Top-k String Similarity Search with Edit-Distance Constraints

Dong Deng[†] Guoliang Li[†] Jianhua Feng[†] Wen-Syan Li[‡]

[†]*Department of Computer Science, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China.*

dd11@mails.tsinghua.edu.cn; {liguoliang, fengjh}@tsinghua.edu.cn

[‡]*SAP Labs, Shanghai, China.*

wen-syan.li@sap.com

***Abstract*— String similarity search is a fundamental operation in many areas, such as data cleaning, information retrieval, and bioinformatics. In this paper we study the problem of top-k string similarity search with edit-distance constraints, which, given a collection of strings and a query string, returns the top-k strings with the smallest edit distances to the query string. Existing methods usually try different edit-distance thresholds and select an appropriate threshold to find top-k answers. However it is rather expensive to select an appropriate threshold. To address this problem, we propose a progressive framework by improving the traditional dynamic-programming algorithm to compute edit distance. We prune unnecessary entries in the dynamic-programming matrix and only compute those *pivotal entries*. We extend our techniques to support top-k similarity search. We develop a range-based method by grouping the pivotal entries to avoid duplicated computations. Experimental results show that our method achieves high performance, and significantly outperforms state-of-the-art approaches on real-world datasets.**

I. Introduction

String similarity search takes as input a set of strings and a query string, and outputs all the strings in the set that are similar to the query string. It is an important operation and has many real applications such as data cleaning, information retrieval, and bioinformatics. For example, most search engines support query suggestions, which can be implemented using the similarity search operation. Consider a query log {"`schwarzenegger`," "`russell`,"...} and a query "*shwarseneger*." String similarity search returns "`schwarzenegger`" as a suggestion. It has attracted significant attention from the database community recently [10].

Existing similarity search methods [10] require users to specify a similarity function and a similarity threshold. They find those strings with similarities to the query string within the given threshold. However it is rather hard to give an appropriate threshold, as a small threshold will involve many dissimilar answers and a large threshold may lead to few results. To address this problem, in this paper we study the problem of top-k string similarity search, which, given a collection of strings and a query string, returns the top-k most similar strings to the query string.

There are many similarity functions to quantify the similarity of two strings, such as Jaccard similarity, Cosine similarity, and edit distance. In this paper, we focus on edit distance. The edit distance of two strings is the minimum number of single-character edit operations (i.e. insertion, deletion, and substitution) needed to transform one string to another. For example, the edit distance between "`schwarzenegger`" and "`shwarseneger`" is 3. Edit distance can be used to capture typographical errors for text documents and evaluate similarities for homologous proteins or genes [18], and is widely adopted in many real-world applications.

We can extend existing threshold-based similarity search methods [10] to support our problem as follows. We increase the edit-distance threshold by 1 each time (initialized as 0). For each threshold, we use existing methods to find those strings with edit distances to the query string no larger than the threshold. If there are smaller than k similar strings, we check the next threshold; otherwise we compute top-k similar strings with this threshold. However this method is rather expensive because it executes multiple similarity search operations for different thresholds. To address this problem, we propose a progressive framework to efficiently find top-k answers.

A well-known method to compute the edit distance between two strings employs a dynamic-programming algorithm using a matrix (see Section III-A). Notice that we do not need to compute all entries of the matrix. Instead we propose a progressive method which prunes unnecessary entries and only computes some entries. We extend this technique to support top-k similarity search (see Section III-B). To further improve the performance, we propose *pivotal entries* and only need to compute the pivotal entries in the matrix (see Section IV). We develop a range-based method to group the pivotal entries to avoid duplicated computations (see Section V). To summarize, we make the following contributions.

- We devise a progressive framework to address the problem of top-k string similarity search.
- We propose pivotal entries to find top-k answers, which can avoid many unnecessary computations by pruning large numbers of useless entries.
- We develop a range-based method to group the pivotal entries so as to avoid duplicated computations.
- Experimental results show that our method significantly outperforms existing approaches.

Paper Structure. We formalize the top-k string similarity search problem and review related works in Section II. A progressive framework is proposed in Section III. We devise a pivotal-entry based method in Section IV and a range-based method in Section V. Experiment results are provided in Section VI. We conclude the paper in Section VII.

II. PRELIMINARIES

A. Problem Formulation

Given a collection of strings and a query string, top-k string similarity search finds top-k strings with the highest similarities to the query string. In this paper we use edit distance to quantify the similarity of two strings. The edit distance of two strings is the minimum number of single-character edit operations (i.e. insertion, deletion, and substitution) needed to transform one string to another. Given two strings r and s, we use $\text{ED}(r, s)$ to denote their edit distance. For example, ED ("`seraji`", "`sraijt`") = 3. Next we formulate the problem of top-k string similarity search with edit-distance constraints.

Definition 1 (Top-k String Similarity Search): Given a string set $\mathcal{S}$ and a query string q, top-k string similarity search returns a string set $\mathcal{R} \subseteq \mathcal{S}$ such that $|\mathcal{R}| = k$ and for any string $r \in \mathcal{R}$ and $s \in \mathcal{S} - \mathcal{R}$, $\text{ED}(r, q) \leq \text{ED}(s, q)$.

Example 1: Consider the string set in Table I and a query "`srajit`". Top-3 string similarity search returns {"`surajit`", "`seraji`", "`sarit`"}. The edit distances of the three strings to the query are respectively 1, 2 and 2. The edit distances of other strings to the query are not smaller than 2.

TABLE I
A STRING SET $\mathcal{S}$ AND A QUERY q = "`srajit`"

ID	s_1	s_2	s_3	s_4	s_5	s_6
String	`sarit`	`seraji`	`suijt`	`suit`	`surajit`	`thrifty`

B. Related Works

Top-k String Similarity Search: Yang et a. [20] proposed a gram-based method to support top-k similarity search. It increased thresholds by 1 each time from 0 and tuned the gram length dynamically. However it needed to build redundant inverted indexes for different gram lengths and resulted in low efficiency. Zhang et a. [21] indexed signatures (e.g., grams) of strings using a B^+-tree and utilized the B^+-tree to compute top-k answers. It iteratively traversed the B^+-tree nodes, computed a lower bound of edit distances between the query and strings under the node, and used the lower bound to update the threshold. However this method had to enumerate many strings to adjust the threshold. Kahveci et a. [9] transformed a set of contiguous substrings into a Minimum Bounding Rectangle (MBR) and used the MBR to estimate the edit-distance threshold of top-k answers. However the MBR-based estimation usually estimated a large threshold and thus this method resulted in low efficiency.

Different from existing methods, we first identify the strings with smaller edit distances to the query string and use the edit distances of these strings as bounds to facilitate computing top-k answers. We propose a progressive method to efficiently identify such strings and compute the tight bounds.

String Similarity Search with Thresholds: There are many studies on approximate string search [3], [10], [6], [21], which, given a set of strings, a query string, a similarity function, and a threshold, finds all strings with similarities to the query string within the threshold. Existing methods usually employed a gram-based method. They first generated q-grams of each string, and proved that two strings are similar only if their gram sets share *enough* common grams. They used inverted indexes to index the grams. Given a query string, they generated its grams, retrieved the corresponding inverted lists, and merged the inverted lists to find similar answers. We can extend these methods to support the top-k similarity search problem as follows. We increase the edit-distance threshold by 1 each time (initialized as 0). For each threshold, we find similar strings of the query using existing methods [10]. If the size of the similar string set is not smaller than k, we terminate and return k strings with the minimum edit distances. However these methods have to enumerate different edit-distance thresholds and involve many unnecessary computations. Navarro studied the approximate string matching problem [14], which, given a query string and a text string, finds all substrings of the text that are similar to the query.

Similarity Joins: There are many studies on string similarity joins [5], [1], [2], [4], [15], [18], [19], [16], [18], [11], [17]. Given two string sets, a similarity join finds all similar string pairs. Xiao et al. [19] studied top-k similarity joins by using a prefix filtering based technique. Their problem is different from ours which finds top-k similar pairs from two sets. They dynamically tuned the thresholds to find top-k pairs. Jestes et al. [7] extended similarity joins to support probabilistic strings.

Fuzzy Prefix Search: Ji et al. [8] and Li et al. [13], [12] utilized trie structures to support fuzzy prefix search. They specified a threshold and computed results based on the thresholds. Wang et al. [16] proposed a trie-based framework to support similarity joins. Although they used a trie structure to support approximate search, they focused on prefix search and their methods cannot support top-k similarity search.

III. A PROGRESSIVE FRAMEWORK

We first propose a method to progressively compute edit distance (Section III-A) and then develop a progressive framework to support top-k search (Section III-B).

A. Progressively Computing Edit Distance

We first consider the problem of computing the edit distance between two strings. The traditional method uses a dynamic programming method. Given two strings r and s, it utilizes a matrix D with $|r| + 1$ rows and $|s| + 1$ columns to compute their edit distance. Let $|s|$ denote s's length, $s[j]$ denote the j-th character of s, and $s[i, j]$ denote s's substring from the i-th character to the j-th character. $D[i][j]$ is the edit distance between the prefix $r[1, i]$ and the prefix $s[1, j]$. Obviously $D[i][0] = i$ for each $0 \leq i \leq |r|$ and $D[0][j] = j$ for each $0 \leq j \leq |s|$. Then it iteratively computes $D[i][j]$ for $1 \leq i \leq |r|$ and $1 \leq j \leq |s|$ as follows:

$$D[i][j] = \min(D[i-1][j]+1, D[i][j-1]+1, D[i-1][j-1]+\delta), \quad (1)$$

where $\delta = 0$ if $r[i] = s[j]$; otherwise $\delta = 1$. $D[|r|][|s|]$ is exactly the edit distance between r and s. The time complexity is $\mathcal{O}(|r| \times |s|)$ and the space complexity is $\mathcal{O}(\min(|r|, |s|))$. For example, Figure 1(a) shows the matrix D to compute the edit distance of r ="`seraji`" and s ="`srajit`".

		s					
	0	1 s	2 r	3 a	4 j	5 i	6 t
0	0	1	2	3	4	5	6
1 s	1	0	1	2	3	4	5
2 e	2	1	1	2	3	4	5
r 3 r	3	2	1	2	3	4	5
4 a	4	3	2	1	2	3	4
5 j	5	4	3	2	1	2	3
6 i	6	5	4	3	2	1	2

(a) Traditional method

		s					
	0	1 s	2 r	3 a	4 j	5 i	6 t
0	0	1	2				
1 s	1	0	1	2			
2 e	2	1	1	2			
r 3 r		2	1	2			
4 a			2	1	2		
5 j				2	1	2	
6 i					2	1	2

(b) Progressive method

Fig. 1. Computing edit distance of two strings

A progressive method: We propose a progressive method to compute the edit distance, which only computes some entries in the matrix, instead of all entries. We use $\langle i, j\rangle$ to denote the entry of the i-th row and the j-th column of the matrix. Let E_x denote the set of entries in the matrix whose values are x. Given two strings r and s, if $\langle |r|, |s|\rangle \in \mathsf{E}_x$, we have $\text{ED}(r, s) = x$. To compute the edit distance of r and s, we iteratively compute E_x from $x = 0$. Initially we compute E_0. If $\langle |r|, |s|\rangle \in \mathsf{E}_0$, the edit distance between r and s is 0 and we terminate the computation; otherwise we compute E_1 based on E_0. Iteratively we compute the value x such that $\langle |r|, |s|\rangle \in \mathsf{E}_x$ and return x as the edit distance.

Example 2: Figure 1(b) shows the matrix D to compute the edit distance between "`seraji`*" and "*`srajit`*". We first compute $\mathsf{E}_0 = \{\langle 0, 0\rangle, \langle 1, 1\rangle\}$. As $\langle |r|, |s|\rangle \notin \mathsf{E}_0$, we compute $\mathsf{E}_1 = \{\langle 0, 1\rangle, \langle 1, 0\rangle, \langle 1, 2\rangle, \langle 2, 1\rangle, \langle 2, 2\rangle, \langle 3, 2\rangle, \langle 4, 3\rangle, \langle 5, 4\rangle, \langle 6, 5\rangle\}$. As $\langle |r|, |s|\rangle \notin \mathsf{E}_1$, we compute $\mathsf{E}_2 = \{\langle 0, 2\rangle, \langle 1, 3\rangle, \langle 2, 0\rangle, \langle 2, 3\rangle, \langle 3, 1\rangle, \langle 3, 3\rangle, \langle 4, 2\rangle, \langle 4, 4\rangle, \langle 5, 3\rangle, \langle 5, 5\rangle, \langle 6, 4\rangle, \langle 6, 6\rangle\}$. As $\langle |r| = 6, |s| = 6\rangle \in \mathsf{E}_2$, the edit distance between "*`seraji`*" and "*`srajit`*" is 2 and we terminate the computation. Notice that we can prune many useless entries.*

Algorithm to compute E_x: We employ an iterative method to compute E_x. Initially we compute E_0, and then we compute E_{x+1} based on E_x. For ease of presentation, we first introduce an operation called FINDMATCH, which, given two prefixes of two strings, finds the entries with matching characters after the two prefixes. Formally, given two strings r and s, and two integers i and j, FINDMATCH(r, s, i, j) returns a set of entries $\langle i + t, j + t\rangle$ such that $s[i + 1, i + t] = r[j + 1, j + t]$. If the two strings are clear in the context, FINDMATCH(r, s, i, j) is abbreviated as FINDMATCH(i, j). For example, consider two strings "`seraji`" and "`srajit`". For $\langle i = 2, j = 1\rangle$, as $s[i + 1, \overline{i + 4}] = r[j + 1, \overline{j + 4}]$ = "`raji`", $\langle i + 4, j + 4\rangle$ will be returned by the FINDMATCH operation. We have FINDMATCH$(2, 1) = \{\langle 3, 2\rangle, \langle 4, 3\rangle, \langle 5, 4\rangle, \langle 6, 5\rangle\}$.

