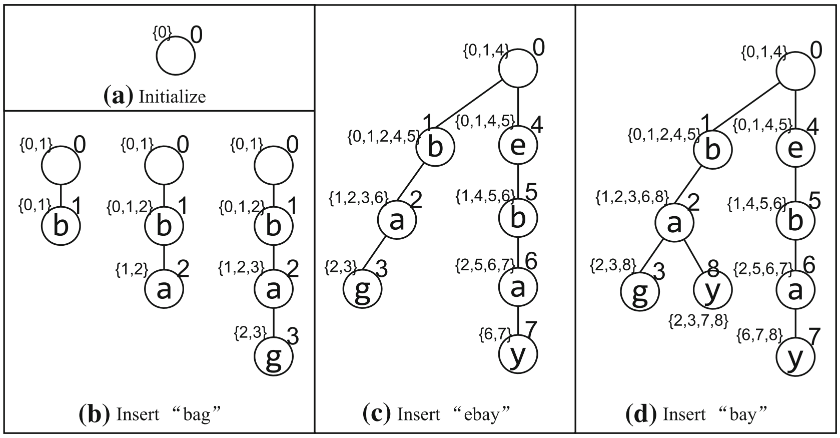
```
Algorithm 2: TRIE-DYNAMIC ($\mathcal{S}, \tau$)
    Input: $\mathcal{S}$: a collection of strings
           $\tau$: a given edit-distance threshold
    Output: $\mathcal{P} = \{(s \in \mathcal{S}, t \in \mathcal{S}) \mid \text{ED}(s,t) \le \tau\}$
 1  begin
 2      $T$ = new $Trie()$;
 3      for each $s = s_1 s_2 \ldots s_m$ in $\mathcal{S}$ do
 4          Find trie node $t = s_1 s_2 \ldots s_i$ 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 $a \in \mathcal{A}_c$ ($c \neq a$) 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
```

Fig. 6 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 Figs. 2, 7 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 Fig. 7a, 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, 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., $\mathcal{A}_0 = \{0, 1\}$; then insert node 2 with label "a" as a child of node 1, 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, 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 7b gives the detailed steps.

Similarly, we can insert "ebay" (Fig. 7c). In Fig. 7d, 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 Fig. 7d with



(a) Initialize

(b) Insert "bag"

(c) Insert "ebay"

(d) Insert "bay"

Fig. 7 An example to use TRIE-DYNAMIC algorithm to find all similar pairs ($\tau = 1$)

Fig. 7c, 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. □

Theorem 2 *Given a set of strings $\mathcal{S}$ and an edit-distance threshold τ,* TRIE-DYNAMIC *can compute all similar string pairs $\langle s \in \mathcal{S}, t \in \mathcal{S}\rangle$ such that $\text{ED}(s, t) \leq \tau$.*

Proof As TRIE-DYNAMIC constructs the trie structure dynamically, let $\mathcal{T}_v$ denote the trie index constructed from all the nodes inserted before v (including itself). For a node v in $\mathcal{T}_v$, let $P(v)$ denote the number of nodes in $\mathcal{T}_v$. $\mathcal{T}_v$'s node set $V(\mathcal{T}_v) = \{u \in V(\mathcal{T}) \mid P(u) \leq P(v)\}$ and its edge set $E(\mathcal{T}_v) = \{(u_i, u_j) \in E(\mathcal{T}) \mid u_i \in V(\mathcal{T}_v), u_j \in V(\mathcal{T}_v)\}$. In addition, let $\mathcal{A}_u^{\mathcal{T}_v} = \{t \in V(\mathcal{T}_v) \mid \text{ED}(u, t) \leq \tau\}$ denote the active-node set of u w.r.t $\mathcal{T}_v$.

We first prove that after adding a new node v in the trie, for each node u of $\mathcal{T}_v$, TRIE-DYNAMIC can compute $\mathcal{A}_u^{\mathcal{T}_v}$ correctly. We prove this claim by induction. The base case for the root node clearly holds since after adding a root node r, $\mathcal{T}_r$ has only one node and $\mathcal{A}_r^{\mathcal{T}_r} = \{r\}$ 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 $\mathcal{T}_{v'}$, if $u = v'$, since the active-node set of u's parent w.r.t $\mathcal{T}_v$ is correctly computed (as u's parent is in $\mathcal{T}_v$), TRIE-DYNAMIC can correctly compute $\mathcal{A}_u^{\mathcal{T}_{v'}}$ (i.e., $\mathcal{A}_{v'}^{\mathcal{T}_{v'}}$); if $u \neq v'$, then u must be a node of $\mathcal{T}_v$, so $\mathcal{A}_u^{\mathcal{T}_v}$ is correctly computed. TRIE-DYNAMIC uses $\mathcal{A}_u^{\mathcal{T}_v}$ to compute $\mathcal{A}_u^{\mathcal{T}_{v'}}$. If u is in $\mathcal{A}_{v'}^{\mathcal{T}_{v'}}$, then $\text{ED}(v', u) \leq \tau$; thus, TRIE-DYNAMIC can obtain $\mathcal{A}_u^{\mathcal{T}_{v'}}$ correctly by adding v' to $\mathcal{A}_u^{\mathcal{T}_v}$; if u is not in $\mathcal{A}_{v'}^{\mathcal{T}_{v'}}$, then $\text{ED}(v', u) > \tau$; thus, TRIE-DYNAMIC can obtain $\mathcal{A}_u^{\mathcal{T}_{v'}}$ correctly by setting it as $\mathcal{A}_u^{\mathcal{T}_v}$. Therefore, for each node u of $\mathcal{T}_{v'}$, TRIE-DYNAMIC can compute $\mathcal{A}_u^{\mathcal{T}_{v'}}$ correctly. Thus, our claim is true.

Obviously for each string pair $\langle s \in \mathcal{S}, t \in \mathcal{S}\rangle$ (Without loss of generality, suppose $P(s) \leq P(t)$.), $\text{ED}(s, t) \leq \tau$ if and only if both the nodes s and t are leaf nodes of $\mathcal{T}_t$, and s is in $\mathcal{A}_t^{\mathcal{T}_t}$. Based on the proved claim, after adding the node t, $\mathcal{A}_t^{\mathcal{T}_t}$ is correctly computed. Also note that for each string s in S and $P(s) \leq P(t)$, its corresponding node in $\mathcal{T}_t$ must be a leaf node. Therefore, the theorem is proved. □

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 Fig. 7d, when inserting a string "bay", TRIE-DYNAMIC generates a new node 8, 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 preorder. 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. Intuitively, TRIE-PATHSTACK can integrate the ideas of TRIE-DYNAMIC and TRIE-TRAVERSE together. To achieve higher

[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).

```
Algorithm 3: TRIE-PATHSTACK ($\mathcal{S}, \tau$)
   Input: $\mathcal{S}$: a collection of strings
          $\tau$: a given edit-distance threshold
   Output: $\mathcal{P} = \{(s \in \mathcal{S}, t \in \mathcal{S}) \mid \text{ED}(s,t) \le \tau\}$
 1 begin
 2     $\mathcal{T}$ = new Trie($\mathcal{S}$);
 3     $\mathbb{S}$ = new Stack();
 4     Let $r$ denote the root of Trie $\mathcal{T}$ and set $r$ as
       "visited".
 5     $\mathcal{A}_r^{'} = \{r\}$;
 6     $\mathbb{S}$.push($\langle r, \mathcal{A}_r{}' \rangle$);
 7     $c = r$.firstchild;
 8     while not $\mathbb{S}$.empty() do
 9         while $c$ is not null do
10             $\langle p, \mathcal{A}_p{}' \rangle = \mathbb{S}$.top();
11             Set $c$ as "visited".
12             $\mathcal{A}_c{}'$ = calcActiveNode'($c, \mathcal{A}_p{}'$);   /* Update
               active nodes */
13             Pruning($\mathcal{A}_c$) ;          /* See Section 3.4 */
14             for each ancestor node of $c$, $t$ do
15                 if $|c| - |t| \le \tau$ then
16                     add $c$ to $\mathcal{A}_t$;
17             if $c$ is a leaf node then
18                 $\mathcal{P} \cup$= outputSimilarPair($c, \mathcal{A}_c{}'$);
19             $\mathbb{S}$.push($\langle c, \mathcal{A}_c{}' \rangle$);
20             $c = c$.firstchild;
21         $\langle p, \mathcal{A}_p{}' \rangle = \mathbb{S}$.pop();
22         $c = p$.nextsibling;
23 end
```

Fig. 8 TRIE-PATHSTACK algorithm

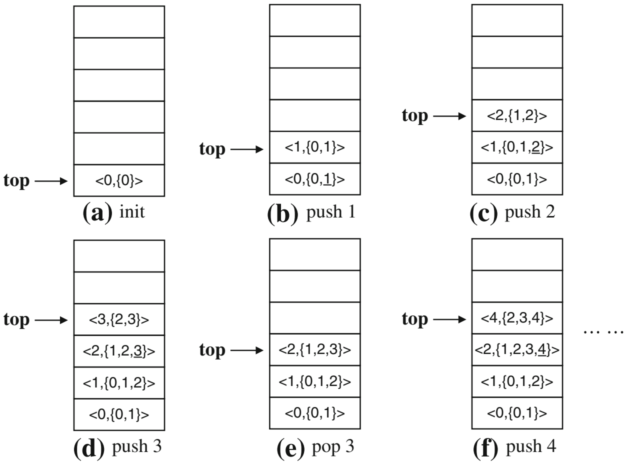


Fig. 9 An example to use TRIE-PATHSTACK algorithm to find all similar pairs ($\tau = 1$)

performance as TRIE-DYNAMIC, when traversing the trie nodes, we maintain a "*virtual partial*" subtrie to keep the visited nodes. For each unvisited node, we first set it as "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. To require little memory space as TRIE-TRAVERSE, 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. This is 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. Figure 8 shows the pseudo-code. 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 as "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 (Lines 14–16). If c is a leaf node, TRIE-PATHSTACK outputs corresponding similar string pairs.

As TRIE-PATHSTACK utilizes the symmetry property of active nodes, its time complexity is the same as TRIE-DYNAMIC, i.e., $\mathcal{O}(\frac{\tau}{2} \cdot |\mathcal{A}_T|)$. As TRIE-PATHSTACK only maintains a stack to store active-node sets whose size is equal to the maximal depth of trie leaf nodes, its space complexity is the same as TRIE-TRAVERSE, i.e., $\mathcal{O}(|T| + C_{\max})$. 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 Figs. 2, 9 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 (Fig. 9a). In Fig. 9b, 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 $\mathcal{A}_0^{'}$

[3] Note that calcActiveNode′ only returns visited nodes.

(the underlined number). We repeat these steps until visiting node 3, which has no children. We pop node 3 (Fig. 9e) from the stack and push its sibling node 4 into the stack (Fig. 9f). We continue to push the first child of node 4 (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 $\mathcal{S}$ and an edit-distance threshold τ,* 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 Let $\mathcal{T}$ be a trie index constructed from $\mathcal{S}$ with nodes $V(\mathcal{T})$ and edges $E(\mathcal{T})$. For a node v in $\mathcal{T}$, let $P(v)$ denote v's preorder number, $v^{(i)}$ denote v's ancestor node with i steps away from v, and $\mathcal{T}_v$ be a partial trie of $\mathcal{T}$ w.r.t v where its node set $V(\mathcal{T}_v) = \{u \in V(\mathcal{T}) \mid P(u) \le P(v)\}$ and its edge set $E(\mathcal{T}_v) = \{(u_i, u_j) \in E(\mathcal{T}) \mid u_i \in V(\mathcal{T}_v), u_j \in V(\mathcal{T}_v)\}$. Let $\mathcal{A}_u^{\mathcal{T}_v} = \{t \in V(\mathcal{T}_v) \mid \text{ED}(u,t) \le \tau\}$ denote the active-node set of u w.r.t $\mathcal{T}_v$.

We first prove that after visiting a node v in the preorder traversal, the current stack $\{\langle v, \mathcal{A}_v^{\mathcal{T}_v}\rangle, \langle v^{(1)}, \mathcal{A}_{v^{(1)}}^{\mathcal{T}_v}\rangle, \langle v^{(2)}, \mathcal{A}_{v^{(2)}}^{\mathcal{T}_v}\rangle, \ldots\}$ is correctly computed. For ease of presentation, we simplify the denotation of the stack as $\{\mathcal{A}_v^{\mathcal{T}_v}, \mathcal{A}_{v^{(1)}}^{\mathcal{T}_v}, \mathcal{A}_{v^{(2)}}^{\mathcal{T}_v}, \ldots\}$. We prove this claim by induction. The base case for the root node holds since after visiting the root node r, the stack is $\{\mathcal{A}_r^{\mathcal{T}_r}\}$, and thus, $\mathcal{A}_r^{\mathcal{T}_r} = \{r\}$ is true. Assume this claim holds for a node v. We want to prove that it holds for the next node v' (i.e., $P(v') = P(v) + 1$). Based on this assumption, we can deduce that when reaching the node v', the stack is $\{\mathcal{A}_{v'^{(1)}}^{\mathcal{T}_v}, \mathcal{A}_{v'^{(2)}}^{\mathcal{T}_v}, \ldots\}$ and $\mathcal{A}_{v'^{(i)}}^{\mathcal{T}_v}$ is correctly computed. Since the active-node set of v''s parent w.r.t $\mathcal{T}_v$ (i.e., the top element of the stack) is correctly computed, Trie-PathStack can correctly compute $\mathcal{A}_{v'}^{\mathcal{T}_{v'}}$. Consider each element $\mathcal{A}_{v'^{(i)}}^{\mathcal{T}_v}$ in the stack. If $i \le \tau$, then $\text{ED}(v', v'^{(i)}) = i \le \tau$, thus Trie-PathStack can obtain $\mathcal{A}_{v'^{(i)}}^{\mathcal{T}_{v'}}$ correctly by adding v' to $\mathcal{A}_{v'^{(i)}}^{\mathcal{T}_v}$; if $i > \tau$, then $\text{ED}(v', v'^{(i)}) = i > \tau$, thus Trie-PathStack can obtain $\mathcal{A}_{v'^{(i)}}^{\mathcal{T}_{v'}}$ correctly by setting it as $\mathcal{A}_{v'^{(i)}}^{\mathcal{T}_v}$. After pushing $\mathcal{A}_{v'}^{\mathcal{T}_{v'}}$ into the stack, Trie-PathStack correctly gets the stack $\{\mathcal{A}_{v'}^{\mathcal{T}_{v'}}, \mathcal{A}_{v'^{(1)}}^{\mathcal{T}_{v'}}, \mathcal{A}_{v'^{(2)}}^{\mathcal{T}_{v'}}, \ldots\}$ for node v'. Therefore, the claim is proved.

Obviously, for each string pair $\langle s \in \mathcal{S}, t \in \mathcal{S}\rangle$ (Without loss of generality, suppose $P(s) \le P(t)$.), $\text{ED}(s,t) \le \tau$ if and only if both the nodes s and t are leaf nodes of $\mathcal{T}_t$ and s is in $\mathcal{A}_t^{\mathcal{T}_t}$. Based on the proved claim, after visiting a node t, the stack $\{\mathcal{A}_t^{\mathcal{T}_t}, \mathcal{A}_{t^{(1)}}^{\mathcal{T}_t}, \ldots\}$ can be correctly computed, and thus, $\mathcal{A}_t^{\mathcal{T}_t}$ is correct. Also note that for each string s in S and $P(s) \le P(t)$, its corresponding node in $\mathcal{T}_t$ must be a leaf node. Therefore, the theorem is proved. □

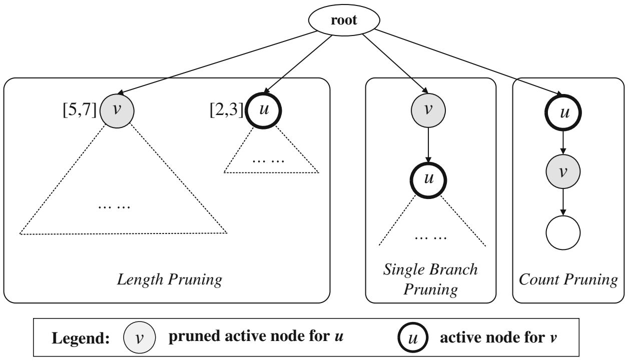


Fig. 10 Three pruning techniques ($\tau = 1$)

3.4 Pruning techniques

Based on dual subtrie pruning, we devise three trie-based algorithms. To further improve the performance, we propose three pruning techniques that can 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 [19]. We exploit this property for pruning in our framework. In Fig. 10, 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.

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 Fig. 10, 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 has 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 Fig. 10, 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 Fig. 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.

The three pruning techniques can be easily plugged into TRIE-TRAVERSE and TRIE-PATHSTACK by adding the pruning process after the calculation of active-node sets (See Figs. 4, 8). However, the pruning techniques are not applicable to TRIE-DYNAMIC, since they rely on the structure of the trie index, but TRIE-DYNAMIC will change the structure.

4 Supporting dynamic data updates and two different sets

4.1 Incremental similarity joins

In this section, we discuss how to extend our methods 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 $\text{ED}(r, s) \leq \tau$.

TRIE-TRAVERSE and TRIE-PATHSTACK can be extended to support incremental similarity joins efficiently, while TRIE-DYNAMIC does not have such advantage since it needs to rebuild the trie for each data update. Next, we use TRIE-PATHSTACK algorithm as an example to show our idea, and the idea can be easily extended to TRIE-TRAVERSE algorithm. 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 that 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 $\mathcal{T}'$. Figure 11 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 $\Delta\mathcal{T}$ as "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. Example 4 shows how the algorithm works.

Example 4 Consider the trie structure $\mathcal{T}$ in Fig. 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" (Fig. 12) 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 (Fig. 12d), we output similar string pair (18,12). The algorithm stops when the stack is empty. □

```
Algorithm 4: 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: $\mathcal{P} = \{(s \in \Delta\mathcal{S}, t \in \mathcal{S} \cup \Delta\mathcal{S}) \mid \text{ED}(s,t) \leq \tau\}$
Change Line 2 in TRIE-PATHSTACK algorithm to
    "$\mathcal{T}$.update($\Delta\mathcal{S}$)" (Insert new added strings into the trie);
Change Line 5 in TRIE-PATHSTACK algorithm to
    "$\mathcal{A}_r^{'}$= {$n$| for each visited trie node $n$, s.t., $|n| \leq \tau$}";
Change Line 9 in TRIE-PATHSTACK algorithm to
    "while $c$ is not null and $c$ is not visited";
Change Line 21- 22 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;"
```

Fig. 11 Incremental TRIE-PATHSTACK algorithm

4.2 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 $\mathcal{S}$. 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 $\mathcal{T}$, a trie node n, and a string set $\mathcal{S}$, node n appears in $\mathcal{S}$ if there exists a string s in $\mathcal{S}$ with a prefix n.

For example, in the left of Fig. 13, given the trie index and $\mathcal{S}$ = {bag, beagy}, node "be" appears to $\mathcal{S}$, 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 Fig. 14. 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 appearing in $\mathcal{R}$, computes its active-node set composed of nodes appearing in $\mathcal{S}$. 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 appear in $\mathcal{S}$. We restrict that only nodes $u \in \mathcal{R}$ can be pushed into the stack (Line 9). Example 5 shows how the algorithm works.

Example 5 In Fig. 13, we illustrate an example to join two different string sets. On the left, it is the trie index for strings

Fig. 12 Incremental similarity joins on sample data set in Fig. 2 ($\Delta\mathcal{S} = \{$"eby"$\}$, $\tau = 1$)

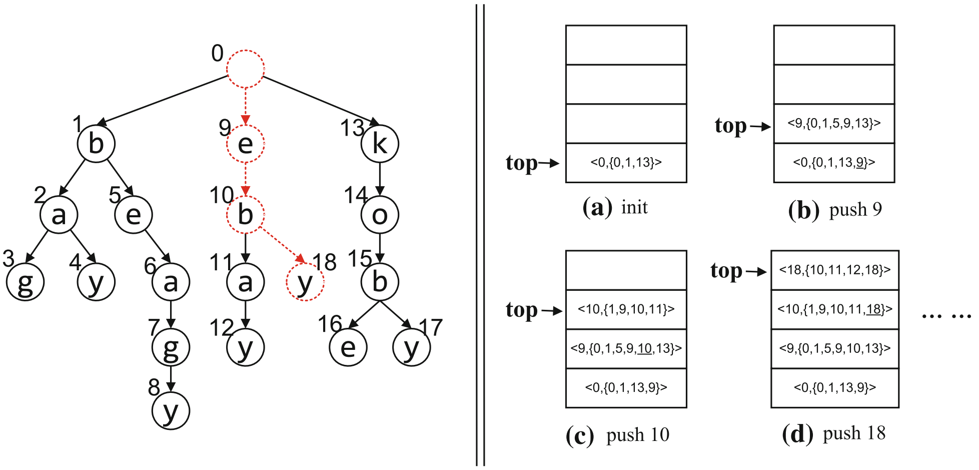
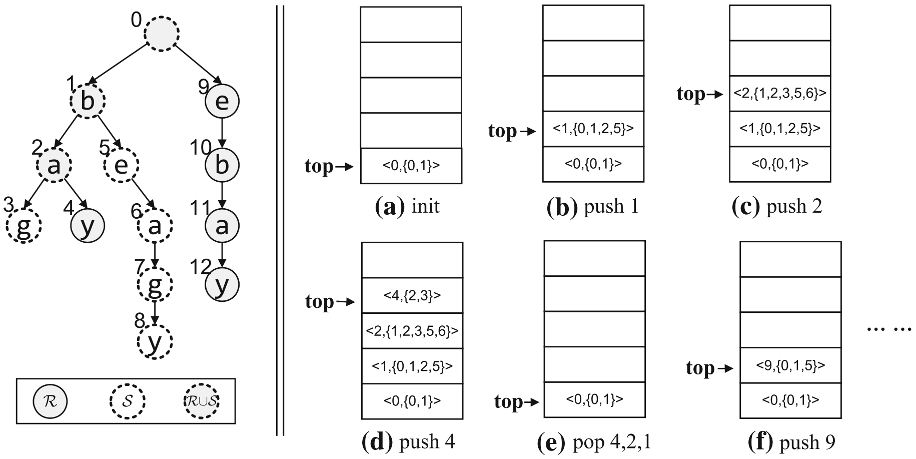


Fig. 13 Similarity joins between $\mathcal{R} = \{\texttt{bay}, \texttt{ebay}\}$ and $\mathcal{S} = \{\texttt{bag}, \texttt{beagy}\}(\tau = 1)$. We push nodes in $\mathcal{R}$ into the stack and find their active nodes in $\mathcal{S}$



in $\mathcal{R} = \{\texttt{bay}, \texttt{ebay}\}$ and $\mathcal{S} = \{\texttt{bag}, \texttt{beagy}\}$. Each node is marked by appearing in which set, such as $\mathcal{R}$, $\mathcal{S}$ or $\mathcal{R} \cup \mathcal{S}$. In Fig. 13a, 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 (Fig. 13c), we then push node 4 into the stack, but will not push node 3 as node $3 \notin \mathcal{R}$. In Fig. 13d, 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. □

Next, we discuss how to extend TRIE-PATHSTACK$^+$ to support dynamic update of data sets efficiently.

Suppose we have gotten the join results of two string sets $\mathcal{R}$ and $\mathcal{S}$. Without loss of generality, assume $\mathcal{R}$ is updated by adding another string set $\Delta\mathcal{R}$, then the incremental TRIE-PATHSTACK$^+$ algorithm can find all similar string pairs $\langle r, s\rangle \in \Delta\mathcal{R} \times \mathcal{S}$ such that $\text{ED}(r, s) \leq \tau$. Figure 15 shows the pseudo-code. Comparing to TRIE-PATHSTACK$^+$, the incremental algorithm firstly updates the original trie by inserting the strings in $\Delta\mathcal{R}$, then for each node appearing in $\Delta\mathcal{R}$, computes its active-node set composed of nodes appearing in $\mathcal{S}$.

5 Improving Trie-PathStack on large edit-distance thresholds

Recall TRIE-PATHSTACK, it traverses the trie constructed from the string set and computes the active-node set for each node. When the edit-distance threshold τ gets larger, the algorithm of computing active-node sets will become more expensive, since the size of the active-node set of each node will increase. To address this problem, we propose a new algorithm BI-TRIE-PATHSTACK to improve TRIE-PATHSTACK on large edit-distance thresholds.