Initially we use the FINDMATCH operation to compute E_0. For each entry $\langle i, j\rangle \in \mathsf{E}_0$, as $D[i][j] = 0$, we have $i = j$ and $r[1, i] = s[1, j]$. Obviously E_0 = FINDMATCH$(-1, -1)$. Next we use an EXTENSION operation to compute E_{x+1} based on E_x. Let H=$\cup_{t=0}^{x}\mathsf{E}_t$. For each $\langle i, j\rangle \in \mathsf{E}_x$, EXTENSION applies the following operations.

(1) Substitution: As we can substitute $r[i + 1]$ for $s[j + 1]$, $D[i+1][j+1] \leq D[i][j]+1 = x+1$. If $\langle i+1, j+1\rangle \notin \mathsf{H}$ (i.e., $D[i+1][j+1] > x$), then $D[i+1][j+1] = x+1$, thus we add $\langle i+1, j+1\rangle$ into E_{x+1}. As $r[i+2]$ may match $s[j+2]$, there may exist entries $\langle i+t, j+t\rangle$ such that $D[i+t][j+t] = x+1$ for $t \geq 2$. We use FINDMATCH$(i+1, j+1)$ find such entries.

(2) Insertion: As we can insert $r[i+1]$ after $s[j]$, $D[i+1][j] \leq D[i][j] + 1 = x + 1$. If $\langle i + 1, j\rangle \notin \mathsf{H}$, then $D[i + 1][j] = x + 1$, thus we add $\langle i + 1, j\rangle$ into E_{x+1}. Similarly, we use FINDMATCH$(i + 1, j)$ to find entries whose values are $x + 1$.

(3) Deletion: As we can delete $s[j + 1]$ from s, $D[i][j + 1] \leq D[i][j] + 1 = x + 1$. If $\langle i, j + 1\rangle \notin \mathsf{H}$, then $D[i][j + 1] = x + 1$, thus we add $\langle i, j + 1\rangle$ into E_{x+1}. Similarly, we use FINDMATCH$(i, j + 1)$ to find entries whose values are $x + 1$.

We can prove that our progressive method correctly computes E_x as formalized in Lemma 1.

Lemma 1: The entry set E_x computed by our method satisfies (1) completeness: if $D[i][j] = x$, $\langle i, j\rangle$ must be in E_x; and (2) correctness: if $\langle i, j\rangle \in \mathsf{E}_x$, $D[i][j] = x$.

Example 3: Table II illustrates how to compute the edit distance between "`seraji`*" and "*`srajit`*". Firstly, we compute E_0 = FINDMATCH$(-1, -1) = \{\langle 0, 0\rangle, \langle 1, 1\rangle\}$. Then we compute E_1 based on E_0. Consider $\langle 0, 0\rangle \in \mathsf{E}_0$. We want to add $\langle 1, 1\rangle$ (substitution), $\langle 1, 0\rangle$ (insertion), and $\langle 0, 1\rangle$ (deletion), into E_1. As $\langle 1, 1\rangle \in \mathsf{H} = \mathsf{E}_0$, we do not add it into E_1. For $\langle 1, 1\rangle$, we want to add $\langle 2, 2\rangle$ (substitution), $\langle 2, 1\rangle$ (insertion), and $\langle 1, 2\rangle$ (deletion), into E_1. For $\langle 2, 1\rangle \in \mathsf{E}_1$, we use* FINDMATCH *operation to add $\langle 2, 1\rangle$, $\langle 3, 2\rangle$, $\langle 4, 3\rangle$, $\langle 5, 4\rangle$, $\langle 6, 5\rangle$ into E_1. Similarly we compute E_2. As $\langle 6, 6\rangle \in \mathsf{E}_2$, we return 2 as the edit distance of the two strings.*

Complexity: Given two strings r and s, suppose their edit distance is τ. For any entry $\langle i, j\rangle \in \mathsf{E}_x$, we have $|i - j| \leq \text{ED}(r[1, i], s[1, j]) \leq x$. For each i, $i - x \leq j \leq i + x$, thus $|\mathsf{E}_x| \leq (2x+1) \times \min(|r|, |s|)$. The space complexity is $\mathcal{O}\big(\tau \times \min(|r|, |s|)\big)$. In addition, each entry in E_{x+1} is computed from at most three entries (left entry, top entry, top-left entry) in E_x. Thus the time complexity is $\mathcal{O}(\sum_{x=0}^{\tau} |\mathsf{E}_x|)$*. As there are at most $(2\tau + 1) \times \min(|r|, |s|)$ entries in $\cup_{x=0}^{\tau}\mathsf{E}_x$, the time complexity is $\mathcal{O}\big(\tau \times \min(|r|, |s|)\big)$.

B. Progressive Similarity Search

We extend the progressive method to support top-k similarity search. Given a collection of strings, $\mathcal{S}$, and a query string q, for each string $s \in \mathcal{S}$, we compute its entry set, denoted by $\mathsf{E}_x(s)$. Let E'_x denote the set of triples $\langle s, i, j\rangle$ where $s \in \mathcal{S}$ and $\langle i, j\rangle$ in $\mathsf{E}_x(s)$. For each triple $\langle s, i, j\rangle \in \mathsf{E}'_x$, if $i = |s|$ and $j = |q|$, the edit distance between s and q is x and we add it into the result set $\mathcal{R}$. If $|\mathcal{R}| \geq k$, we have found the top-k answers and terminate the iteration.

Next we discuss how to compute E'_x. For $x = 0$, we enumerate each string $s \in \mathcal{S}$ and use the FINDMATCH operation to generate the entry set for s. For each entry $\langle i, j\rangle \in \mathsf{E}_0(s)$, we add triple $\langle s, i, j\rangle$ into E'_0. For $x + 1$, we enumerate each triple $\langle s, i, j\rangle \in \mathsf{E}'_x$ and use the EXTENSION operation to compute $\mathsf{E}_{x+1}(s)$ based on $\mathsf{E}_x(s)$. For each pair $\langle i', j'\rangle$ in $\mathsf{E}_{x+1}(s)$, we add $\langle s, i', j'\rangle$ into E'_{x+1}.

*We use a hash table to implement H, and thus the complexity to check whether an entry is in H is $\mathcal{O}$ (1).

TABLE II

PROGRESSIVELY COMPUTING EDIT DISTANCE ("srajit","seraji")

(a) $\mathsf{E}_0 = \text{FINDMATCH}(-1,-1) = \{\langle 0,0\rangle, \langle 1,1\rangle\}$

(b) Computing E_1 based on E_0

E_0	$\langle$0,0$\rangle$			$\langle$1,1$\rangle$		
	Substitution	Insertion	Deletion	Substitution	Insertion	Deletion
EXTENSION	~~$\langle 1,1\rangle$~~	$\langle 1,0\rangle$	$\langle 0,1\rangle$	$\langle 2,2\rangle$	$\langle 2,1\rangle$ $\langle 3,2\rangle$, $\langle 4,3\rangle$, $\langle 5,4\rangle$, $\langle 6,5\rangle$	$\langle 1,2\rangle$
E_1	$\langle 1,0\rangle$, $\langle 0,1\rangle$, $\langle 2,2\rangle$, $\langle 2,1\rangle$, $\langle 3,2\rangle$, $\langle 4,3\rangle$, $\langle 5,4\rangle$, $\langle 6,5\rangle$, $\langle 1,2\rangle$					

(c) Computing E_2 based on E_1

E_1		$\langle$1,0$\rangle$	$\langle$0,1$\rangle$	$\langle$2,2$\rangle$	$\langle$2,1$\rangle$	$\langle$3,2$\rangle$	$\langle$4,3$\rangle$	$\langle$5,4$\rangle$	$\langle$6,5$\rangle$	$\langle$1,2$\rangle$
EXTENSION	Substitution	~~$\langle 2,1\rangle$~~	~~$\langle 1,2\rangle$~~	~~$\langle 3,3\rangle$~~	~~$\langle 3,2\rangle$~~	~~$\langle 4,3\rangle$~~	~~$\langle 5,4\rangle$~~	~~$\langle 6,5\rangle$~~		~~$\langle 2,3\rangle$~~
	Insertion	$\langle 2,0\rangle$	~~$\langle 1,1\rangle$~~	~~$\langle 3,2\rangle$~~	$\langle 3,1\rangle$	$\langle 4,2\rangle$	$\langle 5,3\rangle$	$\langle 6,4\rangle$		~~$\langle 2,2\rangle$~~
	Deletion	~~$\langle 1,1\rangle$~~	$\langle 0,2\rangle$	$\langle 2,3\rangle$	~~$\langle 2,2\rangle$~~	$\langle 3,3\rangle$	$\langle 4,4\rangle$	$\langle 5,5\rangle$	$\langle \mathbf{6},\mathbf{6}\rangle$	$\langle 1,3\rangle$
E_2		$\langle 2,0\rangle$, $\langle 0,2\rangle$, $\langle 2,3\rangle$, $\langle 3,1\rangle$, $\langle 4,2\rangle$, $\langle 3,3\rangle$, $\langle 5,3\rangle$, $\langle 4,4\rangle$, $\langle 6,4\rangle$, $\langle 5,5\rangle$, $\langle \mathbf{6},\mathbf{6}\rangle$, $\langle 1,3\rangle$								

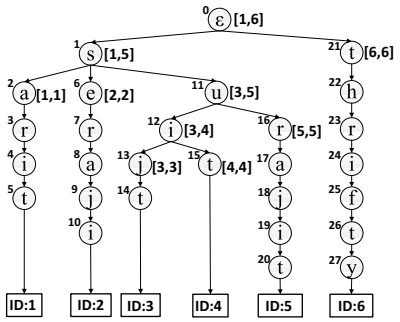


Fig. 2. A trie structure for the strings in Table I

However this method is expensive as it needs to enumerate every strings in $\mathcal{S}$. Notice that we can share computations on common prefixes of different strings. Consider two strings s_1, s_2 and $s_1[1,i] = s_2[1,i]$. For $t \le i$, if $\langle s_1, t, j\rangle \in \mathsf{E}'_x$, then $\langle s_2, t, j\rangle \in \mathsf{E}'_x$, and vice versa. To share the computations on common prefixes, we propose a trie-based method.

We use a trie structure to index the strings in $\mathcal{S}$. Each node on the trie is associated with a character. The character of the root is ϵ. The characters on a path from the root to a leaf node correspond to a string†. Strings with the same prefixes share a common trie node. For simplicity, node n is interchangeably used with its corresponding prefix (the string composed of characters from the root to node n). For example, Figure 2 shows a trie structure for strings in Table I. "suijt", "suit", and "surajit" share a common prefix "su" (node n_{11}).

Next we discuss how to use the trie structure to find top-k answers efficiently. Let T_x denote the set of trie-based entries $\langle n, j\rangle$ with $\text{ED}(n, q[1,j]) = x$, where n is a trie node and j is an integer. For $\langle n, j\rangle$ in T_x, if n is a leaf node and $j = |q|$, we add n into $\mathcal{R}$. If $|\mathcal{R}| \ge k$, we terminate the iteration. Next we discuss how to compute T_x.

For $x = 0$, T_0 is the set of entries $\langle n, j\rangle$, where n matches $q[1,j]$. We compute T_0 as follows. As the root r matches the empty string ϵ, we add $\langle r, 0\rangle$ into T_0. Then we use the operation $\text{FINDMATCH}(r, q, 0)$ to find the entries. If the root has no child with character $q[1]$, FINDMATCH terminates; otherwise, there exists a child n_c with character $q[1]$, FINDMATCH adds $\langle n_c, 1\rangle$ into T_0. Next for node n_c, FINDMATCH checks whether it has a child with label $q[2]$ and repeats the above steps. Iteratively we get T_0.

Next we use the EXTENSION operation to compute T_{x+1} based on T_x. Let $\mathsf{H} = \cup_{t=0}^{x} \mathsf{T}_x$. For $\langle n, j\rangle$ in T_x, EXTENSION applies the following operations.

(1) Substitution: For each child n_c of node n, we can substitute the character of n_c for $q[j+1]$. If $\langle n_c, j+1\rangle \notin \mathsf{H}$, we add $\langle n_c, j+1\rangle$ into T_{x+1}. As n_c may contain a child with label $q[j+2]$, next we call the $\text{FINDMATCH}(n_c, q, j+1)$ operation to find those entries with matching characters.

(2) Insertion: For each child n_c of node n, we can insert character of n_c after $q[j]$. If $\langle n_c, j\rangle \notin \mathsf{H}$, we add $\langle n_c, j\rangle$ into T_{x+1}. We also call the operation $\text{FINDMATCH}(n_c, q, j)$.

(3) Deletion: We can delete $q[j+1]$ from q. If $\langle n, j+1\rangle \notin \mathsf{H}$, we add $\langle n, j+1\rangle$ into T_{x+1}. We also need to call the operation $\text{FINDMATCH}(n, q, j+1)$.

We can prove that T_x computed by our method satisfies completeness and correctness as formalized in Lemma 2.

Lemma 2: The set T_x computed by our method satisfies (1) completeness: If $\text{ED}(n, q[1,j]) = x$, $\langle n, j\rangle$ is in T_x; and (2) correctness: If $\langle n, j\rangle$ is in T_x, $\text{ED}(n, q[1,j]) = x$.