For ease of presentation, given a string $r = r_1 r_2 \ldots r_{|r|}$, we use $\texttt{L}(r) = r_1 r_2 \ldots r_{\lfloor \frac{|r|}{2} \rfloor}$ to denote the left-half part of r and $\texttt{R}(r) = r_{\lfloor \frac{|r|}{2} + 1 \rfloor} \ldots r_{|r|}$ to denote the right-half part of r. We have an observation that for a string s, if r is similar to s within edit-distance threshold τ, then at least one of the following conditions holds: (1) $\texttt{L}(r)$ is similar to a prefix of s within $\lfloor \frac{\tau}{2} \rfloor$; (2) $\texttt{R}(r)$ is similar to a suffix of s within $\lfloor \frac{\tau}{2} \rfloor$. For example, consider a string r = "$\texttt{srivastava}$", and its left-half part $\texttt{L}(r)$ = "$\texttt{sriva}$" and its right-half part $\texttt{R}(r)$ = "$\texttt{stava}$". Given a string s = "$\texttt{sratava}$", as r is

```
Algorithm 5: TRIE-PATHSTACK$^+$ $(\mathcal{R}, \mathcal{S}, \tau)$
   Input: $\mathcal{R}$, $\mathcal{S}$: two collections of strings
          $\tau$: a given edit-distance threshold
   Output: $\mathcal{P} = \{ (s \in \mathcal{R}, t \in \mathcal{S}) \mid \text{ED}(s,t) \le \tau \}$
1  begin
2      $\mathcal{T}$ = new Trie($\mathcal{R} \cup \mathcal{S}$);
3      $\mathbb{S}$ = new Stack();
4      Let $r$ denote the root of Trie $\mathcal{T}$.
5      $\mathcal{A}_r'' = \{n \mid$ for each trie node $n$, s.t., $|n| \le \tau$ and
       $n \in \mathcal{S}\}$;
6      $\mathbb{S}$.push($\langle r, \mathcal{A}_r'' \rangle$);
7      $c = r$.firstchild;
8      while not $\mathbb{S}$.empty() do
9          while $c$ is not null and c $\in \mathcal{R}$ do
10             $\langle p, \mathcal{A}_p'' \rangle = \mathbb{S}$.top();
11             $\mathcal{A}_c'' =$ calcActiveNode$''(c, \mathcal{A}_p'')$;
12             if $c$ is a leaf node then
13                 $\mathcal{P} \cup =$ outputSimilarPair$(c, \mathcal{A}_c'')$;
14             $\mathbb{S}$.push($\langle c, \mathcal{A}_c'' \rangle$);
15             $c = c$.firstchild;
16         if c is not null then
17             $c = c$.nextsibling;
18         else
19             $\langle p, \mathcal{A}_p'' \rangle = \mathbb{S}$.pop();
20             $c = p$.nextsibling;
21 end
```

Fig. 14 TRIE-PATHSTACK$^+$: a similarity-join algorithm for two different sets

```
Algorithm 6: INCREMENTALTRIEJOIN$^+$($\mathcal{T}, \Delta\mathcal{R}, \tau$)
   Input: $\mathcal{T}$: a trie index of original collection of strings
          (i.e. $\mathcal{R} \cup \mathcal{S}$)
          $\Delta\mathcal{R}$: a new added collection of strings to $\mathcal{R}$
          $\tau$: a given edit-distance threshold
   Output: $\mathcal{P} = \{(s \in \Delta\mathcal{R}, t \in \mathcal{S}) \mid \text{ED}(s,t) \le \tau\}$
1  Change Line 2 in TRIE-PATHSTACK$^+$ algorithm to
   "$\mathcal{T}$.update($\Delta\mathcal{R}$)"(Insert new added strings into the
   trie);
2  Change Line 9 in TRIE-PATHSTACK$^+$ algorithm to
   "while $c$ is not null and $c \in \Delta\mathcal{R}$";
```

Fig. 15 Incremental TRIE-PATHSTACK$^+$ algorithm

similar to s within edit-distance threshold 3, we can see the second condition holds, that is $\text{R}(r) =$ "stava" is similar to the suffix "tava" of s within $\lfloor \frac{3}{2} \rfloor = 1$. The following Lemma shows the correctness of this idea.

Lemma 3 *Consider a string* $r = r_1 r_2 \ldots r_{|r|}$. *Its left-half part* $\text{L}(r) = r_1 r_2 \ldots r_{\lfloor \frac{|r|}{2} \rfloor}$ *and right-half part* $\text{R}(r) = r_{\lfloor \frac{|r|}{2}+1 \rfloor} \ldots r_{|r|}$. *Given an edit-distance threshold* τ, *for any string* s, *if* $\text{ED}(r, s) \le \tau$, *then at least one of the following conditions holds:*

1. *There exists a prefix* $\text{P}(s)$ *of* s *such that* $\text{ED}\big(\text{L}(r), \text{P}(s)\big) \le \lfloor \frac{\tau}{2} \rfloor$,
2. *There exists a suffix* $s.$ *of* s *such that* $\text{ED}\big(\text{R}(r),\ s.\big) \le \lfloor \frac{\tau}{2} \rfloor$.

Table 1 A string set $\mathcal{S}$, its left-half part set $\text{L}(\mathcal{S})$, its right-half part set $\text{R}(\mathcal{S})$, its truncated prefix set $\text{P}_{\max}(\mathcal{S}, \tau)$, and its truncated suffix set $\text{S}_{\max}(\mathcal{S}, \tau)$($\text{P}_{\max}(\mathcal{S}, \tau)$ and $\text{S}_{\max}(\mathcal{S}, \tau)$ are defined in Sect. 5.2)

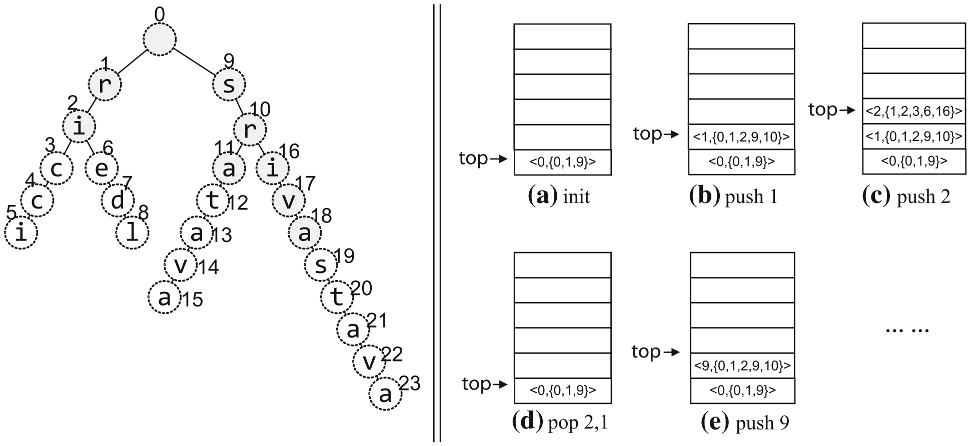
SID	$\mathcal{S}$	$\text{L}(\mathcal{S})$	$\text{R}(\mathcal{S})$	$\text{P}_{\max}(\mathcal{S}, 3)$	$\text{S}_{\max}(\mathcal{S}, 3)$
s_1	ricci	ri	cci	ricc	icci
s_2	riedl	ri	edl	ried	iedl
s_3	srivastava	sriva	stava	srivas	vastava
s_4	sratava	sra	tava	srata	atava

Proof Let $\text{ED}(r, s) = d$ $(d \le \tau)$, i.e., r can be transformed to s with d edit operations. Suppose there are d_1 edit operations on $\text{L}(r)$ and d_2 edit operations on $\text{R}(r)$. If $d_1 \le d_2$, then $d = d_1 + d_2 \ge 2d_1$. As $\tau \ge d \ge 2d_1$, we have $d_1 \le \lfloor \frac{\tau}{2} \rfloor$. That is, there exists a prefix $\text{P}(s)$ of s such that $\text{ED}\big(\text{L}(r), \text{P}(s)\big) = d_1 \le \lfloor \frac{\tau}{2} \rfloor$. Similarly, if $d_1 \ge d_2$, we can prove there exists a suffix $s.$ of s such that $\text{ED}\big(\text{R}(r),\ s.\big) = d_2 \le \lfloor \frac{\tau}{2} \rfloor$. Therefore, the lemma is proved. □

Based on Lemma 3, we propose the algorithm BI-TRIE-PATHSTACK. Given a string set $\mathcal{S}$ and a threshold τ, we first construct a new string set $\text{L}(\mathcal{S})$ that consists of the left-half part of each string in $\mathcal{S}$. Then, we run the TRIE-PATHSTACK$^+$ on $\text{L}(\mathcal{S})$ and $\mathcal{S}$ with edit-distance threshold $\lfloor \frac{\tau}{2} \rfloor$. For a string $\text{L}(r)$ in $\text{L}(\mathcal{S})$, to find all the strings in $\mathcal{S}$ whose prefix is similar to $\text{L}(r)$, we traverse the descendants of each active node of node $\text{L}(r)$ and find the leaf nodes in $\mathcal{S}$. Clearly, these leaf nodes have a prefix that is similar to $\text{L}(r)$. 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 right-half part $\text{R}(r)$ is similar to a suffix of s within $\lfloor \frac{\tau}{2} \rfloor$. We verify the candidate pairs generated from the two cases and obtain final results. Example 6 shows how the algorithm works.

Example 6 Consider the string set $\mathcal{S}$ in Table 1. Given $\tau = 3$, we show how BI-TRIE-PATHSTACK finds all similar string pairs in $\mathcal{S}$. Firstly, we generate the left-half part set $\text{L}(\mathcal{S})$ (see Table 1). To find the string pairs that satisfy the first condition of Lemma 3, we run TRIE-PATHSTACK$^+$ on $\text{L}(\mathcal{S})$ and $\mathcal{S}$ within the edit-distance threshold $\lfloor \frac{3}{2} \rfloor = 1$. Figure 16 shows the algorithm. On the left, it is the trie index constructed from strings in $\text{L}(\mathcal{S}) \cup \mathcal{S}$. The shaded nodes appear in $\text{L}(\mathcal{S})$, and the nodes with dotted-line boundaries appear in $\mathcal{S}$. Firstly, the stack is initialized with node 0 and its active-node set $\{0, 1, 9\}$ (Fig. 16a). Next, node 1,2 and their active-node sets are pushed into the stack, respectively (Fig. 16b, c). As node 2 (i.e., "ri") is a string in $\text{L}(\mathcal{S})$, we traverse the descendants of each active node of node 2 and find the leaf nodes in $\mathcal{S}$. Consider the active-node set $\{1, 2, 3, 6, 16\}$ of node 2. From the descendants of active nodes 1, 2, 3, 6, we obtain the leaf nodes 5, 8; from the descendants of active node 16, we obtain the leaf node 23. Consider the node 2 in $\text{L}(\mathcal{S})$. There are two leaf nodes, node 5 (i.e., "ricci") and node 8 (i.e.,

Fig. 16 Run TRIE-PATHSTACK$^+$ on $L(\mathcal{S})$ and $\mathcal{S}$ (the edit-distance threshold is $\lfloor \frac{\tau}{2} \rfloor = 1$)



(a) init **(b)** push 1 **(c)** push 2 **(d)** pop 2,1 **(e)** push 9

"riedl") whose left-half parts are node 2. For node 5, we can generate two candidates with the obtained leaf nodes in $\mathcal{S}$, i.e., $\langle 5, 8\rangle$, $\langle 5, 23\rangle$. For node 8, we can also generate two candidates with the obtained leaf nodes in $\mathcal{S}$, i.e., $\langle 5, 8\rangle$, $\langle 8, 23\rangle$. Finally, we verify the candidates and output $\langle 5, 8\rangle$ as a result. The TRIE-PATHSTACK$^+$ will continue until the stack is empty (Fig. 16d, e). Using the similar idea, we can also find the string pairs that satisfy the second condition in Lemma 3. □

BI-TRIE-PATHSTACK divides a string into two partitions to improve trie-based methods for large edit distance. Note that it is hard to divide the strings into more than two partitions. Suppose the string r contains three partitions, denoted by the left part $L(r) = r_1 r_2 \ldots r_{\lfloor \frac{|r|}{3} \rfloor}$, the middle part $M(r) = r_{\lfloor \frac{|r|}{3}+1 \rfloor} \cdots r_{\lfloor \frac{2|r|}{3} \rfloor}$, and the right part $R(r) = r_{\lfloor \frac{2|r|}{3}+1 \rfloor} \cdots r_{|r|}$. If r can be transformed into another string s within τ edit operations, there must exist one partition containing no larger than $\lfloor \frac{\tau}{3} \rfloor$ edit operations. That is, at least one of the following three conditions holds: (1) $L(r)$ is similar to some prefixes of s within $\lfloor \frac{\tau}{3} \rfloor$; (2) $R(r)$ is similar to some suffixes of s within $\lfloor \frac{\tau}{3} \rfloor$; (3) $M(r)$ is similar to some substrings of s within $\lfloor \frac{\tau}{3} \rfloor$. Since s may have many substrings, especially for a long string, it would be very expensive to enumerate all substrings of s.

Next, we propose two optimization techniques to further improve BI-TRIE-PATHSTACK.

5.1 Leaf-node optimization

For each string in $L(\mathcal{S})$, BI-TRIE-PATHSTACK will compute its active-node set, and for each node in the active-node set, the algorithm needs to traverse its descendant nodes to get all the leaf nodes. Actually, some leaf nodes can be pruned. Given the left-half part $L(r)$ of a string r, the string r has two possible lengths, $2 \cdot |L(r)|$ and $2 \cdot |L(r)| + 1$. For any string s, if $ED(r, s) \le \tau$, then the length difference $\big||s| - |r|\big|$ between r and s is no larger than τ, thus $|s|$ must be within $\big[2\cdot|L(r)|-\tau,\ 2\cdot|L(r)|+1+\tau\big]$. That is, for each active node of $L(r)$, when traversing its descendant nodes, we can prune the leaf nodes whose depths are smaller than $2 \cdot |L(r)| - \tau$ or larger than $2\cdot|L(r)|+1+\tau$. Moreover, since the depth of the active node of $L(r)$ and the edit distance between the active node and $L(r)$ are known, we can derive a tighter bound for the depths of leaf nodes as shown in Lemma 4.

Lemma 4 *Consider a string $r = r_1 r_2 \ldots r_{|r|}$. Its left-half part $L(r) = r_1 r_2 \ldots r_{\lfloor \frac{|r|}{2} \rfloor}$ and right-half part $R(r) = r_{\lfloor \frac{|r|}{2}+1 \rfloor} \cdots r_{|r|}$. Given an edit-distance threshold τ, for any string s, if $ED(r, s) \le \tau$, then at least one of the following conditions holds:*

1. *There exists a prefix $P(s)$ of s such that $ED\big(L(r), P(s)\big) = d_1 \le \lfloor \frac{\tau}{2} \rfloor$ and the length $|s| \in [\ell_{\min}, \ell_{\max}]$ where $\ell_{\min} = |L(r)| + |P(s)| - \tau + d_1$ and $\ell_{\max} = |L(r)| + |P(s)| + \tau - d_1 + 1$,*
2. *There exists a suffix $\ s.$ of s such that $ED\big(R(r),\ s.\big) = d_2 \le \lfloor \frac{\tau}{2} \rfloor$ and the length $|s| \in [\ell_{\min}, \ell_{\max}]$ where $\ell_{\min} = |R(r)| + |\ s.| - \tau + d_2 - 1$ and $\ell_{\max} = |R(r)| + |\ s.| + \tau - d_2$.*

Proof Since $ED(r, s) \le \tau$ and r is composed of two parts $L(r)$ and $R(r)$, then there exist a prefix $P(s)$ and a suffix $\ s.$ of s such that $|P(s)| + |\ s.| = |s|$, and

$$ED\big(L(r), P(s)\big) + ED\big(R(r),\ s.\big) \le \tau. \tag{5}$$

Firstly, we assume $ED\big(L(r), P(s)\big) \le ED\big(R(r),\ s.\big)$. Let $ED\big(L(r), P(s)\big) = d_1$. Then, $d_1 \le \lfloor \frac{\tau}{2} \rfloor$. Next, we prove $|s| \in [\ell_{\min}, \ell_{\max}]$. Since the length difference between two strings is no larger than their edit distance, we have

$$ED\big(R(r),\ s.\big) \ge \big||R(r)| - |\ s.|\big|. \tag{6}$$

Based on Eqs. 5 and 6, we have

$$\tau - d_1 \ge \big||R(r)| - |\ s.|\big|.$$

As $|P(s)| + |\ s.| = |s|$, we have

$$\tau - d_1 \ge \big||s| - |P(s)| - |R(r)|\big|.$$

Hence,

$$|\mathtt{R}(r)| + |\mathtt{P}(s)| - \tau + d_1 \le |s| \le |\mathtt{R}(r)| + |\mathtt{P}(s)| + \tau - d_1.$$

Based on the definition of $\mathtt{L}(r)$ and $\mathtt{R}(r)$, we have $\mathtt{L}(r) \le \mathtt{R}(r) \le \mathtt{L}(r) + 1$. Thus,

$$|\mathtt{L}(r)| + |\mathtt{P}(s)| - \tau + d_1 \le |s| \le |\mathtt{L}(r)| + |\mathtt{P}(s)| + \tau - d_1 + 1.$$

Secondly, we assume $\mathtt{ED}\big(\mathtt{L}(r), \mathtt{P}(s)\big) \ge \mathtt{ED}\big(\mathtt{R}(r),\ s.\big)$. Let $\mathtt{ED}\big(\mathtt{R}(r),\ s.\big) = d_2$. Then, $d_2 \le \lfloor \frac{\tau}{2} \rfloor$. Similarly, we can prove

$$|\mathtt{L}(r)| + |\ s.| - \tau + d_2 \le |s| \le |\mathtt{L}(r)| + |\ s.| + \tau - d_2.$$

Based on the definition of $\mathtt{L}(r)$ and $\mathtt{R}(r)$, we have $\mathtt{R}(r) - 1 \le \mathtt{L}(r) \le \mathtt{R}(r)$. Thus,

$$|\mathtt{R}(r)| + |\ s.| - \tau + d_2 - 1 \le |s| \le |\mathtt{R}(r)| + |\ s.| + \tau - d_2.$$

Therefore, the lemma is proved. □

Example 7 Recall Example 6, consider node 2 and its active-node set $\{1, 2, 3, 6, 16\}$. By traversing all the descendants of active node 16, we obtain the leaf node 23. Next, we show this leaf node can be pruned based on Lemma 3. For node 2 corresponding to $\mathtt{L}(r) =$ "$\mathtt{ri}$″ and active node 16 corresponding to $\mathtt{P}(s) =$ "$\mathtt{sri}$″, as the edit distance between $\mathtt{L}(r)$ and $\mathtt{P}(s)$ is $d_1 = 1$, the minimum depth of leaf nodes is $\ell_{\min} = |\mathtt{L}(r)| + |\mathtt{P}(s)| - \tau + d_1 = 2 + 3 - 3 + 1 = 3$ and the maximum depth of leaf nodes $\ell_{\max} = |\mathtt{L}(r)| + |\mathtt{P}(s)| + \tau - d_1 + 1 = 2 + 3 + 3 - 1 + 1 = 8$. As the depth of the leaf node 23 is 10 ($\notin [3, 8]$), we can prune it. □

5.2 Trie-size optimization

To find similar string pairs from $\mathcal{S}$, Bi-Trie-PathStack needs to construct two trie indexes: one from $\mathtt{L}(\mathcal{S})$ and $\mathcal{S}$ and the other from $\mathtt{R}(\mathcal{S})$ and $\mathcal{S}$. Reducing the sizes of trie indexes will not only save memory space, but also enhance the efficiency of Trie-PathStack$^+$ for computing active-node sets and traversing the descendants. Recall Lemma 3, consider two similar strings r and s, i.e., $\mathtt{ED}(r, s) \le \tau$. We find that when verifying the first condition, there is no need to retain the whole string s but rather some prefix of s. That is, we only need to verify whether there exists a prefix $\mathtt{P}(s)$ of the retained prefix such that $\mathtt{ED}\big(\mathtt{L}(r), \mathtt{P}(s)\big) \le \lfloor \frac{\tau}{2} \rfloor$. Similarly, when verifying the second condition, there is no need to retain the whole string s but rather some suffix of s. Consider two strings $r = r_1 r_2 \cdots r_{12}$ and $s = s_1 s_2 \cdots s_{10}$. If $\mathtt{ED}(r, s) \le 3$, we only need to retain the prefix $s_1 s_2 s_3 s_4 s_5 s_6$ since transforming the left-half part $\mathtt{L}(r)$ to a longer prefix will make $\mathtt{ED}(r, s)$ larger than 3. For example, if we transform the left-half part $\mathtt{L}(r) = r_1 r_2 r_3 r_4 r_5 r_6$ to a longer prefix $s_1 s_2 s_3 s_4 s_5 s_6 s_7$, there at least needs 1 edit operation. And correspondingly, to transform the right-half part $\mathtt{R}(r) = r_7 r_8 r_9 r_{10} r_{11} r_{12}$ to the rest suffix $s_8 s_9 s_{10}$, there at least needs 3 edit operations. Thus, the minimum number of edit operations of transforming r to s will be 4, which is larger than 3. Based on this idea, given a string s, we prove that the retained prefix length is $\lfloor \frac{|s|+\tau}{2} \rfloor$ and the retained suffix length is $\lfloor \frac{|s|+\tau+1}{2} \rfloor$, as formalized in Lemma 5.

Lemma 5 *Consider a string $r = r_1 r_2 \ldots r_{|r|}$. Its left-half part $\mathtt{L}(r) = r_1 r_2 \ldots r_{\lfloor \frac{|r|}{2} \rfloor}$ and right-half part $\mathtt{R}(r) = r_{\lfloor \frac{|r|}{2}+1 \rfloor} \ldots r_{|r|}$. Given an edit-distance threshold τ, for any string s, if $\mathtt{ED}(r, s) \le \tau$, then at least one of the following conditions holds:*

1. *There exists a prefix $\mathtt{P}(s)$ of s such that $\mathtt{ED}\big(\mathtt{L}(r), \mathtt{P}(s)\big) \le \lfloor \frac{\tau}{2} \rfloor$ and the prefix length $|\mathtt{P}(s)| \le \lfloor \frac{|s|+\tau}{2} \rfloor$,*
2. *There exists a suffix $\ s.$ of s such that $\mathtt{ED}\big(\mathtt{R}(r),\ s.\big) \le \lfloor \frac{\tau}{2} \rfloor$ and the suffix length $|\ s.| \le \lfloor \frac{|s|+\tau+1}{2} \rfloor$.*

Proof Since $\mathtt{ED}(r, s) \le \tau$ and r is composed of two parts $\mathtt{L}(r)$ and $\mathtt{R}(r)$, there exist a prefix $\mathtt{P}(s)$ of s and a suffix $\ s.$ of s such that $|\mathtt{P}(s)| + |\ s.| = |s|$, and

$$\mathtt{ED}\big(\mathtt{L}(r), \mathtt{P}(s)\big) + \mathtt{ED}\big(\mathtt{R}(r),\ s.\big) \le \tau. \tag{7}$$

Firstly, we assume $\mathtt{ED}\big(\mathtt{L}(r), \mathtt{P}(s)\big) \le \mathtt{ED}\big(\mathtt{R}(r),\ s.\big)$. Then, $\mathtt{ED}\big(\mathtt{L}(r), \mathtt{P}(s)\big) \le \lfloor \frac{\tau}{2} \rfloor$. Next, we prove $|\mathtt{P}(s)| \le \lfloor \frac{|s|+\tau}{2} \rfloor$. Since the length difference between two strings is no larger than their edit distance, we have

$$\begin{aligned} &\mathtt{ED}\big(\mathtt{L}(r), \mathtt{P}(s)\big) \ge \big||\mathtt{L}(r)| - |\mathtt{P}(s)|\big|, \\ &\mathtt{ED}\big(\mathtt{R}(r),\ s.\big) \ge \big||\mathtt{R}(r)| - |\ s.|\big|. \end{aligned} \tag{8}$$

Based on Eqs. 7 and 8, we have

$$\begin{aligned} \tau &\ge \big||\mathtt{L}(r)| - |\mathtt{P}(s)|\big| + \big||\mathtt{R}(r)| - |\ s.|\big| \\ &= \big||\mathtt{L}(r)| - |\mathtt{P}(s)|\big| + \big||\mathtt{R}(r)| - |s| + |\mathtt{P}(s)|\big| \\ &\ge 2 \cdot |\mathtt{P}(s)| - |s| + |\mathtt{R}(r)| - |\mathtt{L}(r)|. \end{aligned}$$

Since $|\mathtt{R}(r)| - |\mathtt{L}(r)| \ge 0$, we have $\tau \ge 2 \cdot |\mathtt{P}(s)| - |s|$, thus $|\mathtt{P}(s)| \le \lfloor \frac{|s|+\tau}{2} \rfloor$.