*Example 4: Table III illustrates how to compute top-3 results of query "*srajit*". First, we add $\langle n_0, 0\rangle$ into T_0. Then we call* $\text{FINDMATCH}(n_0, q, 0)$ *and add $\langle n_1, 1\rangle$ into T_0 as n_1 matches $q[1]$. Next we extend each entry in T_0 to generate T_1. For $\langle n_0, 0\rangle$, we want to add $\langle n_1, 1\rangle$, $\langle n_{21}, 1\rangle$ (substitution), $\langle n_1, 0\rangle$, $\langle n_{21}, 0\rangle$ (insertion), and $\langle n_0, 1\rangle$ (deletion). As $\langle n_1, 1\rangle$ is in T_0, we do not add it into T_1. For $\langle n_1, 1\rangle$ in T_0, we want to add $\langle n_2, 2\rangle$ $\langle n_6, 2\rangle$ $\langle n_{11}, 2\rangle$ (substitution), $\langle n_2, 1\rangle$ $\langle n_6, 1\rangle$ $\langle n_{11}, 1\rangle$ (insertion), and $\langle n_1, 2\rangle$ (deletion) into T_1. For $\langle n_1, 2\rangle$, as its child n_2 matches $q[3]$, we use the* FINDMATCH *operation to add $\langle n_2, 3\rangle$ into T_1. Similarly we compute T_1 and T_2. As there are 3 answers in T_2, we terminate and return n_{20} ("*surajit*") with edit distance 1, and n_5 ("*sarit*") and n_{10} ("*seraji*") with edit distance 2.*

Complexity: Given a query string q, an integer k and a string set $\mathcal{S}$, suppose the maximum edit distance between the top-k results and q is τ. For any entry $\langle n, j\rangle \in \mathsf{T}_x$, we have $\big||n| - j\big| \le \text{ED}(n, s[1,j]) \le x$. Thus for each node n, we have $|n| - x \le j \le |n| + x$. Let $|\mathcal{T}|$ denote the number of trie nodes in the first $|q| + \tau$ levels. We have $|\mathsf{T}_x| \le (2x+1)|\mathcal{T}|$. In practice, we can prune many trie nodes, and $|\mathsf{T}_x|$ is much smaller than $(2x+1)|\mathcal{T}|$. Thus the worst-case space complexity is $\mathcal{O}\big(\tau \times |\mathcal{T}|\big)$. For any entry $\langle n, j\rangle \in \cup_{x=0}^{\tau} \mathsf{T}_x$, we compute it from at most three entries, thus the worst-case time complexity is $\mathcal{O}\big(|\cup_{x=0}^{\tau} \mathsf{T}_x| = \tau \times |\mathcal{T}|\big)$.

IV. PIVOTAL ENTRY BASED METHOD

In this section, we propose a method to reduce the size of E_x in order to improve the performance of computing the edit distance between two strings (Section IV-A) and then extend this technique to support similarity search (Section IV-B).

A. Using Pivotal Entries to Compute Edit Distance

Based on the complexity analysis in Section III-A, if we can reduce the size of E_x, we can improve the performance. Here we discuss how to reduce the size of E_x. Consider an entry $\langle i, j\rangle$ in E_x. The goal of keeping $\langle i, j\rangle$ in E_x is to add $\langle i+1, j\rangle$, $\langle i, j+1\rangle$, and $\langle i+1, j+1\rangle$ into E_{x+1}. If $\langle i+1, j+1\rangle$

†To make each tire leaf node corresponds to a string and vice versa is to add a special mark to the end of each string. For simplicity we do not show the mark in the figure.

TABLE III

AN EXAMPLE FOR TOP-3 SIMILARITY SEARCH "srajit" ON $\mathcal{S}$ USING THE PROGRESSIVE SEARCH FRAMEWORK

(a) $\mathsf{T}_0 = \langle n_0, 0\rangle \cup \text{FINDMATCH}(0,0) = \{\langle n_0,0\rangle, \langle n_1,1\rangle\}$

(b) Computing T_1 based on T_0

T_0	$\langle n_0,0\rangle$			$\langle n_1,1\rangle$		
	Substitution	Insertion	Deletion	Substitution	Insertion	Deletion
EXTENSION	~~$\langle n_1,1\rangle$~~ $\langle n_{21},1\rangle$	$\langle n_1,0\rangle$ $\langle n_{21},0\rangle$	$\langle n_0,1\rangle$	$\langle n_2,2\rangle$ $\langle n_6,2\rangle$ $\langle n_{11},2\rangle$	$\langle n_2,1\rangle$ $\langle n_6,1\rangle$ $\langle n_{11},1\rangle$ $\langle n_3,2\rangle$ $\langle n_7,2\rangle$ $\langle n_8,3\rangle$ $\langle n_9,4\rangle$ $\langle n_{10},5\rangle$ $\langle n_{16},2\rangle$ $\langle n_{17},3\rangle$ $\langle n_{18},4\rangle$ $\langle n_{19},5\rangle$ $\langle \mathbf{n_{20}},\mathbf{6}\rangle$	$\langle n_1,2\rangle$ $\langle n_2,3\rangle$
T_1	$\langle n_{21},1\rangle$ $\langle n_1,0\rangle$ $\langle n_{21},0\rangle$ $\langle n_0,1\rangle$ $\langle n_2,2\rangle$ $\langle n_6,2\rangle$ $\langle n_{11},2\rangle$ $\langle n_2,1\rangle$ $\langle n_6,1\rangle$ $\langle n_{11},1\rangle$ $\langle n_3,2\rangle$ $\langle n_7,2\rangle$ $\langle n_8,3\rangle$ $\langle n_9,4\rangle$ $\langle n_{10},5\rangle$ $\langle n_{16},2\rangle$ $\langle n_{17},3\rangle$ $\langle n_{18},4\rangle$ $\langle n_{19},5\rangle$ $\langle \mathbf{n_{20}},\mathbf{6}\rangle$ $\langle n_1,2\rangle$ $\langle n_2,3\rangle$					

(c) Computing T_2 based on T_1

T_1		$\langle n_{21},1\rangle$	$\langle n_0,1\rangle$	$\langle n_1,0\rangle$	$\langle n_{21},0\rangle$	$\langle n_2,2\rangle$	$\langle n_6,2\rangle$	$\langle n_{11},2\rangle$	$\langle n_1,2\rangle$	$\langle n_2,1\rangle$	$\langle n_6,1\rangle$	$\langle n_{11},1\rangle$
EXTENSION	Substitution	$\langle n_{22},2\rangle$	~~$\langle n_1,2\rangle$~~ ~~$\langle n_{21},2\rangle$~~	~~$\langle n_2,1\rangle$~~ ~~$\langle n_6,1\rangle$~~ ~~$\langle n_{11},1\rangle$~~	~~$\langle n_{22},1\rangle$~~	$\langle n_3,3\rangle$	$\langle n_7,3\rangle$	$\langle n_{12},3\rangle$ $\langle n_{16},3\rangle$ $\langle n_{13},4\rangle$	~~$\langle n_2,3\rangle$~~ ~~$\langle n_6,3\rangle$~~ ~~$\langle n_{11},3\rangle$~~	~~$\langle n_3,2\rangle$~~	~~$\langle n_7,2\rangle$~~	~~$\langle n_{12},2\rangle$~~ ~~$\langle n_{16},2\rangle$~~
	Insertion	$\langle n_{22},1\rangle$ $\langle n_{23},2\rangle$	~~$\langle n_1,1\rangle$~~ ~~$\langle n_{21},1\rangle$~~	$\langle n_2,0\rangle$ $\langle n_6,0\rangle$ $\langle n_{11},0\rangle$	$\langle n_{22},0\rangle$	~~$\langle n_3,2\rangle$~~	~~$\langle n_7,2\rangle$~~	$\langle n_{12},2\rangle$ ~~$\langle n_{16},2\rangle$~~	~~$\langle n_2,2\rangle$~~ ~~$\langle n_6,2\rangle$~~ ~~$\langle n_{11},2\rangle$~~	$\langle n_3,1\rangle$	$\langle n_7,1\rangle$	$\langle n_{12},1\rangle$ $\langle n_{16},1\rangle$
	Deletion	$\langle n_{21},2\rangle$	$\langle n_0,2\rangle$	~~$\langle n_1,1\rangle$~~	~~$\langle n_{21},1\rangle$~~	~~$\langle n_2,3\rangle$~~	$\langle n_6,3\rangle$	$\langle n_{11},3\rangle$	$\langle n_1,3\rangle$	~~$\langle n_2,2\rangle$~~	~~$\langle n_6,2\rangle$~~	~~$\langle n_{11},2\rangle$~~
T_2		$\langle n_{22},2\rangle$ $\langle n_{21},2\rangle$ $\langle n_{22},1\rangle$ $\langle n_{23},2\rangle$ $\langle n_0,2\rangle$ $\langle n_2,0\rangle$ $\langle n_6,0\rangle$ $\langle n_{11},0\rangle$ $\langle n_{22},0\rangle$ $\langle n_3,3\rangle$ $\langle n_7,3\rangle$ $\langle n_6,3\rangle$ $\langle n_{12},3\rangle$ $\langle n_{16},3\rangle$ $\langle n_{11},3\rangle$ $\langle n_{12},2\rangle$ $\langle n_{13},4\rangle$ $\langle n_1,3\rangle$ $\langle n_3,1\rangle$ $\langle n_7,1\rangle$ $\langle n_{12},1\rangle$ $\langle n_{16},1\rangle$										

T_1		$\langle n_2,3\rangle$	$\langle n_3,2\rangle$	$\langle n_7,2\rangle$	$\langle n_8,3\rangle$	$\langle n_9,4\rangle$	$\langle n_{10},5\rangle$	$\langle n_{16},2\rangle$	$\langle n_{17},3\rangle$	$\langle n_{18},4\rangle$	$\langle n_{19},5\rangle$	$\langle n_{20},6\rangle$
EXTENSION	Substitution	$\langle n_3,4\rangle$ $\langle n_4,5\rangle$ $\langle \mathbf{n_5},\mathbf{6}\rangle$	$\langle n_4,3\rangle$	~~$\langle n_8,3\rangle$~~	~~$\langle n_9,4\rangle$~~	~~$\langle n_{10},5\rangle$~~		~~$\langle n_{17},3\rangle$~~	~~$\langle n_{18},4\rangle$~~	~~$\langle n_{19},5\rangle$~~	~~$\langle n_{20},6\rangle$~~	
	Insertion	~~$\langle n_3,3\rangle$~~	$\langle n_4,2\rangle$	$\langle n_8,2\rangle$	$\langle n_9,3\rangle$	$\langle n_{10},4\rangle$		$\langle n_{17},2\rangle$	$\langle n_{18},3\rangle$	$\langle n_{19},4\rangle$	$\langle n_{20},5\rangle$	
	Deletion	$\langle n_2,4\rangle$	~~$\langle n_3,3\rangle$~~	~~$\langle n_7,3\rangle$~~	$\langle n_8,4\rangle$	$\langle n_9,5\rangle$	$\langle \mathbf{n_{10}},\mathbf{6}\rangle$	$\langle n_{16},3\rangle$	$\langle n_{17},4\rangle$	$\langle n_{18},5\rangle$	$\langle n_{19},6\rangle$	
T_2		$\langle n_3,4\rangle$ $\langle n_2,4\rangle$ $\langle \mathbf{n_5},\mathbf{6}\rangle$ $\langle n_4,3\rangle$ $\langle n_4,2\rangle$ $\langle n_8,2\rangle$ $\langle n_8,4\rangle$ $\langle n_9,3\rangle$ $\langle n_{18},3\rangle$ $\langle n_{19},4\rangle$ $\langle n_{20},5\rangle$ $\langle n_9,5\rangle$ $\langle \mathbf{n_{10}},\mathbf{6}\rangle$ $\langle n_{10},4\rangle$ $\langle n_{16},3\rangle$ $\langle n_{17},2\rangle$ $\langle n_{17},4\rangle$ $\langle n_{18},5\rangle$ $\langle n_{19},6\rangle$										

is also in E_x, we can remove $\langle i,j\rangle$ from E_x. The main reason is as follows. First, $\langle i+1,j+1\rangle$ is already in E_x, thus it cannot be added into E_{x+1}. Second we prove that $\langle i+1,j\rangle$ and $\langle i,j+1\rangle$ are not needed to add into E_{x+1}. Consider $\langle i+1,j\rangle$. If $D[i+1][j] < x+1$, it will not be in E_{x+1}. If $D[i+1][j] = x+1$, we have $D[i+2][j+1] = x+1$ as stated in Lemma 3. As $D[i+1][j] = D[i+2][j+1]$, we keep $D[i+2][j+1]$ and do not keep $\langle i+1,j\rangle$ in E_{x+1} (Here we only show the idea and the details will be discussed later). Similarly, we do not need to add $\langle i+1,j\rangle$ into E_{x+1}.

Lemma 3: Consider $D[i][j] = D[i+1][j+1] = x$. If $D[i+1][j] = x+1$, we have $D[i+2][j+1] = x+1$. If $D[i][j+1] = x+1$, we have $D[i+1][j+2] = x+1$.

Iteratively, if $\langle i+1,j+1\rangle$, ..., $\langle i+\Delta, j+\Delta\rangle$ are in E_x and $\langle i+\Delta+1, j+\Delta+1\rangle$ are not in E_x, we only keep the last one $\langle i+\Delta, j+\Delta\rangle$ in E_x. Next we formalize our idea. For ease of presentation, we first extend $D[i][j]$ in Equation 1 to support $i > |r|$ or $j > |s|$ as follows.