Secondly, we assume $\mathtt{ED}\big(\mathtt{L}(r), \mathtt{P}(s)\big) \ge \mathtt{ED}\big(\mathtt{R}(r),\ s.\big)$. Then, $\mathtt{ED}\big(\mathtt{R}(r),\ s.\big) \le \lfloor \frac{\tau}{2} \rfloor$. Similarly, we can prove

$$\tau \ge 2 \cdot |\ s.| - |s| + |\mathtt{L}(r)| - |\mathtt{R}(r)|.$$

Since $|\mathtt{L}(r)| - |\mathtt{R}(r)| \ge -1$, we have $\tau \ge 2 \cdot |\ s.| - |s| - 1$, thus $|\ s.| \le \lfloor \frac{|s|+\tau+1}{2} \rfloor$.

Therefore, the lemma is proved. □

To incorporate trie-size optimization into Bi-Trie-PathStack, for each string $s \in \mathcal{S}$, we truncate its prefix with the length $\lfloor \frac{|s|+\tau}{2} \rfloor$ (if the length is larger than $|s|$, we keep the original length) and generate a truncated prefix set, denoted by $\mathtt{P}_{\max}(\mathcal{S}, \tau)$; for each string $s \in \mathcal{S}$, we truncate its suffix with the length $\lfloor \frac{|s|+1+\tau}{2} \rfloor$ (if the length is larger than $|s|$, we keep the original length) and generate a truncated suffix set, denoted by $\mathtt{S}_{\max}(\mathcal{S}, \tau)$. Based on Lemma 5, we only need to construct two trie indexes: one is for $\mathtt{L}(\mathcal{S})$ and

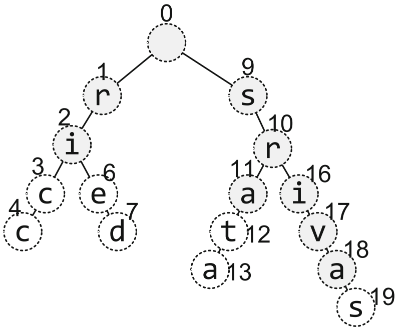


Fig. 17 The trie index over $\mathtt{L}(\mathcal{S})$ and $\mathtt{P}_{\max}(\mathcal{S}, 3)$

Table 2 Data set statistics

Data sets	Sizes	*avg_len*	*max_len*	*min_len*	$\|\Sigma\|$
`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
`DBLP Authors+ Title`	863,267	104.78	1, 743	10	37

$\mathtt{P}_{\max}(\mathcal{S}, \tau)$ the other is for $\mathtt{R}(\mathcal{S})$ and $\mathtt{S}_{\max}(\mathcal{S}, \tau)$. They will be smaller than the trie indexes constructed from $\mathtt{L}(\mathcal{S})$ and $\mathcal{S}$, $\mathtt{R}(\mathcal{S})$ and $\mathcal{S}$, respectively. For example, consider the string set $\mathcal{S} = \{\mathtt{ricci}, \mathtt{riedl}, \mathtt{srivastava}, \mathtt{srastava}\}$ in Table 1. We truncate their prefixes with the length $\lfloor\frac{5+3}{2}\rfloor = 4, \lfloor\frac{5+3}{2}\rfloor = 4, \lfloor\frac{10+3}{2}\rfloor = 6, \lfloor\frac{7+3}{2}\rfloor = 5$ and obtain the truncated prefix set $\mathtt{P}_{\max}(\mathcal{S}, 3) = \{\mathtt{ricc}, \mathtt{ried}, \mathtt{srivas}, \mathtt{srata}\}$ as shown in Table 1. Figure 17 shows the trie index over $\mathtt{L}(\mathcal{S})$ and $\mathtt{P}_{\max}(\mathcal{S}, 3)$. We can see that it is smaller than the trie index in Fig. 16, which is constructed from $\mathtt{L}(\mathcal{S})$ and $\mathcal{S}$.

6 Experiments

6.1 Experiment setup

Data sets: We conducted an extensive set of experimental studies on four real 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 data set.[4] (3) `AOL Query Log`.[5] We randomly chose one million distinct queries. (4) `DBLP Authors+Title` [56]. Each string is a concatenation of author names and the title of a publication. Table 2 illustrates detailed statistical information of the four data sets. Figure 18a–d show their length distribution, respectively.

Common prefix sharing: Given a string set $\mathcal{S}$ and a prefix length ℓ. Let P_ℓ denote the number of the prefixes in $\mathcal{S}$ whose lengths are no larger than ℓ and P_ℓ^d denote the number of the *distinct* prefixes in $\mathcal{S}$ whose lengths are no larger than ℓ. We define the compression ratio of prefixes as $\frac{P_\ell}{P_\ell^d}$. It is easy to see that the higher $\frac{P_\ell}{P_\ell^d}$, the larger numbers of shared common prefixes. We illustrate $\frac{P_\ell}{P_\ell^d}$ for various prefix length on four data sets. Figure 19 shows the results. We can see that a large number of strings share common prefixes. For example, when $\ell = 5$, the compression ratios on every data set are larger than 10.

Implementation of existing algorithms: We compared our algorithms with state-of-the-art methods.

All-Pairs-Ed [7] 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 predefined 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 [56] 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 [5] 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.

M-Tree [15] is a tree-based data structure designed to index metric data sets. It can provide effective pruning for range queries by relying on the triangle inequality. Since edit distance is a metric distance function, we can use M-Tree to index strings to solve our problem. Given an edit-distance threshold τ, for each string s, we search the range query (s, τ) in M-Tree and find all the strings whose edit distances with s are not larger than τ.

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. For M-Tree, we extended the source code in “The M-Tree Project” site[8] to support edit-distance metric. For our trie-based

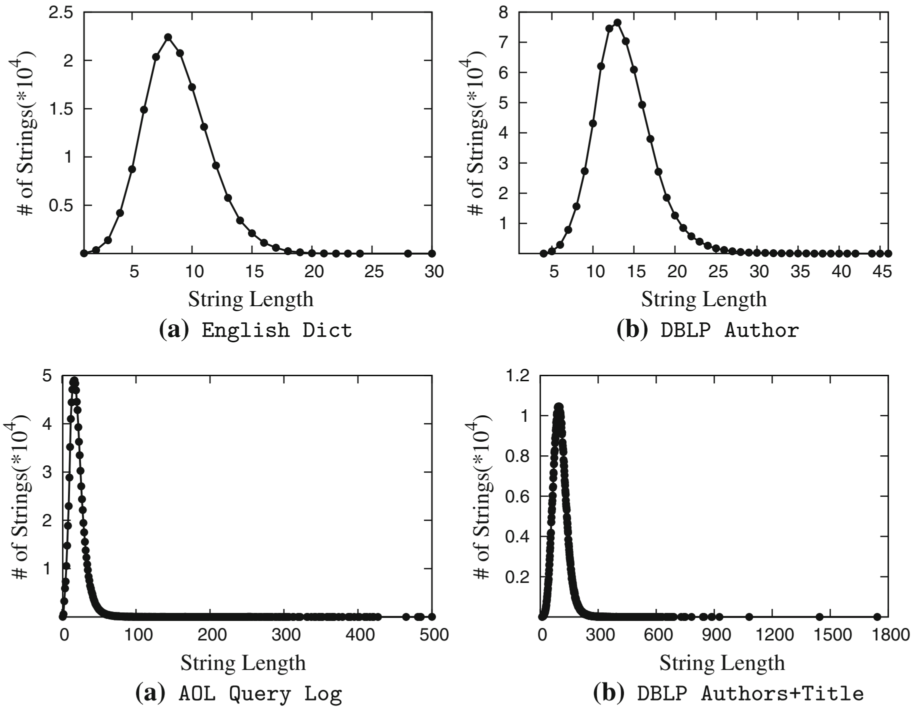
[4] http://www.informatik.uni-trier.de/~ley/db.

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

[6] http://www.cse.unsw.edu.au/~weiw/project/simjoin.html.

[7] http://flamingo.ics.uci.edu/.

[8] http://www-db.deis.unibo.it/Mtree/.



(a) English Dict **(b)** DBLP Author

(a) AOL Query Log **(b)** DBLP Authors+Title

Fig. 18 String length distribution

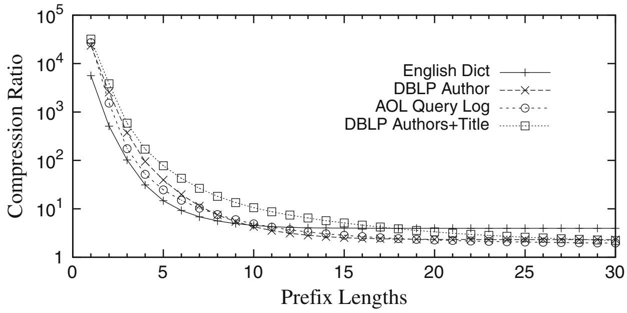


Fig. 19 Evaluation of common prefixes sharing on four data sets

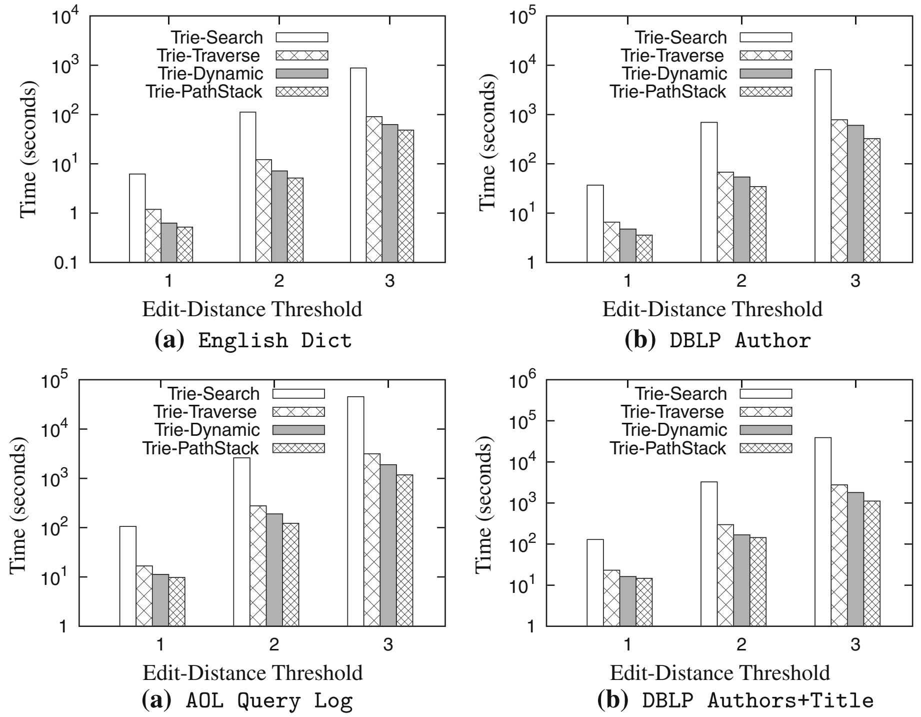
algorithms, we used a radix trie to index strings, which is a compact representation of a trie where any node with only one child is merged with its child. All the algorithms were implemented in C++ and compiled using GCC 4.2.3 with -O3 flag. All the experiments were run on a Ubuntu machine with an Intel Core 2 Quad X5450 3.00 GHz processor and 4 GB memory.

6.2 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 four data sets. As described in Sect. 2.3, Trie-Search is a trie-search-based method algorithm, which only utilizes subtrie pruning. Figure 20a–d 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 AOL Query Log. The results indicate the superiority of using dual subtrie pruning method. 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. Table 3 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.