If $i > |r|$ or $j > |s|$,

$$D[i][j] = \min(D[i][j-1]+1, D[i-1][j]+1, D[i-1][j-1]+1);$$

If $i \le |r|$ and $j \le |s|$,

$$D[i][j] = \min(D[i][j-1]+1, D[i-1][j]+1, D[i-1][j-1]+\delta),$$

where $\delta = 0$ if $r[i] = s[j]$; otherwise $\delta = 1$. If $D[i][j] = D[i+\Delta][j+\Delta]$, we call $\langle i,j\rangle$ is *dominated* by $\langle i+\Delta, j+\Delta\rangle$.

Then we introduce a concept called *pivotal entry.*

Definition 2 (Pivotal Entry): An entry $\langle i,j\rangle$ in E_x is called a pivotal entry, if $D[i+1][j+1] \neq D[i][j]$.

Obviously $\langle |r|, |s|\rangle$ is a pivotal entry. Let E_x^p denote the set of pivotal entries in E_x. If $\langle |r|,|s|\rangle \in \mathsf{E}_x^p$, $\text{ED}(r,s) = x$. To compute the edit distance of r and s, we iteratively compute E_x^p from $x = 0$. If $\langle |r|,|s|\rangle \in \mathsf{E}_x^p$, we return x as their edit distance. For example, in Figure 3, $\langle 0,0\rangle$ is not a pivotal entry as $D[1][1] = D[0][0]$. $\langle 1,1\rangle$ is a pivotal entry and $\mathsf{E}_0^p = \{\langle 1,1\rangle\}$. Although $\langle 2,1\rangle$, $\langle 3,2\rangle$, $\langle 4,3\rangle$, $\langle 5,4\rangle$ are in E_1, they are not pivotal entries, as they are dominated by $\langle 6,5\rangle$. We have E_1^p = $\{\langle 1,2\rangle, \langle 2,2\rangle, \langle 6,5\rangle\}$ and E_2^p = $\{\langle 1,3\rangle, \langle 2,3\rangle, \langle 6,6\rangle\}$.

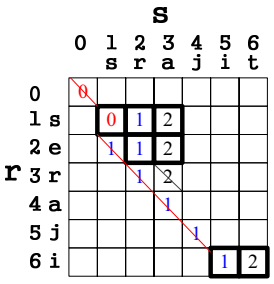


Fig. 3. Using pivotal entries (highlighted ones) to compute edit distance

Algorithm to Compute E_x^p: To compute E_0^p, we extend the FINDMATCH operation and propose a new operation FINDPIVOTAL, which finds the maximal value m such that $r[1,m]$=$s[1,m]$ by checking whether $r[i]$=$s[i]$ for $i \in [1,m]$.

Next we discuss how to compute E_{x+1}^p based on E_x^p. We first prove that there are at most $2x+1$ pivotal entries in E_x^p. First, for any entry $\langle i,j\rangle$, if $|i-j| > x$, we have $D[i][j] > x$ (As the edit distance between $r[1,i]$ and $s[1,j]$ is not smaller than their length difference, i.e., $D[i][j] \ge |i-j|$). Thus for any entry $\langle i,j\rangle$ in E_x, we have $-x \le i-j \le x$. Second, let $\mathsf{E}_x^p[y]$ denote the set of entries whose i-value minus j-value is y, i.e., $y = i-j$. For any $y \in [-x,x]$, there is at most one pivotal entry in $\mathsf{E}_x^p[y]$. We can prove it by contradiction. Suppose there are two entries $\langle i,j\rangle$ and $\langle i',j'\rangle$ in $\mathsf{E}_x^p[y]$. Without loss of generality, suppose $i' > i$. As $D[i][j] = D[i'][j'] = x$, we can prove that for $0 \le \triangle \le i'-i$, $D[i+\triangle][j+\triangle] = x$, as formalized in Lemma 4. Thus $\langle i,j\rangle$ is not a pivotal entry, which contradicts with the assumption.

Lemma 4: Given any two entries $\langle i,j\rangle$ and $\langle i',j'\rangle$ in $\mathsf{E}_x^p[y]$ $(i < i')$, $\forall \triangle \in [0, i'-i]$, $D[i+\triangle][j+\triangle] = x$.

Based on Lemma 4, we prove that $\mathsf{E}_x^p[y]$ has at most one pivotal entry and E_x^p has at most $2x+1$ pivotal entries, as formalized in Lemma 5.

Lemma 5: $\mathsf{E}_x^p[y]$ $(-x \le y \le x)$ has at most one pivotal entry and E_x^p has at most $2x+1$ pivotal entries.

Based on Lemma 5, we only keep $2x+1$ entries in E_x^p. For each entry $\mathsf{E}_x^p[y]$, it may be computed from three entries, and we only need to keep the entry with the maximal i-value. Using this property, next we propose an extension operation called PIVOTALEXTENSION to compute E_{x+1}^p based on E_x^p. For any entry $\langle i,j\rangle = \mathsf{E}_x^p[|i-j|] \in \mathsf{E}_x^p$[‡], PIVOTALEXTENSION applies the following operations.

(1) Substitution: We can substitute $r[i+1]$ for $s[j+1]$, and $\langle i+1, j+1\rangle$ may be a pivotal entry. As $D[i+2][j+2]$ may be equal to $D[i+1][j+1]$, we use the FINDPIVOTAL operation to find entry $\langle i', j'\rangle$ =FINDPIVOTAL$(i+1, j+1)$. If there is no entry in $\mathsf{E}_{x+1}^p[i-j]$, we add $\langle i', j'\rangle$ into $\mathsf{E}_{x+1}^p[i-j]$; otherwise suppose $\langle i'', j''\rangle$ has been added into $\mathsf{E}_{x+1}^p[i-j]$. If $i'>i''$, we use $\langle i', j'\rangle$ to update $\langle i'', j''\rangle$ and $\mathsf{E}_{x+1}^p[i-j]=\langle i', j'\rangle$.

(2) Insertion: We can insert $r[i+1]$ after $s[j]$, and $\langle i+1, j\rangle$ may be a pivotal entry. As $D[i+2][j+1]$ may be equal to $D[i+1][j]$, we use the FINDPIVOTAL operation to find the entry $\langle i', j'\rangle$ =FINDPIVOTAL $(i+1, j)$. If there is no entry in $\mathsf{E}_{x+1}^p[i+1-j]$, we add $\langle i', j'\rangle$ into $\mathsf{E}_{x+1}^p[i+1-j]$; otherwise suppose $\langle i'', j''\rangle$ has been added into $\mathsf{E}_{x+1}^p[i+1-j]$. If $i' > i''$, we use $\langle i', j'\rangle$ to update $\langle i'', j''\rangle$ and $\mathsf{E}_{x+1}^p[i+1-j] = \langle i', j'\rangle$.

(3) Deletion: We can delete $s[j+1]$ from s, and $\langle i, j+1\rangle$ may be a pivotal entry. As $D[i+1][j+2]$ may be equal to $D[i][j+1]$, we use the FINDPIVOTAL operation to find the entry $\langle i', j'\rangle$ =FINDPIVOTAL $(i, j+1)$. If there is no entry in $\mathsf{E}_{x+1}^p[i-(j+1)]$, we add $\langle i', j'\rangle$ into $\mathsf{E}_{x+1}^p[i-(j+1)]$; otherwise suppose $\langle i'', j''\rangle$ has been added into $\mathsf{E}_{x+1}^p[i-(j+1)]$. If $i'>i''$, we use $\langle i', j'\rangle$ to update $\langle i'', j''\rangle$ and $\mathsf{E}_{x+1}^p[i-(j+1)]=\langle i', j'\rangle$.

Iteratively, we can compute E_{x+1}^p[§]. Lemma 6 proves that our method can correctly compute the pivotal set.

Lemma 6: E_x^p computed by our method satisfies (1) completeness: if $\langle i,j\rangle$ is a pivotal entry, $\langle i,j\rangle \in \mathsf{E}_x^p$; and (2) correctness: if $\langle i,j\rangle \in \mathsf{E}_x^p$, $\langle i,j\rangle$ must be a pivotal entry.

Example 5: Table IV illustrates how to use pivotal entries to compute the edit distance of "`srajit`*" and "*`seraji`*". First,* E_0^p = FINDPIVOTAL$(-1,-1) = \{\langle 1,1\rangle\}$. *Then we compute* E_1^p *based on* E_0^p. *Consider* $\langle 1,1\rangle \in \mathsf{E}_0^p$. *We add* $\mathsf{E}_1^p[0] = \langle 2,2\rangle$ *(substitution),* $\mathsf{E}_1^p[1] = \langle 2,1\rangle$ *(insertion) and* $\mathsf{E}_1^p[-1] = \langle 1,2\rangle$ *(deletion) into* E_1^p. *For* $\langle 2,1\rangle \in \mathsf{E}_1^p$, *we use* FINDPIVOTAL *operation to find pivotal entries and set* $\mathsf{E}_1^p[1] = \langle 6,5\rangle$. *For* $\langle 2,2\rangle \in \mathsf{E}_1^p$, *we set* $\mathsf{E}_1^p[0] = \langle 2,2\rangle$. *For* $\langle 1,2\rangle \in \mathsf{E}_1^p$, *we set* $\mathsf{E}_1^p[-1] = \langle 1,2\rangle$. *Then we apply* PIVOTALEXTENSION *operation on* E_1^p. *We add* $\mathsf{E}_2^p[0] = \langle 6,6\rangle$ *into* E_2^p *instead of* $\langle 2,2\rangle$ *and* $\langle 3,3\rangle$ *which are not pivotal entries. Finally, as* $\langle 6,6\rangle \in \mathsf{E}_2^p$, *we return 2 as the edit distance.*

[‡]As $\mathsf{E}_x^p[y]$ has at most one pivotal entry, we refer to $\mathsf{E}_x^p[y]$ as the corresponding pivotal entry.

[§]We can keep entries in E_x^p in order sorted by i-values and visit the entries in descending order to avoid unnecessary operations.

TABLE IV

PIVOTAL ENTRIES TO COMPUTE EDIT DISTANCE("`srajit`","`seraji`")

(a) $\mathsf{E}_0^p = \{\langle 1,1\rangle\}$

(b) Computing E_1^p based on E_0^p

E_0^p	$\mathsf{E}_0^p[0] = \langle 1,1\rangle$		
	Substitution	Insertion	Deletion
EXTENSION	$\mathsf{E}_1^p[0] = \langle 2,2\rangle$	~~$\mathsf{E}_1^p[1] = \langle 2,1\rangle$~~ $\mathsf{E}_1^p[1] = \langle 6,5\rangle$	$\mathsf{E}_1^p[-1] = \langle 1,2\rangle$
E_1^p	$\langle 1,2\rangle$, $\langle 2,2\rangle$, $\langle 6,5\rangle$		

(c) Computing E_2^p based on E_1^p. E_2^p=$\{\langle 1,3\rangle, \langle 2,3\rangle, \langle 6,6\rangle, \langle 7,5\rangle, \langle 7,6\rangle\}$

E_1^p	$\mathsf{E}_1^p[-1] = \langle 1,2\rangle$		
	Substitution	Insertion	Deletion
EXTENSION	$\mathsf{E}_2^p[-1]$=$\langle 2,3\rangle$	$\mathsf{E}_2^p[0]$=~~$\langle 2,2\rangle$~~ $\mathsf{E}_2^p[0]$=$\langle 6,6\rangle$	$\mathsf{E}_2^p[-2]$=$\langle 1,3\rangle$
E_1^p	$\mathsf{E}_1^p[0] = \langle 2,2\rangle$		
	Substitution	Insertion	Deletion
EXTENSION	$\mathsf{E}_2^p[0]$=~~$\langle 3,3\rangle$~~ $\mathsf{E}_2^p[0]$=$\langle 6,6\rangle$	$\mathsf{E}_2^p[1]$=~~$\langle 3,2\rangle$~~ $\mathsf{E}_2^p[1]$=$\langle 7,6\rangle$	$\mathsf{E}_2^p[-1]$=~~$\langle 2,3\rangle$~~ $\mathsf{E}_2^p[-1]$=$\langle 2,3\rangle$
E_1^p	$\mathsf{E}_1^p[1] = \langle 6,5\rangle$		
	Substitution	Insertion	Deletion
EXTENSION	$\mathsf{E}_2^p[1]$=$\langle 7,6\rangle$	$\mathsf{E}_2^p[2]$=$\langle 7,5\rangle$	$\mathsf{E}_2^p[0]$=$\langle 6,6\rangle$

Complexity: As $|\mathsf{E}_x^p| \le 2x+1$ and each row (column) has at most $|r|$ ($|s|$) entries, the space complexity is $\mathcal{O}\big(\min(\tau, |r|, |s|)\big)$. The worst-case time complexity is still $\mathcal{O}\big(\tau \times \min(|r|, |s|)\big)$. Since we can prune many useless entries, this method can improve the performance.