6.3 Evaluation of pruning techniques

To evaluate the effect of the three pruning techniques, we incorporated them into Trie-PathStack and compared them with Trie-PathStack without pruning on AOL Query Log. We used the number of pruned active nodes to test the pruning power. Figure 21 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



(a) `English Dict` **(b)** `DBLP Author`

(a) `AOL Query Log` **(b)** `DBLP Authors+Title`

Fig. 20 Comparison of the four algorithms

Table 3 Maximal #active nodes on `AOL Query Log`

τ	Trie-Search, Trie-Traverse	Trie-Dynamic	Trie-PathStack
1	2444	42346799	**2172**
2	31374	230511829	**18477**
3	257896	2444928000	**201825**

Table 4 Performance improvement of three pruning techniques on `AOL Query Log`

τ	No pruning (second)	Pruning (second)	Improvement (%)
1	9.76	7.35	24.7
2	122.48	104.48	14.7
3	1,174.94	1,066.12	9.3

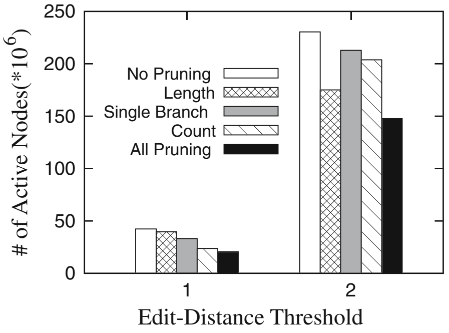


Fig. 21 The number of active nodes of Trie-PathStack with different pruning techniques on `AOL Query Log`

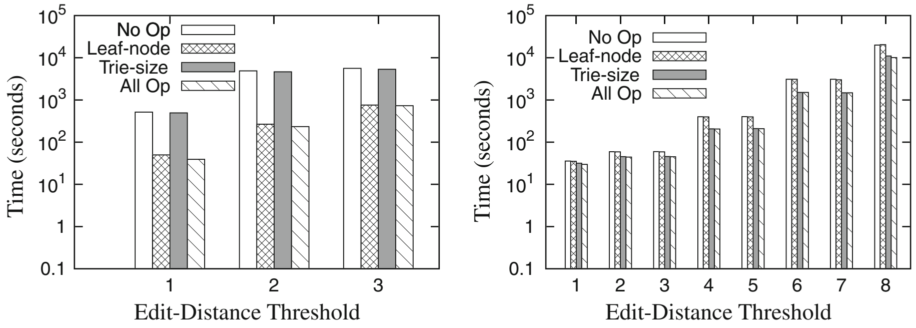
single-branch pruning, with count pruning, and with all three pruning techniques. We can see that the three pruning techniques indeed can prune useless active nodes. For example, length pruning can prune 25% useless active nodes for the edit-distance threshold $\tau = 2$ and count pruning nearly prunes 50% useless active nodes for $\tau = 1$.

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.

6.4 Evaluation of Bi-Trie-PathStack

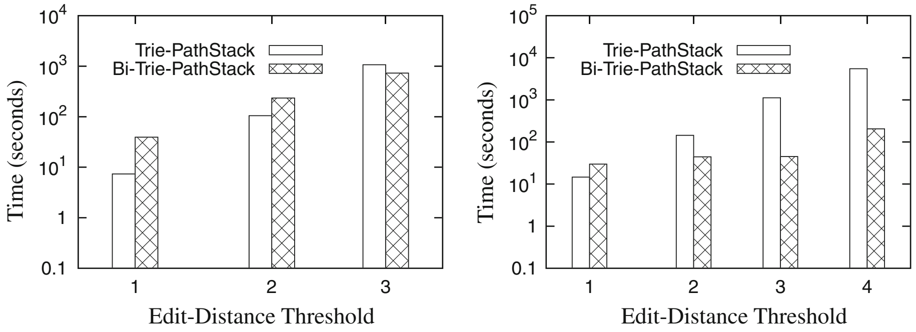
We first evaluated the effect of optimization techniques for Bi-Trie-PathStack. We incorporated the optimization techniques into Bi-Trie-PathStack and compared them with Bi-Trie-PathStack without optimization techniques on `AOL Query Log` and `DBLP Authors+Title`. Figure 22 shows the results. In the figure, we vary the edit-distance threshold and report the running time of four algorithms, "No Op", "Leaf-node", "Trie-size", and "All Op", which respectively denote Bi-Trie-PathStack without any optimization technique, with Leaf-node optimization, with Trie-size optimization, and with both optimization

Fig. 22 Comparison of running time of Bi-Trie-PathStack with different optimization techniques



(a) AOL Query Log **(b)** DBLP Authors+Title

Fig. 23 Comparison of running time between Trie-PathStack and Bi-Trie-PathStack



(a) AOL Query Log **(b)** DBLP Authors+Title

techniques. We can see the two optimization techniques can significantly improve the efficiency of Bi-Trie-PathStack. For example, in Fig. 22a for the edit-distance threshold $\tau = 2$, Bi-Trie-PathStack with both optimization techniques can reduce the computation time from 4882s to 234s. We also observe that on AOL Query Log, the Leaf-node optimization is more effective, while on DBLP Authors+Title, the Trie-size optimization is more effective. In comparison of DBLP Authors+Title, AOL Query Log contains a large number of short strings. For the short strings, there are much more leaf nodes in the descendants of active nodes of their left (right)-half part. Therefore, the Leaf-node optimization is more effective. In comparison of AOL Query Log, DBLP Authors+Title contains a large number of long strings. For the long strings, when truncating their prefixes or suffixes using Lemma 5, we can remove more characters. Therefore, the trie-size optimization is more effective on DBLP Authors+Title. As Bi-Trie-PathStack with both optimization techniques achieves the best performance, we use it to compare with other algorithms in the remainder of this paper.

Next, we compared the efficiency of Bi-Trie-PathStack and Trie-PathStack. We run the two algorithms on AOL Query Log and DBLP Authors+Title by varying the edit-distance thresholds. Figure 23 shows the results. We can see Trie-PathStack performs better than Bi-Trie-PathStack for smaller edit-distance thresholds. For example, in Fig. 23a, we can see that Trie-PathStack took 7.4 s on DBLP Authors+Title with $\tau = 1$, which is about three times faster than Bi-Trie-PathStack. But with the increase in edit-distance thresholds, Bi-Trie-PathStack outperforms Trie-PathStack. For example, in Fig. 23b when τ is larger than 2, Bi-Trie-PathStack outperforms Trie-PathStack by an order of magnitude. These results confirm that Bi-Trie-PathStack improves Trie-PathStack on large edit-distance thresholds.

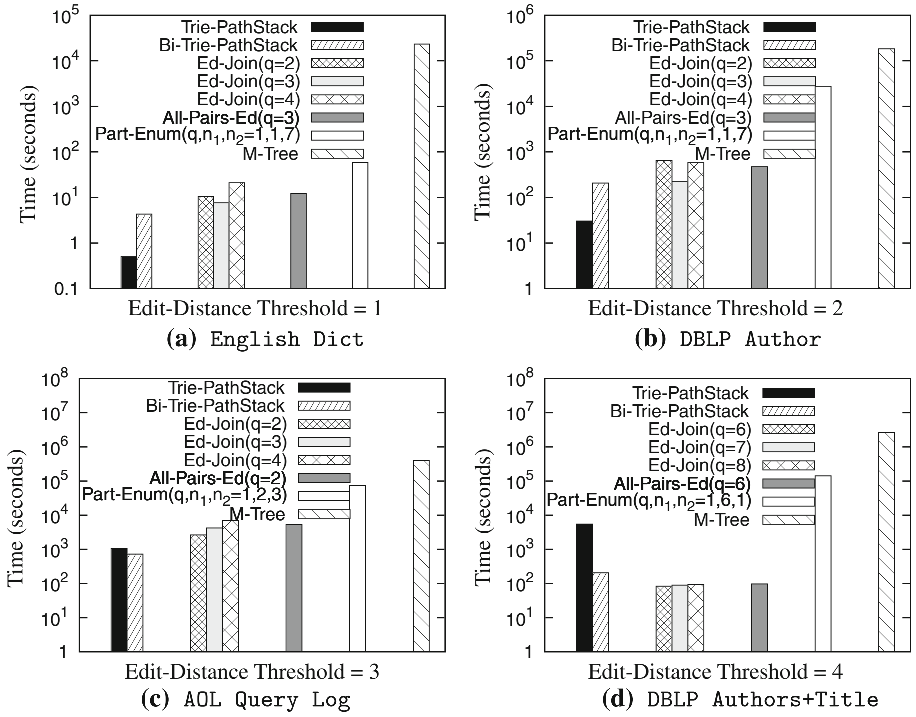
6.5 Comparison with existing methods

Index sizes: We compared index sizes with the state-of-the-art methods, Ed-Join, All-Pairs-Ed, Part-Enum, on four data sets (We exclude the M-Tree method since its index resides in external memory). We tuned their parameters and compared with their best performance. For Bi-Trie-PathStack, we report the index size when $\tau = 1$. Table 5 shows the results. We can observe that existing methods involve much more memory than our methods. For example, their index sizes for AOL Query Log are larger than 100 MB, while Bi-Trie-PathStack has 80 MB and Trie-PathStack only has 29 MB. 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.

Efficiency: We compared efficiency with the state-of-the-art methods, Ed-Join, All-Pairs-Ed, Part-Enum, M-Tree, on four data sets. As the performance of state-of-the-art methods

Table 5 Comparison of index sizes (MB) on four data sets

Data sets	TRIE-PATHSTACK	BI-TRIE-PATHSTACK	Part-Enum	All-Pairs-Ed	Ed-Join
`English Dict`	2	4	16	30	10
`DBLP Author`	16	25	54	155	65
`AOL Query Log`	29	80	120	305	160
`DBLP Authors +Title`	96	127	142	830	751



(a) `English Dict` **(b)** `DBLP Author` **(c)** `AOL Query Log` **(d)** `DBLP Authors+Title`

Fig. 24 Comparison of running time with state-of-the-art methods on four data sets

highly depends on parameters settings, it took considerable time for tuning parameters to optimize their runtime for each experiment. Figure 24 depicts the results. In Fig. 24, 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. We can see M-Tree performs the worst among all algorithms, indicating that M-Tree cannot provide effective pruning for our problem. Figure 24a shows that TRIE-PATHSTACK is about 15 times faster than the best existing method Ed-Join ($q = 3$) on `English Dict` with $\tau = 1$. Figure 24b shows that TRIE-PATHSTACK outperforms the best existing method Ed-Join ($q = 3$), by an order of magnitude on `DBLP Author` with $\tau = 2$. In Fig. 24c, BI-TRIE-PATHSTACK performs the best on `AOL Query Log` with $\tau = 3$. It took 728 s, while Ed-Join ($q = 2$) involved 2,646 s. In Fig. 24d, on `DBLP Authors+Title` with $\tau = 4$, Ed-Join outperforms our methods. Ed-Join ($q = 6$) took 89 seconds, while BI-TRIE-PATHSTACK involved 256 s.

In summary, our methods are always better than Ed-Join on the three data sets with short strings (the average string length is no larger than 30) while on the data set with long strings, Ed-Join performs better. This is because short strings can share a large number of prefixes and our trie-based algorithms have high pruning power for strings with large numbers of shared common prefixes. For example, in both TRIE-PATHSTACK and BI-TRIE-PATHSTACK algorithms, the active nodes for a trie node are computed exactly once even though the node is a prefix of a potentially large number of strings. However, the q-gram-based methods such as Ed-Join do not aggressively share computations across strings. On the other hand, for long strings, the q-gram-based methods have effective pruning power, since they can use large q values. But for short strings, they cannot choose large q values, since such values will destroy the majority or all of grams while small q values will increase the inverted-list sizes and generate large numbers of candidates that need to be further verified.

We also conduct an experiment by varying the amount of shared common prefixes. Given a string set $\mathcal{S}$, and a corresponding trie index $\mathcal{T}$, we define compression ratio as $\frac{\sum_{s \in \mathcal{S}} |s|}{|\mathcal{T}|}$ where $|\mathcal{T}|$ denotes the number of nodes in $\mathcal{T}$. We use

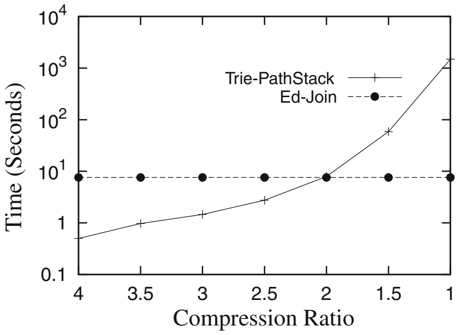


Fig. 25 Comparison of Trie-PathStack and Ed-Join for different compression ratios on `English Dict` ($\tau = 1$)

compression ratio to quantify the ratio of the total number of strings in $\mathcal{S}$ to the number of shared prefixes in $\mathcal{T}$. To evaluate our trie-based methods for different compression ratios, we repeat the following process to decrease compression ratio of the trie. Each time we randomly choose a trie node n with at least two child nodes, split node n into two new nodes, and then evenly append the subtries under node n to this two new nodes. After splitting k nodes, we can decrease the compression ratio of the trie to $\frac{\sum_{s\in\mathcal{S}}|s|}{|\mathcal{T}|+k}$. We compare the running time of Trie-PathStack and Ed-Join by varying the compression ratio of the trie on `English Dict`. Figure 25 shows the result. We can see Trie-PathStack consumed more time with the decreasing of compression ratio. When $\tau = 1$, for compression ratio larger than 2, Trie-PathStack outperforms Ed-Join while for compression ratios smaller than 2, Ed-Join is better. This is because Trie-PathStack utilizes common prefixes to share computations among strings, but for smaller compression ratio, Trie-PathStack cannot utilize much shared computations. However, as shown in Fig. 19, large numbers of strings in real data sets share common prefixes. Therefore, trie-based methods perform much better on real data sets with short strings.