B. Using Pivotal Triples to Support Similarity Search

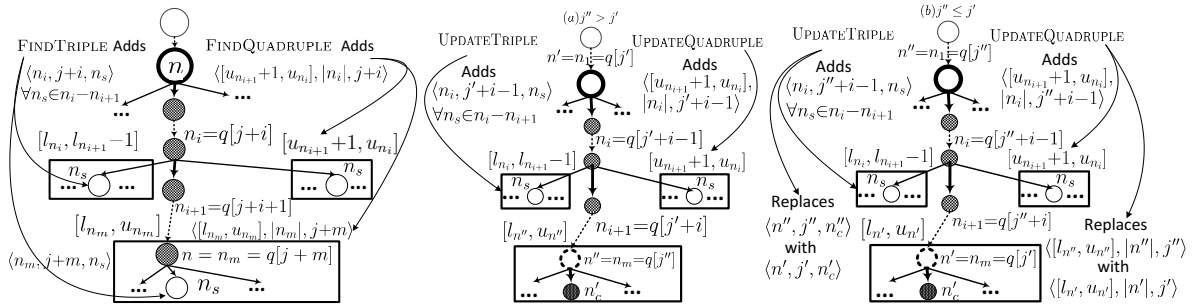
The definition of pivotal entries depends on the two given strings. Consider a trie node n, a query string q, and an integer j. Suppose n_c and n'_c are two children of n. If $\text{ED}(n'_c, q[1, j+1]) \neq \text{ED}(n, q[1,j])$, $\langle n, j\rangle$ is a pivotal entry for strings under node n'_c. If $\text{ED}(n_c, q[1, j+1]) = \text{ED}(n, q[1,j])$, $\langle n, j\rangle$ is not a pivotal entry for strings under node n_c. Thus pivotal entries cannot apply to support multiple strings. To address this issue, we introduce a new concept.

Definition 3 (Pivotal Triple): Given an entry $\langle n, j\rangle$, one of n's children n_c, and a query q, triple $\langle n, j, n_c\rangle$ is called a pivotal triple, if $\text{ED}(n_c, q[1, j+1]) \neq \text{ED}(n, q[1,j])$.

The pivotal triple $\langle n, j, n_c\rangle$ means that for all strings under node n_c, $\langle n, j\rangle$ is a pivotal entry. Let T_x^p denote the pivotal triple set of pivotal triples $\langle n, j, n_c\rangle$ such that $\text{ED}(n, q[1,j]) = x$. We use $\mathsf{T}_x^p[y]$ to denote the subset of pivotal triples in T_x^p with $y = |n| - j$. For example, consider the trie in Figure 2 and query "`srajit`". Consider node n_0 and its child n_1 ("`s`") and child n_{21} ("`t`"). Let $D[n][j] = \text{ED}(n, q[1,j])$. As $D[n_0][0] \neq D[n_{21}][1]$, $\langle n_0, 0, n_{21}\rangle$ is a pivotal triple. As $D[n_0][0] = D[n_1][1]$, $\langle n_0, 0, n_1\rangle$ is not a pivotal triple. Similarly $\langle n_1, 1, n_2\rangle$, $\langle n_1, 1, n_6\rangle$, $\langle n_1, 1, n_{11}\rangle$ are pivotal triples. Thus T_0^p=$\{\langle n_0, 0, n_{21}\rangle, \langle n_1, 1, n_2\rangle, \langle n_1, 1, n_6\rangle, \langle n_1, 1, n_{11}\rangle\}$.

We still iteratively compute T_x^p from $x=0$. For each triple $\langle n, j, n_c\rangle$ in T_x^p, if n is a leaf node and $j = |q|$, the string corresponding to n is an answer and we add it into the result set $\mathcal{R}$. If there are k answers in $\mathcal{R}$, we terminate the iteration. Iteratively, we can compute the top-k answers efficiently. Next we discuss how to compute T_x^p.

Algorithm to Compute T_x^p: For $x=0$, from the root r, for each of its children, n_c, we find pivotal triples as follows.



(a) FINDTRIPLE/FINDQUADRUPLE (b) UPDATETRIPLE/UPDATEQUADRUPLE

Fig. 4. Operations of the pivotal triple based method and the range-based method

Suppose n_c is the child of $n = r$ with label $q[1]$ and let $n - n_c$ denote the set of other children of n except n_c. For each node $n_s \in n - n_c$, as $\text{ED}(n, q[0]) \neq \text{ED}(n_s, q[1])$ ($q[0] = \epsilon$), $\langle n, 0, n_s\rangle$ is a pivotal triple and added into T^p_0 [0] (the entry $\langle n, 0\rangle$ is a pivotal entry for all strings under node n_s). For node n_c, as $\text{ED}(n, q[0]) = \text{ED}(n_c, q[1])$, entry $\langle n, 0\rangle$ is not a pivotal entry for strings under node n_c. Next for each child of node n_c, we repeat the above step to find pivotal triples under node n_c. Iteratively we can compute T^p_0. This is an iterative method and next we introduce an operation called FINDTRIPLE to directly compute the pivotal entries by calling $\text{FINDTRIPLE}(n = r, 0, n_c, q)$ for every child n_c of r.

$\text{FINDTRIPLE}(n, j, n_c, q)$ is extended from the FINDPIVOTAL operation (Figure 4(a)). Let $n_1 = n_c$ and n_m denote the last matching node, that is the label of n_m is $q[j + m]$ and none of its children has a label of $q[j + m + 1]$. FINDPIVOTAL adds the following possible pivotal triples (we will use UPDATETRIPLE to remove the non-pivotal triples later). Algorithm 1 shows the pseudo-code of FINDTRIPLE.

(1) For each node n_i ($1 \leq i \leq m-1$), its child n_{i+1} matches $q[j + i]$. Thus $\text{ED}(n_i, q[1, j + i]) = \text{ED}(n_{i+1}, q[1, j + i + 1])$ and $\langle n_i, j+i, n_{i+1}\rangle$ is not a pivotal triple. For $n_s \in n_i - n_{i+1}$, n_s does not match $q[j+i+1]$ and $\langle n_i, j+i, n_s\rangle$ is a possible pivotal triple. Thus FINDTRIPLE adds $\langle n_i, j + i, n_s\rangle$.

(2) For n_m and each of its child n_s, n_s does not match $q[j + m + 1]$ and $\langle n_m, j + m, n_s\rangle$ is a possible pivotal triple. Thus FINDTRIPLE adds $\langle n_m, j + m, n_s\rangle$.

For example, consider the query "srajit" and the trie structure in Figure 2. For the root, as its child n_{21} does not match $q[1]$, $\langle n_0, 0\rangle$ is a pivotal entry for node n_{21}. Thus $\langle n_0, 0, n_{21}\rangle$ is a pivotal triple. As the child n_1 matches $q[1]$, $\langle n_0, 0\rangle$ is not a pivotal entry for node n_1. Thus $\langle n_0, 0, n_1\rangle$ is not a pivotal triple. Next for n_1, all of its children do not match $q[2]$, $\langle n_1, 1\rangle$ is a pivotal entry for nodes n_2, n_6, n_{11}. Thus $\langle n_1, 1, n_2\rangle$, $\langle n_1, 1, n_6\rangle$, $\langle n_1, 1, n_{11}\rangle$ are pivotal triples.

Next we discuss how to compute T^p_{x+1} based on T^p_x. We propose a new extension operation TRIPLEEXTENSION. For each pivotal triple $\langle n, j, n_c\rangle$ in $\mathsf{T}^p_x[y = |n| - j]$, TRIPLEEXTENSION applies the following operations.

(1) Substitution: For the child n_c of node n, we can substitute the character of n_c for $q[j + 1]$. Thus for each child n_d of n_c, $\langle n_c, j + 1\rangle$ may be a pivotal entry for strings under node n_c and we want to add triple $\langle n_c, j + 1, n_d\rangle$ into $\mathsf{T}^p_{x+1}[y = |n_c| - (j + 1)]$. As $\langle n_c, j + 1, n_d\rangle$ may affect (or be affected by) other triples in $\mathsf{T}^p_{x+1}[y]$. We use function UPDATETRIPLE ($\langle n_c, j + 1, n_d\rangle$) to update pivotal triples as follows.

Algorithm 1: $\text{TOPKPIVOTALSEARCH}(\mathcal{S}, q, k)$

Input: $\mathcal{S}$: A string set; q: A query; k: No of answers;
Output:$\mathcal{R}$: A set of top-k answers for $\mathcal{S}$ and q;

```
x = 0 ;
for each child r_c of root r do
    T^p_x = FINDTRIPLE(r, x, r_c, q) ;
if |R| ≥ k then return R ;
while true do
    T^p_{x+1} = TRIPLEEXTENSION (T^p_x, x) ;
    if |R| ≥ k then return R ;
    ++x ;
```

Function FINDTRIPLE $(n,\ j,\ n_c,\ q)$

Input: n:A node; q:A query; j:An integer; n_c:n's child
Output: T^p: A set of matching entries;

```
if n_c.label≠q[j+1] then T^p ←⟨n, j, n_c⟩;return T^p ;
n_1 = n_c and n_m is the last matching node ;
for i∈[1, m−1] do T^p ←⟨n_i, j+i−1, n_s∈n_i−n_{i+1}⟩;
if n_m is a leaf and |q| = j + m then R ← n_m ;
for each child n_s of n_m do T^p ←⟨n_m, j + m, n_s⟩;
return T^p ;
```

Function TRIPLEEXTENSION $(\mathsf{T}^p_x,\ x)$

Input: T^p_x: A set of entries; x: An integer;
Output: T^p_{x+1}: A set of entries;

```
foreach ⟨n, j, n_c⟩ ∈ T^p_x do
    for each child node n_d of n_c do
        T^p_{x+1}=UPDATETRIPLE(⟨n_c, j + 1, n_d⟩) ;
        T^p_{x+1}=UPDATETRIPLE(⟨n_c, j, n_d⟩) ;
    T^p_{x+1} = UPDATETRIPLE (⟨n, j + 1, n_c⟩) ;
return T^p_{x+1};
```

UPDATETRIPLE (Figure 4(b)): We use function FINDTRIPLE (n_c, $j + 1$, n_d, q) to find possible pivotal triples in $\mathsf{T}^p_{x+1}[y]$, denoted by $\tilde{\mathsf{T}}^p_{x+1}$. For each triple $\langle n', j', n'_c\rangle$ in $\tilde{\mathsf{T}}^p_{x+1}$, let U denote the set of triples $\langle n'', j'', n''_c\rangle$ in $\mathsf{T}^p_{x+1}[y]$ such that $|n''_c| - j'' = y$ and n''_c is a descendant or an ancestor of n'_c. If $\mathsf{U} = \phi$, $\langle n', j', n'_c\rangle$ does not affect (and is not affected by) other triples, and we add $\langle n', j', n'_c\rangle$ into $\mathsf{T}^p_{x+1}[y]$; otherwise for each triple $\langle n'', j'', n''_c\rangle$, we check whether it affects (or is affected by) $\langle n', j', n'_c\rangle$ and update T^p_{x+1} as follows.

TABLE V

AN EXAMPLE FOR TOP-3 SIMILARITY SEARCH "srajit" ON $\mathcal{S}$ USING THE PIVOTAL-BASED SEARCH FRAMEWORK

(a) $\mathsf{T}_0^p = \{\langle n_0, 0, n_{21}\rangle, \langle n_1, 1, n_2\rangle, \langle n_1, 1, n_6\rangle, \langle n_1, 1, n_{11}\rangle\}$

(b) Computing T_1^p based on T_0^p

T_0^p		T_0^p [0]	$\langle n_0, 0, n_{21}\rangle$	$\langle n_1, 1, n_2\rangle$	$\langle n_1, 1, n_6\rangle$	$\langle n_1, 1, n_{11}\rangle$
EXTENSION	Substitution	T_1^p[0]:	$\langle n_{21}, 1, n_{22}\rangle$	$\langle n_2, 2, n_3\rangle$	$\langle n_6, 2, n_7\rangle$	$\langle n_{11}, 2, n_{12}\rangle$ ~~$\langle n_{11}, 2, n_{16}\rangle$~~ $\langle \mathbf{n_{20}}, \mathbf{6}, \phi\rangle$
	Insertion	T_1^p[1]:	$\langle n_{21}, 0, n_{22}\rangle$	~~$\langle n_2, 1, n_3\rangle$~~ $\langle n_3, 2, n_4\rangle$	~~$\langle n_6, 1, n_7\rangle$~~ $\langle n_{10}, 5, \phi\rangle$	$\langle n_{11}, 1, n_{12}\rangle$ $\langle n_{11}, 1, n_{16}\rangle$
	Deletion	T_1^p[−1]:	$\langle n_0, 1, n_{21}\rangle$	~~$\langle n_1, 2, n_2\rangle$~~ $\langle n_2, 3, n_3\rangle$	$\langle n_1, 2, n_6\rangle$	$\langle n_1, 2, n_{11}\rangle$
T_1^p			$\langle n_{21}, 1, n_{22}\rangle$ $\langle n_2, 2, n_3\rangle$ $\langle n_6, 2, n_7\rangle$ $\langle n_{11}, 2, n_{12}\rangle$ $\langle n_0, 1, n_{21}\rangle$ $\langle n_1, 2, n_6\rangle$ $\langle n_1, 2, n_{11}\rangle$ $\langle n_{21}, 0, n_{22}\rangle$ $\langle n_{11}, 1, n_{12}\rangle$ $\langle n_{11}, 1, n_{16}\rangle$ $\langle n_3, 2, n_4\rangle$ $\langle n_2, 3, n_3\rangle$ $\langle n_{10}, 5, \phi\rangle$ $\langle \mathbf{n_{20}}, \mathbf{6}, \phi\rangle$			

(c) Computing T_2^p based on T_1^p

T_1^p		T_1^p [0]	$\langle n_{21}, 1, n_{22}\rangle$	$\langle n_2, 2, n_3\rangle$	$\langle n_6, 2, n_7\rangle$	$\langle n_{11}, 2, n_{12}\rangle$	$\langle n_{20}, 6, \phi\rangle$
EXTENSION	Substitution	T_2^p[0]:	$\langle n_{22}, 2, n_{23}\rangle$	$\langle n_3, 3, n_4\rangle$	~~$\langle n_7, 3, n_8\rangle$~~	~~$\langle n_{12}, 3, n_{13}\rangle$~~ $\langle n_{12}, 3, n_{15}\rangle$	$\langle \phi, 7, \phi\rangle$
	Insertion	T_2^p[1]:	~~$\langle n_{22}, 1, n_{23}\rangle$~~ $\langle n_{23}, 2, n_{24}\rangle$	~~$\langle n_3, 2, n_4\rangle$~~	~~$\langle n_7, 2, n_8\rangle$~~	$\langle n_{12}, 2, n_{13}\rangle$ $\langle n_{12}, 2, n_{15}\rangle$	$\langle \phi, 6, \phi\rangle$
	Deletion	T_2^p[−1]:	$\langle n_{21}, 2, n_{22}\rangle$	~~$\langle n_2, 3, n_3\rangle$~~	~~$\langle n_6, 3, 7\rangle$~~	$\langle n_{11}, 3, n_{12}\rangle$	$\langle n_{20}, 7, \phi\rangle$
T_2^p			$\langle n_{22}, 2, n_{23}\rangle$ $\langle n_3, 3, n_4\rangle$ $\langle n_{12}, 3, n_{15}\rangle$ $\langle \phi, 7, \phi\rangle$ $\langle n_{21}, 2, n_{22}\rangle$ $\langle n_{11}, 3, n_{12}\rangle$ $\langle n_{20}, 7, \phi\rangle$ $\langle n_{12}, 2, n_{13}\rangle$ $\langle n_{12}, 2, n_{15}\rangle$ $\langle \phi, 6, \phi\rangle$ $\langle n_{23}, 2, n_{24}\rangle$ $\langle n_{13}, 4, n_{14}\rangle$			

T_1^p		T_1^p [-1]	$\langle n_0, 1, n_{21}\rangle$	$\langle n_2, 3, n_3\rangle$	$\langle n_1, 2, n_6\rangle$	$\langle n_1, 2, n_{11}\rangle$
EXTENSION	Substitution	T_2^p[−1]:	~~$\langle n_{21}, 2, n_{22}\rangle$~~	~~$\langle n_3, 4, n_4\rangle$~~ $\langle \mathbf{n_5}, \mathbf{6}, \phi\rangle$	~~$\langle n_6, 3, n_7\rangle$~~	~~$\langle n_{11}, 3, n_{12}\rangle$~~ $\langle n_{11}, 3, n_{16}\rangle$
	Insertion	T_2^p[0]:	~~$\langle n_{21}, 1, n_{22}\rangle$~~	~~$\langle n_3, 3, n_4\rangle$~~	~~$\langle n_6, 2, n_7\rangle$~~	~~$\langle n_{11}, 2, n_{12}\rangle$~~ ~~$\langle n_{11}, 2, n_{16}\rangle$~~
	Deletion	T_2^p[−2]:	$\langle n_0, 2, n_{21}\rangle$	$\langle n_2, 4, n_3\rangle$	$\langle n_1, 3, n_6\rangle$	$\langle n_1, 3, n_{11}\rangle$
T_2^p			$\langle n_{11}, 3, n_{16}\rangle$ $\langle n_0, 2, n_{21}\rangle$ $\langle n_2, 4, n_3\rangle$ $\langle n_1, 3, n_6\rangle$ $\langle n_1, 3, n_{11}\rangle$ $\langle \mathbf{n_5}, \mathbf{6}, \phi\rangle$			

T_1^p		T_1^p [1]	$\langle n_{21}, 0, n_{22}\rangle$	$\langle n_3, 2, n_4\rangle$	$\langle n_{10}, 5, \phi\rangle$	$\langle n_{11}, 1, n_{12}\rangle$	$\langle n_{11}, 1, n_{16}\rangle$
EXTENSION	Substitution	T_2^p[1]:	~~$\langle n_{22}, 1, n_{23}\rangle$~~	$\langle n_4, 3, n_5\rangle$	$\langle \phi, 6, \phi\rangle$	~~$\langle n_{12}, 2, n_{13}\rangle$~~ ~~$\langle n_{12}, 2, n_{15}\rangle$~~	~~$\langle n_{16}, 2, n_{17}\rangle$~~
	Insertion	T_2^p[2]:	$\langle n_{22}, 0, n_{23}\rangle$	$\langle n_4, 2, n_5\rangle$	$\langle \phi, 5, \phi\rangle$	$\langle n_{12}, 1, n_{13}\rangle$ $\langle n_{12}, 1, n_{15}\rangle$	$\langle n_{16}, 1, n_{17}\rangle$
	Deletion	T_2^p[0]:	~~$\langle n_{21}, 1, n_{22}\rangle$~~	~~$\langle n_3, 3, n_4\rangle$~~	$\langle \mathbf{n_{10}}, \mathbf{6}, \phi\rangle$	~~$\langle n_{11}, 2, n_{12}\rangle$~~	~~$\langle n_{11}, 2, n_{16}\rangle$~~
T_2^p			$\langle n_4, 3, n_5\rangle$ $\langle \phi, 6, \phi\rangle$ $\langle \mathbf{n_{10}}, \mathbf{6}, \phi\rangle$ $\langle n_{22}, 0, n_{23}\rangle$ $\langle n_4, 2, n_5\rangle$ $\langle \phi, 5, \phi\rangle$ $\langle n_{12}, 1, n_{13}\rangle$ $\langle n_{12}, 1, n_{15}\rangle$ $\langle n_{16}, 1, n_{17}\rangle$			

(i) If $j'' > j'$, n_c'' must be a descendant of n_c'. For strings under node n_c'', $\langle n'', j''\rangle$ is a pivotal entry, and we still keep triple $\langle n'', j'', n_c''\rangle$. Let $n_1, n_2, \ldots, n_m$ denote the nodes on the path from $n_1 = n'$ to $n_m = n''$. We have $\text{ED}(n_1, q[1, j']) = \text{ED}(n_2, q[1, j'+1]) = \ldots = \text{ED}(n_m, q[1, j'']) = x + 1$. Thus $\langle n', j', n_c'\rangle$ is not a pivotal triple and we need to add the following triples into $\mathsf{T}_{x+1}^p[y]$: $\langle n_i, j' + i + 1, n_s\rangle$ for $i \in [1, m-1]$ and $n_s \in n_i - n_{i+1}$ (n_i's children except n_{i+1}).

(ii) If $j'' \leq j'$, n_c'' must be an ancestor of n_c'. Let $n_1, n_2, \ldots, n_m$ denote the nodes on the path from $n_1 = n''$ to $n_m = n'$. We have $\text{ED}(n_1, q[1, j'']) = \text{ED}(n_2, q[1, j''+1]) = \ldots = \text{ED}(n_m, q[1, j']) = x + 1$. Thus $\langle n'', j'', n_c''\rangle$ is not a pivotal triple. As for strings under node n_c', $\langle n', j'\rangle$ is a pivotal entry, we replace triple $\langle n'', j'', n_c''\rangle$ with $\langle n', j', n_c'\rangle$. We also add the following triples into $\mathsf{T}_{x+1}^p[y]$: $\langle n_i, j'' + i + 1, n_s\rangle$ for $i \in [1, m-1]$ and $n_s \in n_i - n_{i+1}$ (n_i's children except n_{i+1}).

(2) Insertion: For the child n_c of node n, we can insert character of n_c after $q[j]$. Thus for each child n_d of n_c, $\langle n_c, j\rangle$ may be a pivotal entry for strings under node n_d. We call function UPDATETRIPLE ($\langle n_c, j, n_d\rangle$) to add triples.

(3) Deletion: We can delete $q[j+1]$ from q. Thus $\langle n, j+1\rangle$ may be a pivotal entry for strings under node n_c. We call function UPDATETRIPLE ($\langle n, j+1, n_c\rangle$) to add triples.

Iteratively we can compute the pivotal triple set and Algorithm 1 shows the pseudo-code. The correctness of the algorithm is formalized in Lemma 7.

Lemma 7: T_x^p computed by our method satisfies (1) completeness: If $\langle n, j, n_c\rangle$ is a pivotal triple, it is in T_x^p; and (2) correctness: If $\langle n, j, n_c\rangle$ is in T_x^p, it is a pivotal triple.

*Example 6: Consider the trie in Figure 2. Table V shows the example to find top-3 answers of query "*srajit*". For $\langle n_1, 1, n_2\rangle$ in T_0^p, we do substitution and add $\langle n_2, 2, n_3\rangle$. For insertion, $\langle n_2, 1, n_3\rangle$ is not a pivotal entry as n_3 matches $q[2]$. Thus we call* FINDPIVOTAL *operation and add $\langle n_3, 2, n_4\rangle$. For deletion, we also extend $\langle n_1, 2, n_2\rangle$ to $\langle n_2, 3, n_3\rangle$. Similarly we compute all pivotal triples in Table V.*

Complexity: Let $|\mathcal{B}|$ denote the number of trie nodes at the $(|q|+\tau)$-th level. As we only keep T_x^p to compute T_{x+1}^p, the space complexity is $\mathcal{O}(\tau|\mathcal{B}|)$. As the update operation (e.g., using a hash table) takes $\mathcal{O}(1)$ time, the worst-case time complexity is $\mathcal{O}(\tau \times |\mathcal{T}|)$. As the method prunes many useless trie nodes, it improves the performance(Section VI-A).

V. A RANGE-BASED METHOD

As there may be multiple triples with the same entry $\langle n, j\rangle$, we want to group them to improve the performance. Consider an entry $\langle n, j\rangle$ in E_x^p. Node n may have multiple children such that $\langle n, j, n_c\rangle$ is a pivotal triple. It is expensive to keep all such triples. For example, $\langle n_1, 1\rangle$ is a pivotal entry for nodes n_2, n_6, n_{11}, and we need to keep three triples $\langle n_1, 1, n_2\rangle$, $\langle n_1, 1, n_6\rangle$, $\langle n_1, 1, n_{11}\rangle$. In addition, it is expensive to enumerate the nodes in $n - n_c$ (the set of children of n except n_c). To address this issue, we propose a range-based method by grouping trie nodes.

We encode the trie structure as follows. For each leaf node, we assign an ID in a pre-order, which is also the ID of its corresponding string. For each internal node n, we maintain an ID range $[l_n, u_n]$, where l_n (u_n) is the minimum (maximum) ID of strings under the node. In Figure 2, the ID range of node n_1 is $[1, 5]$ which denotes all strings with a prefix of n_1 ("s") have IDs in $[1, 5]$ and all IDs in $[1, 5]$ have a prefix "s".

The basic idea of the range-based method is as follows. Consider node n and its child node n_c with label $q[j+1]$. The previous method needs to enumerate all n_c's siblings in $n - n_c$. Instead the range-based method is to use a range $[l, u]$ to denote the nodes in $n - n_c$ and use an integer d to denote $|n|$. Suppose the range of node $n(n_c)$ is $R_n = [l_n, u_n]$($R_{n_c} = [l_{n_c}, u_{n_c}]$). As $R_{n_c} \subseteq R_n$, we use $R_n - R_{n_c} = [l_n, l_{n_c} - 1] \cup [u_{n_c} + 1, u_n]$ to denote the nodes in $n - n_c$. To this end, we propose a concept, called *pivotal quadruple*.

Definition 4 (Pivotal Quadruple): A quadruple $\langle [l, u], d, j\rangle$ is a pivotal quadruple¶, if it satisfies (1) $\langle l, u\rangle$ is a sub-range of

¶The quadruple should be $\langle l, u, d, j\rangle$. For clarity, we use $\langle [l, u], d, j\rangle$.

TABLE VI

AN EXAMPLE FOR TOP-3 SIMILARITY SEARCH "srajit" ON $\mathcal{S}$ USING THE RANGE-BASED METHOD

(a) T_0^r = FINDMATCH$(0,0)$ = $\{\langle[6,6],0,0\rangle, \langle[1,5],1,1\rangle\}$

(b) T_1^r = $\{\langle[1,5],2,2\rangle, \langle[6,6],1,1\rangle, \langle[6,6],0,1\rangle, \langle[1,1],2,3\rangle, \langle[2,5],1,2\rangle, \langle[1,1],3,2\rangle, \langle[2,2],6,5\rangle, \langle[3,4],2,1\rangle, \langle[5,5],7,6\rangle, \langle[6,6],1,0\rangle\}$

T_0^r	$\langle[1,5],1,1\rangle$			$\langle[6,6],0,0\rangle$		
EXTENSION	Substitution $\mathsf{E}_1[0]$	Insertion $\mathsf{E}_1[1]$	Deletion $\mathsf{E}_1[-1]$	Substitution $\mathsf{E}_1[0]$	Insertion $\mathsf{E}_1[1]$	Deletion $\mathsf{E}_1[-1]$
	$\langle[1,5],2,2\rangle$	~~$\langle[1,5],2,1\rangle$~~ $\langle[1,1],3,2\rangle$ $\langle[2,2],6,5\rangle$ $\langle[3,4],2,1\rangle$ $\langle[5,5],7,6\rangle$	~~$\langle[1,5],1,2\rangle$~~ $\langle[1,1],2,3\rangle$ $\langle[2,5],1,2\rangle$	$\langle[6,6],1,1\rangle$	$\langle[6,6],1,0\rangle$	$\langle[6,6],0,1\rangle$
T_1^r	$\langle[1,5],2,2\rangle$, $\langle[6,6],1,1\rangle$, $\langle[6,6],0,1\rangle$, $\langle[1,1],2,3\rangle$, $\langle[2,5],1,2\rangle$, $\langle[1,1],3,2\rangle$, $\langle[2,2],6,5\rangle$, $\langle[3,4],2,1\rangle$, $\langle[5,5],7,6\rangle$, $\langle[6,6],1,0\rangle$					