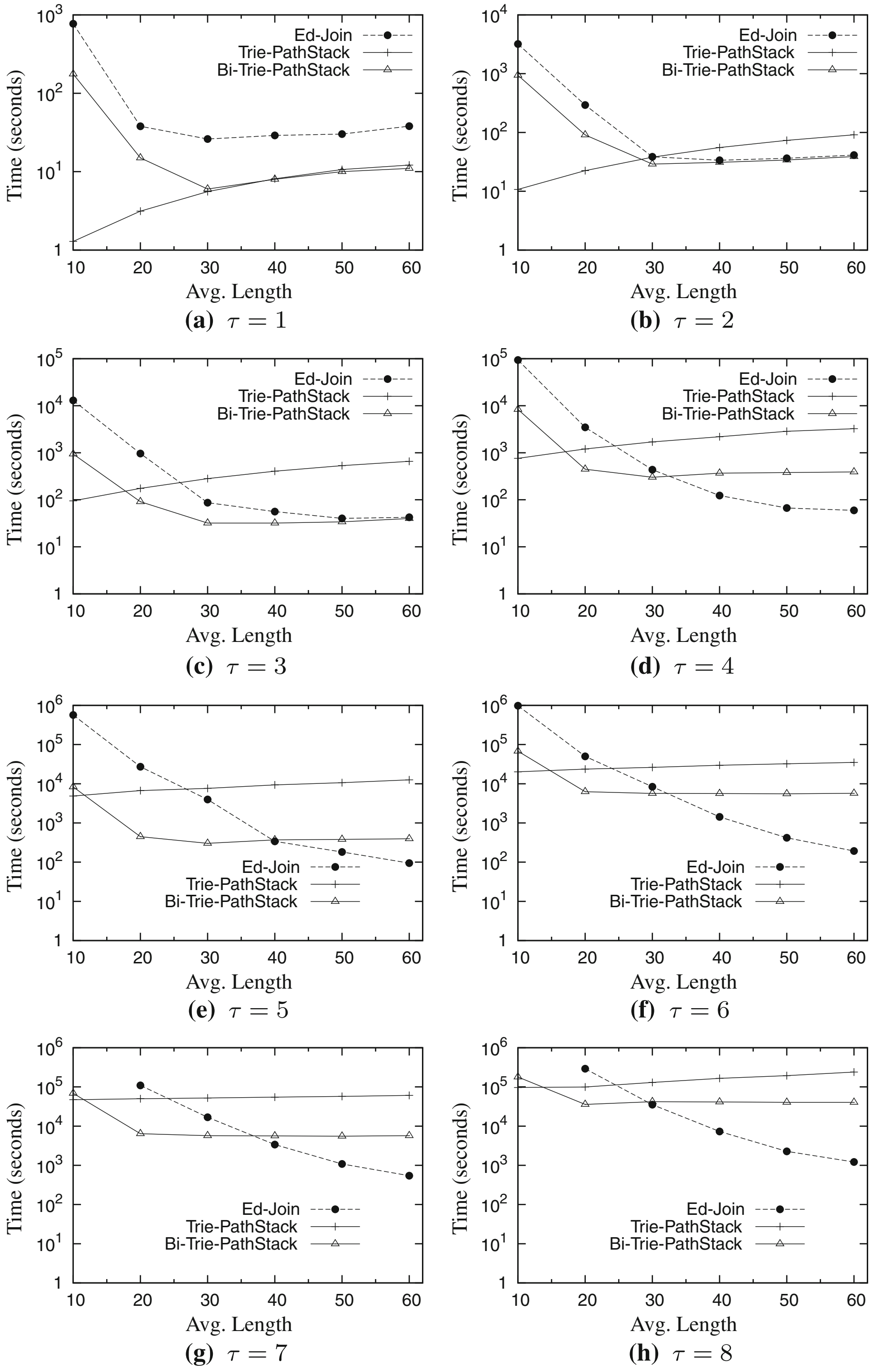
Algorithm selection: To help users select a good algorithm, we conducted an experiment to suggest which algorithms should be used for different data sets. We truncated the prefix of each string in `DBLP Authors+Title` with lengths of 10, 20, 30, 40, 50, and 60 and accordingly generated 6 data sets with different length distributions. In Figure 26, we compared the running time of three algorithms, Ed-Join, Trie-PathStack, 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 Fig. 26a–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 Fig. 26a–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.

Figure 26a–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-PathStack 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 ($\tau \geq 4$) and longer strings (the average string length is larger than 30), as shown in Fig. 26d–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 6 illustrates how to select a good algorithm based on the results from Fig. 26, 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].

Since Ed-Join, Trie-PathStack, Bi-Trie-PathStack algorithms employ different experimental settings, we describe an empirical solution to integrate three algorithms to efficiently support similarity joins in a real system. Given a string set $\mathcal{S}$ and a threshold τ, let $\mathcal{S}_{\text{Short}} = \{s \in \mathcal{S} \mid |s| \leq 30\}$ denote a short string set, and $\mathcal{S}_{\text{Long}} = \{s \in \mathcal{S} \mid |s| > 30 - \tau\}$ denote a long string set. To perform an efficient similarity join on $\mathcal{S}$, we use different algorithms on $\mathcal{S}_{\text{Short}}$ and $\mathcal{S}_{\text{Long}}$. According to the results of algorithm selection in Table 6, when the edit-distance threshold is smaller ($\tau \leq 3$), we use Trie-PathStack and Bi-Trie-PathStack to perform a similarity join on $\mathcal{S}_{\text{Short}}$ and $\mathcal{S}_{\text{Long}}$, respectively; when the edit-distance threshold is larger ($\tau \geq 4$), we use Bi-Trie-PathStack and Ed-Join to perform a similarity join on $\mathcal{S}_{\text{Short}}$ and $\mathcal{S}_{\text{Long}}$, respectively. Note that this method will not miss any result. This is because for two similar strings (i.e., $\text{ED}(r, s) \leq \tau$), their length difference is at most τ, thus they are either both in $\mathcal{S}_{\text{Short}}$ or both in $\mathcal{S}_{\text{Long}}$.



(a) $\tau = 1$ **(b)** $\tau = 2$

(c) $\tau = 3$ **(d)** $\tau = 4$

(e) $\tau = 5$ **(f)** $\tau = 6$

(g) $\tau = 7$ **(h)** $\tau = 8$

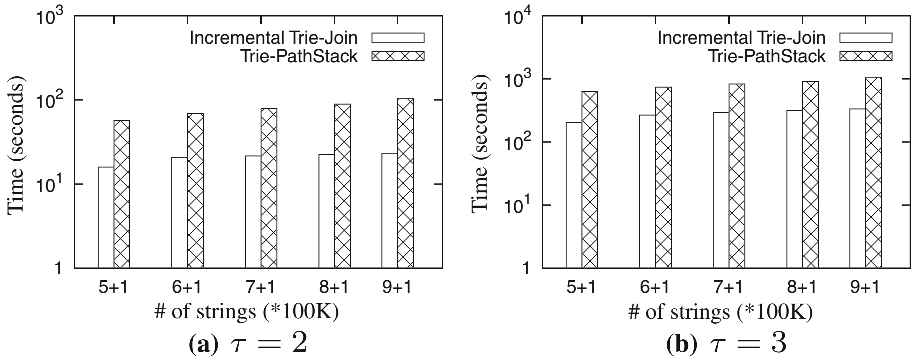
Fig. 26 Comparison of Ed-Join, Trie-PathStack, and Bi-Trie-PathStack on `DBLP Authors+Title` (Note that Ed-Join did not finish in 10^6 s for $\tau = 7, 8$ and the string length 10.)

6.6 Evaluation of update

In this section, we evaluate the performance of our incremental similarity-join algorithm on `AOL Query Log`. Initially, we selected 500 K strings ($\mathcal{S}$), and for each time, we updated it by inserting 100 K strings ($\Delta\mathcal{S}$). We compared therunning time between incremental Trie-Join and Trie-PathStack. We used *speed-up* to evaluate the benefit of our

Table 6 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



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

Fig. 27 Evaluation of update on `AOL Query Log` (e.g., 6+1 denotes $|\mathcal{S}| = 600$ K, $|\Delta\mathcal{S}| = 100$ K)

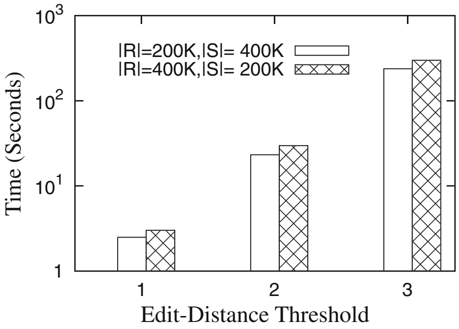


Fig. 28 Evaluation of joining two different data sets on `DBLP Author`

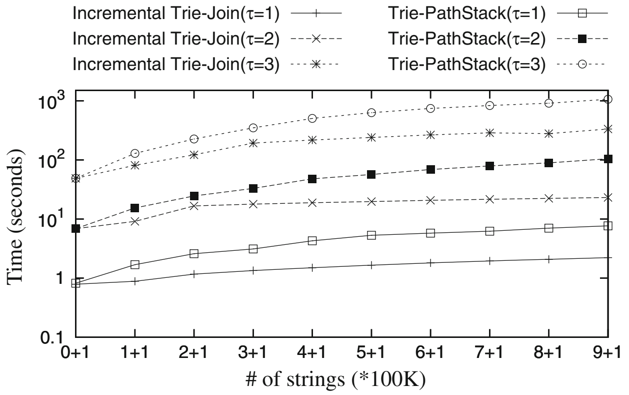


Fig. 29 Scalability on `AOL Query Log` (e.g., 6+1 denotes $|\mathcal{S}| = 600$ K, $|\Delta\mathcal{S}| = 100$ K)

method, which is the ratio between the running time of two algorithms. Figure 27 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 Fig. 27a, the speed-up for $|\mathcal{S}| = 500$ K is 3.5 and that for $|\mathcal{S}| = 900$ K is 4.5. This result shows the superiority of our incremental algorithms.

6.7 Evaluation of joining two data sets

To evaluate the similarity join between two different data sets, we selected 200 and 400 K strings from `DBLP Author` and tested the running time of joining them and the experimental results are shown in Fig. 28. 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.

6.8 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 100 K strings at each time. We compared the running time of the two algorithms and Fig. 29 shows the experimental results with the increase of data sets. We observe that our incremental algorithm scales better than TRIE-PATHSTACK. For example, for 100 K strings, both TRIE-PATHSTACK and incremental trie-join algorithm took 6.88 s ($\tau = 2$); For 1 million strings, TRIE-PATHSTACK increased to 104.65 s, while incremental trie-join algorithm only took 23 s ($\tau = 2$).

We also evaluated BI-TRIE-PATHSTACK for the case that the active nodes cannot fit in main memory. We used the `DBLP Authors+Title` data set with average string length 50 (Sect. 6.5). 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 s from 4×10^4 s for in-memory setting.