(c) Computing T_2^r based on T_1^r

T_1^r		$\mathsf{T}_1^r[-1]$	$\langle[1,1],2,3\rangle$	$\langle[2,5],1,2\rangle$	$\langle[6,6],0,1\rangle$	$\mathsf{T}_1^r[0]$	$\langle[1,5],2,2\rangle$	$\langle[6,6],1,1\rangle$
EXTENSION	Substitution	$\mathsf{E}_2[-1]$	~~$\langle[1,1],3,4\rangle$~~ $\langle[1,1],5,6\rangle$	$\langle[2,5],2,3\rangle$	~~$\langle[6,6],1,2\rangle$~~	$\mathsf{E}_2[0]$	~~$\langle[1,5],3,3\rangle$~~ ~~$\langle[3,4],3,3\rangle$~~ $\langle[3,3],4,4\rangle$	$\langle[6,6],2,2\rangle$
	Insertion	$\mathsf{E}_2[0]$	$\langle[1,1],3,3\rangle$	~~$\langle[2,5],2,2\rangle$~~	~~$\langle[6,6],1,1\rangle$~~	$\mathsf{E}_2[1]$	~~$\langle[1,5],3,2\rangle$~~	~~$\langle[6,6],2,1\rangle$~~ $\langle[6,6],3,2\rangle$
	Deletion	$\mathsf{E}_2[-2]$	$\langle[1,1],2,4\rangle$	$\langle[2,5],1,3\rangle$	$\langle[6,6],0,2\rangle$	$\mathsf{E}_2[-1]$	~~$\langle[1,5],2,3\rangle$~~ $\langle[4,4],3,3\rangle$	$\langle[6,6],1,2\rangle$
T_2^r	$\langle[2,5],2,3\rangle$ $\langle[6,6],0,2\rangle$ $\langle[1,1],2,4\rangle$ $\langle[2,5],1,3\rangle$ $\langle[6,6],2,2\rangle$ $\langle[1,1],3,3\rangle$ $\langle[1,1],5,6\rangle$ $\langle[3,3],4,4\rangle$ $\langle[4,4],3,3\rangle$ $\langle[6,6],3,2\rangle$ $\langle[6,6],1,2\rangle$							

T_1^r		$\mathsf{T}_1^r[1]$	$\langle[1,1],3,2\rangle$	$\langle[2,2],6,5\rangle$	$\langle[3,4],2,1\rangle$	$\langle[5,5],7,6\rangle$	$\langle[6,6],1,0\rangle$
EXTENSION	Substitution	$\mathsf{E}_2[1]$	$\langle[1,1],4,3\rangle$	$\langle[2,2],7,6\rangle$	$\langle[3,4],3,2\rangle$	$\langle[5,5],8,7\rangle$	~~$\langle[6,6],2,1\rangle$~~
	Insertion	$\mathsf{E}_2[2]$	$\langle[1,1],4,2\rangle$	$\langle[2,2],7,5\rangle$	$\langle[3,4],3,1\rangle$	$\langle[5,5],8,6\rangle$	$\langle[6,6],2,0\rangle$
	Deletion	$\mathsf{E}_2[0]$	~~$\langle[1,1],3,3\rangle$~~	$\langle[2,2],6,6\rangle$	~~$\langle[3,4],2,2\rangle$~~	$\langle[5,5],7,7\rangle$	~~$\langle[6,6],1,1\rangle$~~
T_2^r	$\langle[1,1],4,3\rangle$ $\langle[2,2],7,6\rangle$ $\langle[3,4],3,2\rangle$ $\langle[5,5],8,7\rangle$ $\langle[2,2],6,6\rangle$ $\langle[5,5],7,7\rangle$ $\langle[6,6],2,0\rangle$ $\langle[1,1],4,2\rangle$ $\langle[2,2],7,5\rangle$ $\langle[3,4],3,1\rangle$ $\langle[5,5],8,6\rangle$						

a d-th level node's range; (2) for any string s with ID in $[l,u]$, ED$(s[1,d+1],q[1,j+1]) \neq$ ED$(s[1,d],q[1,j])$; (3) strings with ID $l-1$ or $u+1$ do not satisfy conditions (1) or (2).

The quadruple $\langle[l,u],d,j\rangle$ means that for each string s in range $[l,u]$, $\langle d,j\rangle$ is a pivotal entry for s and q. Let T_x^r denote the set of pivotal quadruples $\langle[l,u],d,j\rangle$ such that ED$(s[1,d],q[1,j]) = x$ where s is a string with ID in $[l,u]$. We use $\mathsf{T}_x^r[y]$ to denote the subset of T_x^r with $y = d-j$.

For example, consider the trie in Figure 2 and query "srajit". For any string s in [1,5], $D[1][1]\neq D[2][2]$, $\langle[1,5],1,1\rangle$ is a pivotal quadruple. As $D[0][0]=D[1][1]$, $\langle[1,5],0,0\rangle$ is not a pivotal quadruple. Similarly $\langle[6,6],0,0\rangle$ is also a pivotal quadruple. Thus $\mathsf{T}_0^r=\{\langle[1,5],1,1\rangle, \langle[6,6],0,0\rangle\}$.

We still iteratively compute T_x^r from $x=0$. If we find k results from T_x^r, our algorithm terminates.

Algorithm to Compute T_x^r: For $x=0$, from the root r, for each of its children, n_c, we find pivotal quadruples as follows. Suppose n_c is the child of $n=r$ with label $q[1]$. For any string s with ID in $[l_n,u_n]-[l_{n_c},u_{n_c}]$, ED$(s[0],q[0]) \neq$ ED$(s[1],q[1])$. Thus $\langle[l_n,l_{n_c}-1],0,0\rangle$ and $\langle[u_{n_c}+1,u_n],0,0\rangle$ are pivotal quadruples and added into T_0^r [0]. Next for node n_c, we repeat the above step to find pivotal quadruples under node n_c. Iteratively we can compute T_0^r. This is an iterative method and we introduce a function FINDQUADRUPLE(n,j,q) to directly compute the pivotal entries (using parameters $(r,0,q)$).

FINDQUADRUPLE extends FINDTRIPLE by grouping nodes (Figure 4(a)). Algorithm 2 shows the pseudo-code. It first finds the matching nodes $n_1=n,n_2,\cdots,n_m$, where n_m is the last matching node. For each node n_i for $1 \leq i \leq m-1$, its child n_{i+1} matches the query character $q[j+i]$. Instead of enumerating each node in $n_i - n_{i+1}$, we group the siblings of n_{i+1} into two groups based on n_{i+1}: $[l_{n_i}, l_{n_{i+1}}-1]$ and $[u_{n_{i+1}}+1, u_{n_i}]$. FINDQUADRUPLE adds $\langle[l_{n_i}, l_{n_{i+1}}-1], |n_i|, j+i\rangle$ and $\langle[u_{n_{i+1}}+1, u_{n_i}], |n_i|, j+i\rangle$. Similarly for n_m, FINDQUADRUPLE adds $\langle[l_{n_m}, u_{n_m}], n_m, j+m\rangle$.

Next we discuss how to compute T_{x+1}^r based on T_x^r. We propose a new operation QUADRUPLEEXTENSION to support quadruple extensions. For each pivotal quadruple $\langle[l,u],d,j\rangle$ in $\mathsf{T}_x^r[y=d-j]$, it applies the following extensions.

(1) Substitution: For any strings with ID in $[l,u]$, we can substitute the $(d+1)$-th character of these strings with $q[j+1]$. Thus $\langle d+1, j+1\rangle$ may be a pivotal entry for strings in $[l,u]$, $\langle[l,u],d+1,j+1\rangle$ is a potential pivotal quadruple and we want to add it into T_{x+1}^r. However, there may be some strings with the $(d+2)$-th character matching $q[j+2]$. For such strings, $\langle d+1, j+1\rangle$ is not a pivotal entry and $\langle d+2, j+2\rangle$ may be a pivotal entry. To address this issue, we propose a function UPDATEQUADRUPLE ($\langle[l,u],d+1,j+1\rangle$) to find quadruples.

UPDATEQUADRUPLE (Figure 4(b)): Based on the above observation, we want to find the nodes in the $(d+2)$-th level with character $q[j+2]$ within the range $[l,u]$. As there may be multiple such nodes, to accelerate this operation we build an inverted index $\mathcal{I}$, where each entry is an integer d and a character c and the corresponding value is a set of d-th level nodes with label c. Thus we first find all the nodes in the $(d+2)$-th level with character $q[j+2]$ using the index $\mathcal{I}$ and then use a binary search to find the nodes within $[l,u]$. For each of these nodes, n, we use the aforementioned function FINDQUADRUPLE (n, $j+2$, q) to find all possible pivotal quadruples, denoted by $\tilde{\mathsf{T}}_{x+1}^r$.

Notice that the quadruples in $\tilde{\mathsf{T}}_{x+1}^r$ may affect (or be affected by) quadruples in T_{x+1}^r. Thus we need to update quadruples as follows. For each quadruple $\langle[l',u'],d',j'\rangle$ in $\tilde{\mathsf{T}}_{x+1}^r$, let U denote the set of quadruples $\langle[l'',u''],d'',j''\rangle$ in $\mathsf{T}_{x+1}^r[y]$ such that $d''-j''=y$ and $[l'',u'']$ has an overlap with $[l',u']$. If $\mathsf{U}=\phi$, we do not need to update and add $\langle[l',u'],d',j'\rangle$ into $\mathsf{T}_{x+1}^r[y]$; otherwise for each quadruple $\langle[l'',u''],d'',j''\rangle$ in U, we update T_{x+1}^r as follows.

(i) $j''>j'$ and $d''>d'$: For any string s in $[l'',u'']$, as ED$(s[d''],q[j'']) \neq$ ED$(s[d''+1],q[j''+1])$, $\langle d'',j''\rangle$ is still a pivotal entry and we keep $\langle[l'',u''],d'',j''\rangle$. $\langle d',j'\rangle$ is not a pivotal entry as ED$(s[d''],q[j'']) =$ ED$(s[d'],q[j']) = x+1$ and $j''>j'$. We add $\langle[l',l''-1],d',j'\rangle$ and $\langle[u''+1,u'],d',j'\rangle$.

(ii) $j''\leq j'$ and $d''\leq d'$: For any string s in $[l',u']$, $\langle d',j'\rangle$ is a pivotal entry as ED$(s[d'],q[j']) \neq$ ED$(s[d'+1],q[j'+1])$.

Algorithm 2: TOPKRANGESEARCH($\mathcal{S}, q, k$)

Input: $\mathcal{S}$: A string set; q: A query; k: No of answers;
Output:$\mathcal{R}$: A set of top-k answers for $\mathcal{S}$ and q;

```
x = 0 ;
T^r_x = FINDQUADRUPLE(r, 0, q) ;
if |R| ≥ k then return R ;
while true do
    T^r_{x+1} = QUADRUPLEEXTENSION (T^r_x, x) ;
    if |R| ≥ k then return R ;
    ++x ;
```

Function FINDQUADRUPLE(n, j, q)

Input: n: A node; j: An integer; q: A query;
Output: T^r: A set of matching ranges;

```
n_1 = n and n_m is the last matching node ;
for n_i ∈ {n_1, n_2, ..., n_{m-1}} do
    T^r ← ⟨[l_{n_i}, l_{n_{i+1}} − 1], |n_i|, j + i − 1⟩ ;
    T^r ← ⟨[u_{n_{i+1}} + 1, u_{n_i}], |n_i|, j + i − 1⟩ ;
if n_m is a leaf and |q| = j + m then R ← n_m ;
T^r ← ⟨[l_{n_m}, u_{n_m}], |n_m|, j + m⟩ ;
return T^r ;
```

Function QUADRUPLEEXTENSION(T^r_x, x)

Input: T^r_x: A set of entries; x: An integer;
Output: T^r_{x+1}: A set of entries;

```
foreach ⟨[l, u], d, j⟩ ∈ T^r_x do
    T^r_{x+1}=UPDATEQUADRUPLE (⟨[l, u], d+1, j+1⟩);
    T^r_{x+1}=UPDATEQUADRUPLE (⟨[l, u], d + 1, j⟩) ;
    T^r_{x+1}=UPDATEQUADRUPLE (⟨[l, u], d, j + 1⟩) ;
return T^r_{x+1};
```

$\langle d'', j''\rangle$ is not as $\text{ED}(s[d'], q[j']) = \text{ED}(s[d''], q[j'']) = x + 1$ and $j'' \leq j'$. We replace $\langle [l'', u''], d'', j''\rangle$ with $\langle [l', u'], d', j'\rangle$ and add $\langle [l'', l' - 1], d'', j''\rangle$, $\langle [u' + 1, u''], d'', j''\rangle$.

(2) Insertion: For any strings with ID in range $[l, u]$, we can insert the $(d+1)$-th character after $q[j]$. Thus $\langle d + 1, j\rangle$ may be a pivotal entry for strings in $[l, u]$. We call function UPDATEQUADRUPLE ($\langle [l, u], d + 1, j\rangle$) to update quadruples.

(3) Deletion: For any strings with ID in range $[l, u]$, we can delete $q[j + 1]$ from q. Thus $\langle d, j + 1\rangle$ may be a pivotal entry for strings in $[l, u]$. We call function UPDATEQUADRUPLE ($\langle [l, u], d, j + 1\rangle$) to update quadruples.

Iteratively we can compute the pivotal quadruple set and Algorithm 2 shows the pseudo-code. The correctness of the algorithm is formalized in Lemma 8.

Lemma 8: T^r_x computed by our method satisfies (1) completeness: If $\langle [l, u], d, j\rangle$ is a pivotal quadruple, it is in T^r_x; (2) correctness: If $\langle [l, u], d, j\rangle$ is in T^r_x, it is a pivotal quadruple.

Example 7: Consider the trie in Figure 2. Table VI shows the example to find top-3 answers of query "`srajit`*". Consider* $\langle [6, 6], 1, 1\rangle$ *in* T^r_1*. For substitution, we add* $\langle [6, 6], 2, 2\rangle$*. For insertion,* $\langle [6, 6], 2, 1\rangle$ *is not a pivotal entry as the third character of string* s_6*="*`thrifty`*" matches* $q[2]$*=*`'r'`*. Thus we apply the* FINDQUADRUPLE *operation and add* $\langle [6, 6], 3, 2\rangle$*. For deletion, we add* $\langle [6, 6], 1, 2\rangle$*. Similarly we compute all pivotal quadruples as illustrated in Table VI.*

TABLE VII
DATASETS

Datasets	Cardinality	Avg Len	Max Len	Min Len
Word	146,033	16.01	35	1
Author	10.27 million	22.02	383	8
Email	6.4 million	26.58	57	7

Complexity: Let $|\mathcal{B}'|$ denote the number of trie nodes at the $(|q|+\tau-1)$-th level, which is not larger than the number of trie nodes at the $(|q|+\tau)$-th level $|\mathcal{B}|$. As we only keep T^r_x to compute T^r_{x+1}, the space complexity is $\mathcal{O}(\tau \times |\mathcal{B}'|)$. As the update operation (e.g., using a hash table) takes $\mathcal{O}(1)$ time, the worst-case time complexity is $\mathcal{O}(\tau \times |\mathcal{T}|)$. As the method groups many pivotal triples, it improves the performance(Section VI-A).

VI. EXPERIMENTAL STUDY

We conducted an extensive set of experimental studies on three real datasets. The first one is the Author dataset which is a set of author names and extracted from the publications in PubMed ‖. The second one is the Word dataset, which is a set of common English words. The third one is a set of Email addresses. We randomly selected 100 queries from the datasets and compared the average elapsed time. Table VII shows the detailed information of the three datasets. Figure 5 shows the string length distributions of the three datasets.

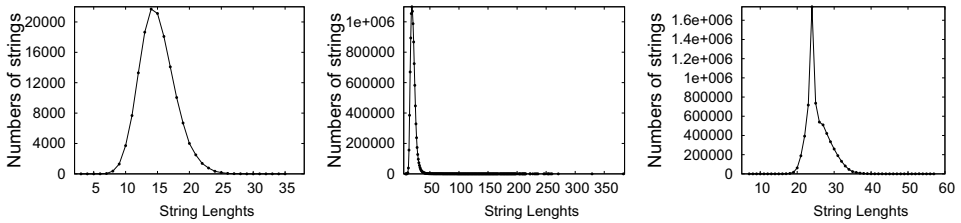
We compared our algorithms with state-of-the-art methods, AQ [20], B^{ed}-Tree [21], and Flamingo [10]. The code of B^{ed}-Tree was provided by the authors. The code of Flamingo was download from their website **. We extended it to support top-k search by increasing the thresholds (initialized as 0). As it is a famous threshold-based string similarity search algorithm, we selected it as a baseline for comparison. AQ was implemented by ourselves in C++. All the algorithms were implemented in C++ and compiled using GCC 4.2.4 with -O3 flag. All the experiments were run on a Ubuntu machine with an Intel Xeon X5670 2.93GHz CPU and 32 GB memory.

A. Evaluation on Our Techniques

In this section, we compare our proposed techniques, the progressive-based method, the pivotal entry based method, and the range-based method. We first compared the number of entries that were needed to compute of the three methods. Figure 6 shows the results by varying k on the three datasets.

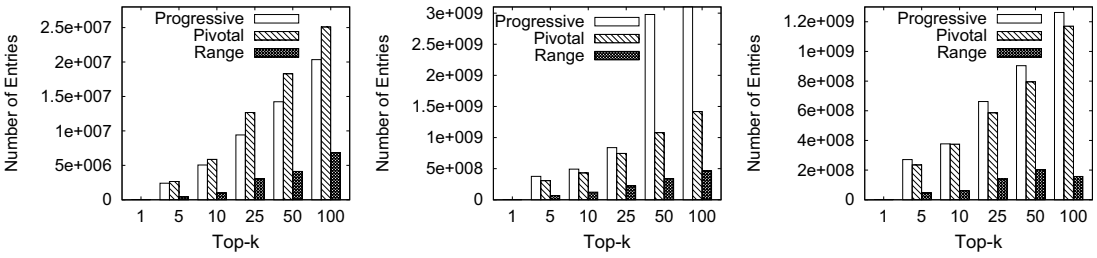
We can see that the pivotal entry based method involved smaller numbers of entries than the progressive-based method on the Email dataset and the Author dataset. This is because the pivotal entry based method only computed pivotal entries and pruned large numbers of useless entries. For example, on the Email dataset, for $k = 50$, the progressive-based method computed 0.8 billion entries, and the pivotal entry based method computed 0.6 billion entries. On the Word dataset, the pivotal entry based method was worse than the progressive-based method. This is because if an entry is a pivotal entry for all children, the pivotal entry based method

‖ http://www.ncbi.nlm.nih.gov/pubmed/
** http://flamingo.ics.uci.edu/



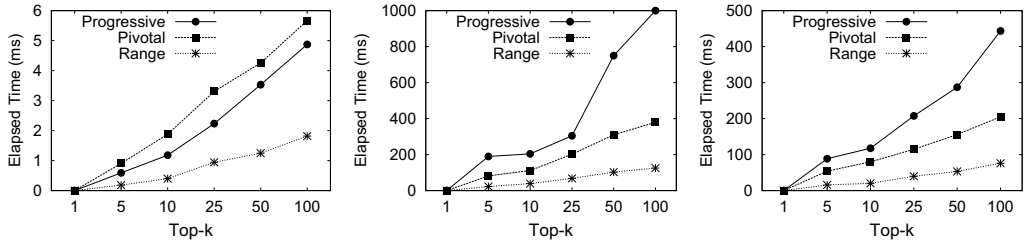
(a) The Word Dataset (b) The Author Dataset (c) The Email Dataset

Fig. 5. String length distribution



(a) The Word Dataset (b) The Author Dataset (c) The Email Dataset

Fig. 6. Evaluating Our Techniques - Number of Entries



(a) The Word Dataset (b) The Author Dataset (c) The Email Dataset

Fig. 7. Evaluating Our Techniques - Elapsed Time

needed to maintain all such triples which may be expensive. However the range-based method grouped such triples and significantly reduced the number of entries.

In addition, the range-based method computed much smaller numbers of entries than the pivotal entry based method and the progressive-based method on the three datasets. The main reason is that the range-based method grouped large numbers of pivotal entries and reduced the number of pivotal entries significantly. For example, on the Email dataset, for $k = 100$, the pivotal entry based method and the progressive-based method involved more than 1 billion entries, and the range-based method had only 0.1 billion entries. The experimental results consist with our analysis in Section IV.

Next we compare the average elapsed time of different methods by varying k. Figure 7 shows the results.

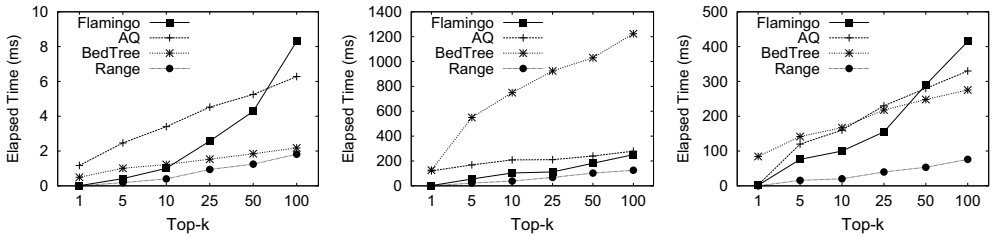
We can see that the range-based method achieved the best performance and outperformed the other two methods. The main reason is that, the range-based method pruned many non-pivotal entries against the progressive-based method and grouped the pivotal entries to avoid unnecessary computations. For example, on the Email dataset, for $k = 50$, the progressive-based method took 300 milliseconds, the pivotal entry based method improved the time to 150 milliseconds, and the range-based method further reduced time to 50 milliseconds. This shows that our range-based pruning technique can prune large numbers of unnecessary entries and improve the performance significantly.

B. Comparison with Existing Methods

In this section, we compare our range-based method with state-of-the-art methods, AQ, B^{ed}-Tree, and Flamingo by varying different k on the three datasets. As they needed tune some parameters (e.g., gram length), we reported their best results. Figure 8 shows the experimental results.

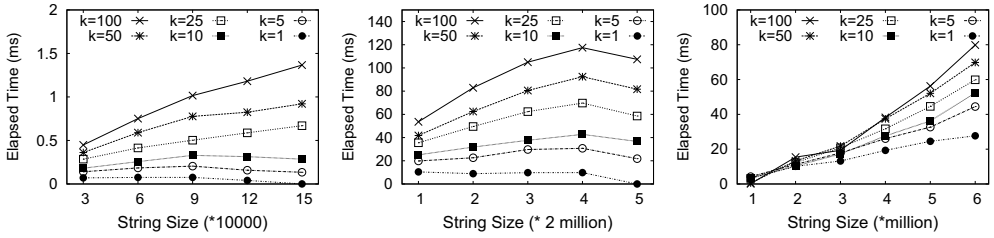
We can see that for small k values ($k < 50$), AQ had the worst performance as it is rather time consuming to adaptively select a good gram length. For large k values, AQ was better than Flamingo as the search time was larger than the time to select a good gram length. For example, on the Word dataset, for $k = 25$, AQ took 5 milliseconds and other methods took less than 3 milliseconds. In addition, B^{ed}-Tree was better than AQ and Flamingo for large k values, as it dynamically updated the threshold and used the threshold to do pruning. However it had low pruning power for small k values. This is because for small k values, it may scan many irrelevant strings and cannot use a tighter bound to do pruning.

Our method achieved the highest performance and outperformed existing methods. This is because B^{ed}-Tree only used the string level pruning, while our method can utilize the character-level pruning. That is for a string, B^{ed}-Tree can only take its edit distance to the query as a threshold. Our method can progressively compute edit distance and can use the edit distance of prefixes of a string and the query as a threshold. Thus our method outperformed B^{ed}-Tree. For example, on the Email dataset, for each k value, our method



(a) The Word Dataset (b) The Author Dataset (c) The Email Dataset

Fig. 8. Comparison with Existing Methods



(a) The Word Dataset (b) The Author Dataset (c) The Email Dataset

Fig. 9. Scalability of Our Method

achieved the best performance. For $k = 100$, Flamingo took more than 400 milliseconds, B^{ed}-Tree improved the time to 300 milliseconds, and our method only took less than 80 milliseconds. The results show the superiority of our progressive based framework and our pivotal-entry-based and range-based pruning techniques.

C. Scalability

In this section, we evaluate the scalability of our range-based method. We varied the number of strings and evaluated our method for finding top-k answers. Figure 9 shows the results on the three datasets. We can see that with the datasets increased, our method scaled very well for different k values. For example, on the Email dataset, for k=100, our method took 27 milliseconds for 1 million strings, and the time increased to 52 milliseconds for 3 million strings and 79 milliseconds for 6 million strings. Note that on the Author dataset the elapsed time for 10 million strings was smaller than that for 8 million strings. This is because a large string set may have more possible answers and thus it may lead to early termination for finding top-k answers.

VII. Conclusion

In this paper, we have studied the problem of top-k string similarity search. We proposed a progressive framework to support top-k similarity search. We proposed pivotal entries to avoid unnecessary computations which can prune large numbers of useless entries. We extended this technique to support similarity search. We devised a range-based method by grouping the pivotal entries which can further reduce the number of entries. Experimental results show that our method significantly outperforms existing methods.

Acknowledgment

This work was partly supported by the National Natural Science Foundation of China under Grant No. 61003004 and No. 61272090, National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, and a Tsinghua project under Grant No. 20111081073.

References

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[2] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[3] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, pages 313–324, 2003.

[4] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–16, 2006.

[5] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[6] M. Hadjieleftheriou, N. Koudas, and D. Srivastava. Incremental maintenance of length normalized indexes for approximate string matching. In *SIGMOD Conference*, pages 429–440, 2009.

[7] J. Jestes, F. Li, Z. Yan, and K. Yi. Probabilistic string similarity joins. In *SIGMOD Conference*, pages 327–338, 2010.

[8] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 433–439, 2009.

[9] T. Kahveci and A. K. Singh. Efficient index structures for string databases. In *VLDB*, pages 351–360, 2001.

[10] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[11] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[12] G. Li, S. Ji, C. Li, and J. Feng. Efficient fuzzy full-text type-ahead search. *VLDB J.*, 20(4):617–640, 2011.

[13] G. Li, J. Wang, C. Li, and J. Feng. Supporting efficient top-k queries in type-ahead search. In *SIGIR*, pages 355–364, 2012.

[14] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[15] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.

[16] J. Wang, G. Li, and J. Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *VLDB*, 1219–1230, 2010.

[17] J. Wang, G. Li, and J. Feng. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *ICDE*, pages 458–469, 2011.

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

[19] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.

[20] Z. Yang, J. Yu, and M. Kitsuregawa. Fast algorithms for top-k approximate string matching. In *AAAI*, 2010.

[21] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD Conference*, pages 915–926, 2010.