7 Related work

String similarity joins have been extensively studied [4,5,7,8,13,19,23,27,38,48,51,53,56,57]. Gravano et al. [19]

proposed to use SQL statements for similarity joins in relational databases. Chaudhuri et al. [13] proposed a primitive operator for effective similarity joins. Arasu et al. [5] developed a signature scheme that can be used as a filter for effective similarity joins. Arasu et al. [4] proposed transformation-based framework for similarity join by using functions to define similar pairs, such as synonyms. Bayardo et al. [7] proposed all-pair similarity joins, a prefix-filtering-based algorithm. Xiao et al. [58] proposed ppjoin to improve all-pair algorithm by introducing positional filtering and suffix filtering. Xiao et al. [57] also studied top-k similarity joins, which can directly find the top-k string pairs without a given threshold. Bryan et al. [8] studied the "output explosion" problem in similarity joins and proposed compact similarity join algorithms to reduce the output size. Recently, Vernica et al. [51] proposed efficient parallel set-similarity joins using the popular MapReduce framework. Lian et al. [38] and Jestes et al. [27] studied similarity joins on probabilistic data. Wang et al. [53] proposed a hybrid similarity metrics, called "fuzzy token matching-based similarity", and studied similarity joins using this new similarity metrics. In our paper, we used edit distance to quantify string similarity. Since edit distance is a metric distance function, some methods of using metric indexes such as M-Tree [15] can be extended to solve our problem. But the methods cannot achieve good performance as shown in our experiments. There are many other string similarity functions such as Jaro [26], Soundex [44], Cosine [47]. *SecondString* [1] and *SimMetrics* [2] are two open-source Java packages that implement a large collection of string functions. Wang et al. [54] studied how to automatically select appropriate similarity functions to address entity-matching problem.

In addition, there have been many studies on approximate string search [9,11,19,21,22,34,35,59]. That is, given a collection of data strings and a query string, it finds all the strings in the collection similar to the query string. Schulz and Mihov [49] proposed an automaton-based approach to address this problem. Kahveci et al. [29] developed an effective index structure for substring matches. Sahinalp et al. [45] proposed an index structure called "VP-tree" for answering NN queries in terms of an edit-distance function. Navarro [41] gave a deep comparison of methods based on edit distance and its variants. Kim et al. [30] proposed a novel technique called "n-Gram/2L" to improve the space and time complexity for q-gram index structures. Li et al. [35] 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. Yang et al. [59] proposed to use a cost model to extend VGRAM. Li et al. [34] developed several list-merging algorithms to improve search efficiency by skipping elements on q-gram inverted lists. Thomas et al. [9] proposed an effective signature scheme by generating a deletion neighborhood for each string, which can improve filtering effect for small edit-distance thresholds.

Another related problem is 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. Navarro [41] gave an excellent survey. These studies can be used to look for common gene expressions. There are also many other related studies, such as estimating selectivity of approximate string queries [24,32,33], approximate entity extraction [3,10,16,36,39,55], and approximate XML joins [6,20]. Note that these problems are different from our similarity-join problem.

The name of trie was derived from "information re*trie*val" [17]. It has many real applications in various fields such as spell checking in natural language processing [43], IP routing in network [42]. In some cases, trie may involve large memory [25]. There are many studies on reducing the trie size [18, 31,40,50]. The trie-based approach to deal with string similarity for edit distance has been proposed in [14,28,37], but they focus on a different problem, fuzzy type-ahead search, which returns answers as users type in keywords letter by letter. They emphasized on 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 Sect. 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.

Compared with our previous work in [52], this article includes the following additional materials. (1) We proposed new optimization techniques to improve our original algorithm on large edit-distance thresholds in Sect. 5 and conducted additional experiments to evaluate the techniques in Sect. 6.4. (2) We formally proved the correctness of our proposed algorithms. (3) We added a new data set in Sect. 6 to help users to select algorithms on different data sets.

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 trie-join algorithms based on dual subtrie pruning to achieve high performance. We developed several pruning techniques to enhance performance. We also extended our method to efficiently support dynamic update of data sets. To support large edit-distance thresholds efficiently, we devised an improved algorithm with two optimization techniques. 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).

Acknowledgments The authors thank the anonymous reviewers for their insightful suggestions. This work was partly supported by the National Natural Science Foundation of China under Grant No. 60873065 and 61003004, the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, National S&T Major Project of China under Grant No. 2011ZX01042-001-002, and the "NExT Research Center" funded by MDA, Singapore, under the research Grant No. WBS:R-252-300-001-490.

References

1. http://secondstring.sourceforge.net/
2. http://www.dcs.shef.ac.uk/~sam/simmetrics.html
3. Agrawal, S., Chakrabarti, K., Chaudhuri, S., Ganti, V.: Scalable ad-hoc entity extraction from text collections. PVLDB **1**(1), 945–957 (2008)
4. Arasu, A., Chaudhuri, S., Kaushik, R.: Transformation-based framework for record matching. In: ICDE, pp. 40–49 (2008)
5. Arasu, A., Ganti, V., Kaushik, R.: Efficient exact set-similarity joins. In: VLDB, pp. 918–929 (2006)
6. Augsten, N., Böhlen, M.H., Dyreson, C.E., Gamper, J.: Approximate joins for data-centric xml. In: ICDE, pp. 814–823 (2008)
7. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: WWW, pp. 131–140 (2007)
8. Bryan, B., Eberhardt, F., Faloutsos, C.: Compact similarity joins. In: ICDE, pp. 346–355 (2008)
9. Celikik, M., Bast, H.: Fast error-tolerant search on very large texts. In: SAC, pp. 1724–1731 (2009)
10. Chakrabarti, K., Chaudhuri, S., Ganti, V., Xin, D.: An efficient filter for approximate membership checking. In: SIGMOD Conference, pp. 805–818 (2008)
11. Chaudhuri, S., Ganjam, K., Ganti, V., Motwani, R.: Robust and efficient fuzzy match for online data cleaning. In: SIGMOD Conference, pp. 313–324 (2003)
12. Chaudhuri, S., Ganti, V., Kaushik, R.: Data debugger: An operator-centric approach for data quality solutions. IEEE Data Eng. Bull. **29**(2), 60–66 (2006)
13. Chaudhuri, S., Ganti, V., Kaushik, R.: A primitive operator for similarity joins in data cleaning. In: ICDE, pp. 5–16 (2006)
14. Chaudhuri, S., Kaushik, R.: Extending autocompletion to tolerate errors. In: SIGMOD Conference, pp. 707–718 (2009)
15. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: VLDB, pp. 426–435 (1997)
16. Cole, R., Gottlieb, L.-A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: STOC, pp. 91–100 (2004)
17. Fredkin, E.: Trie memory. Commun. ACM **3**(9), 490–499 (1960)
18. Gonnet, G.H.: Handbook of Algorithms and Data structures. Addison-Wesley , Reading (1984)
19. Gravano, L., Ipeirotis, P.G., Jagadish, H.V., Koudas, N., Muthukrishnan, S., Srivastava, D.: Approximate string joins in a database (almost) for free. In: VLDB, pp. 491–500 (2001)
20. Guha, S., Koudas, N., Srivastava, D., Yu, T.: Index-based approximate xml joins. In: ICDE, pp. 708–710 (2003)
21. Hadjieleftheriou, M., Chandel, A., Koudas, N., Srivastava, D.: Fast indexes and algorithms for set similarity selection queries. In: ICDE, pp. 267–276 (2008)
22. Hadjieleftheriou, M., Koudas, N., Srivastava, D.: Incremental maintenance of length normalized indexes for approximate string matching. In: SIGMOD Conference, pp. 429–440 (2009)
23. Hadjieleftheriou, M., Srivastava, D.: Weighted set-based string similarity. IEEE Data Eng. Bull. **33**(1), 25–36 (2010)
24. Hadjieleftheriou, M., Yu, X., Koudas, N., Srivastava, D.: Hashed samples: selectivity estimators for set similarity selection queries. PVLDB **1**(1), 201–212 (2008)
25. Heinz, S., Zobel, J., Williams, H.E.: Burst tries: a fast, efficient data structure for string keys. ACM Trans. Inf. Syst. **20**(2), 192–223 (2002)
26. Jaro, M.A. Unimatch: A record linkage system: User's manual. Technical report, U.S. Bureau of the Census, Washington, D.C., (1976)
27. Jestes, J., Li, F., Yan, Z., Yi, K.: Probabilistic string similarity joins. In: SIGMOD Conference, pp. 327–338 (2010)
28. Ji, S., Li, G., Li, C., Feng, J.: Efficient interactive fuzzy keyword search. In WWW, pp. 433–439 (2009)
29. Kahveci, T., Singh, A.K.: Efficient index structures for string databases. In: VLDB, pp. 351–360 (2001)
30. Kim, M.-S., Whang, K.-Y., Lee, J.-G., Lee, M.-J. n-Gram/2L: A space and time efficient two-level n-gram inverted index structure. In: VLDB, pp. 325–336 (2005)
31. Knuth, D.E.: The Art of Computer Programming, Volume 1: Fundamental algorithms. Addison-Wesley, Reading (1968)
32. Lee, H., Ng, R.T., Shim, K.: Extending q-grams to estimate selectivity of string matching with low edit distance. In: VLDB, pp. 195–206 (2007)
33. Lee, H., Ng, R.T., Shim, K.: Power-law based estimation of set similarity join size. PVLDB **2**(1), 658–669 (2009)
34. Li, C., Lu, J., Lu, Y.: Efficient merging and filtering algorithms for approximate string searches. In: ICDE, pp. 257–266 (2008)
35. Li, C., Wang, B., Yang, X. VGRAM: Improving performance of approximate queries on string collections using variable-length grams. In: VLDB, pp. 303–314 (2007)
36. Li, G., Deng, D., Feng, J. Faerie: efficient filtering algorithms for approximate dictionary-based entity extraction. In: SIGMOD Conference, pp. 529–540 (2011)
37. Li, G., Ji, S., Li, C., Feng, J.: Efficient fuzzy full-text type-ahead search. VLDB J. **20**(4), 617–640 (2011)
38. Lian, X., Chen, L.: Set similarity join on probabilistic data. PVLDB **3**(1), 650–659 (2010)
39. Lu, J., Han, J., Meng, X.: Efficient algorithms for approximate member extraction using signature-based inverted lists. In: CIKM, pp. 315–324 (2009)
40. Morrison, D.R.: Patricia: practical algorithm to retrieve information coded in alphanumeric. J. ACM **15**, 514–534 (1968)
41. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. **33**(1), 31–88 (2001)
42. Nilsson, S., Karlsson, G.: Ip-address lookup using lc-tries. IEEE J. Selected Areas Commun. **17**, 1083–1092 (1999)
43. Peterson, J.L.: Computer programs for detecting and correcting spelling errors. Commun. ACM **23**(12), 676–687 (1980)
44. Russell, R.C.: Available at http://patft.uspto.gov/netacgi/nph-Parser?patentnumber=1261167 (1918)
45. Sahinalp, S.C., Tasan, M., Macker, J., Özsoyoglu, Z.M.: Distance based indexing for string proximity search. In: ICDE, pp. 125–136 (2003)
46. Sakoe, H., Chiba, S.: Dynamic programming algorithm optimization for spoken word recognition. IEEE Trans. Acoust Speech Signal Process **26**, 43–49 (1978)
47. Salton, G.: Introduction to Modern Information Retrieval. McGraw Hill, NY (1987)
48. Sarawagi, S., Kirpal, A.: Efficient set joins on similarity predicates. In: SIGMOD Conference, pp. 743–754 (2004)
49. Schulz, K.U., Mihov, S.: Fast string correction with levenshtein automata. Intl J Doc Anal Recognit **5**(1), 67–85 (2002)
50. Sussenguth, E.H.: Use of tree structures for processing files. Commun. ACM **6**, 272–279 (1963)

51. Vernica, R., Carey, M.J., Li, C.: Efficient parallel set-similarity joins using mapreduce. In: SIGMOD Conference, pp. 495–506 (2010)
52. Wang, J., Li, G., Feng, J.: Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. PVLDB **3**(1), 1219–1230 (2010)
53. Wang, J., Li, G., Feng, J.: Fast-join: An efficient method for fuzzy token matching based string similarity join. In: ICDE pp. 458–469 (2011)
54. Wang, J., Li, G., Yu, J.X., Feng, J.: Entity matching: how similar is similar. PVLDB **4**(10), 622–633 (2011)
55. Wang, W., Xiao, C., Lin, X., Zhang, C.: Efficient approximate entity extraction with edit distance constraints. In: SIGMOD Conference, pp. 759–770 (2009)
56. Xiao, C., Wang, W., Lin, X.: Ed-join: an efficient algorithm for similarity joins with edit distance constraints. PVLDB **1**(1), 933–944 (2008)
57. Xiao, C., Wang, W., Lin, X., Shang, H.: Top-k set similarity joins. In: ICDE, pp. 916–927 (2009)
58. Xiao, C., Wang, W., Lin, X., Yu, J.X.: Efficient similarity joins for near duplicate detection. In: WWW, pp. 131–140 (2008)
59. Yang, X., Wang, B., Li, C.: Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In: SIGMOD Conference, pp. 353–364 (2008)