Received May 6, 2020, accepted May 20, 2020, date of publication May 25, 2020, date of current version June 5, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2996945

Optimized Signature Selection for Efficient String Similarity Search

TAEGYOUNG LEE[1], TAE-SUN CHUNG[2], AND JONGIK KIM[3]
[1]Department of Computer Science and Engineering, The Hong Kong University of Science and Technology (HKUST), Hong Kong
[2]Department of Computer Engineering, Ajou University, Suwon 16499, South Korea
[3]Division of Computer Science and Engineering, Jeonbuk National University, Jeonju 54896, South Korea

Corresponding author: Jongik Kim (jongik@jbnu.ac.kr)

This work was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Korean Government (Ministry of Science and ICT) under Grant 2019R1F1A1059795 and Grant 2019R1F1A1058548.

ABSTRACT In this paper, we study the problem of string similarity search to retrieve in a database all strings similar to a query string within a given threshold. To measure the similarity between strings, we use edit distance. Many algorithms have been proposed under a filtering-and-verification framework to solve the problem. To reduce the overhead of edit distance verification, it is crucial to efficiently generate a small number of candidates in the filtering phase. Recently, an index structure named HSTree has been proposed for efficiently generating candidate strings. To generate candidates, they select and utilize HSTree nodes at a specific level calculated from a given threshold. In this paper, we observe that there are many alternative ways to select HSTree nodes, and propose a novel technique that selects HSTree nodes in an optimized way based on the observation. We also propose a modified HSTree, named a threaded HSTree, which connects inverted lists of an HSTree node to inverted lists of its child nodes. With a threaded HSTree, we can reduce the overhead of index lookups in HSTree nodes while selecting optimal tree nodes. Experimental results show that the proposed technique significantly outperforms the existing technique using the HSTree.

INDEX TERMS Edit distance, hierarchical tree index, optimized signature selection, partition signature scheme, string similarity search.

I. INTRODUCTION

Finding similar objects is essential in data analytics, and many similarity measures have been developed for different types of data. For example, SimRank [13] and its variants [26], [37], [47], [49], [50], [52] have been proposed to measure the similarity between objects in an information network; common subgraphs [5], [36], missing edges and features [51], [53], and graph edit distance [15], [32], [34] have been developed to quantify the similarity between complex objects that are represented by graph models; Jaccard [12], Cosine, and Dice [9] similarities are commonly used for set data; and LSA [19] have been developed to measure similarity between documents through corpus analysis.

In this paper, we focus on syntactic similarity between unstructured text data. Because text data are abundant, and typographical errors and different representations of text data cannot be avoidable, finding syntactically similar strings is an fundamental operation required in a wide range of applications including data cleaning [8], query relaxation [33], DNA read mapping [17], [18], and near duplicate detection [45]. To measure the similarity between two strings, we use edit distance [11], [27], [28], [38], which is the minimum number of edit operations (insertion, deletion, and substitution) to transform one string to the other. Edit distance has the following advantages over alternative measure: it reflects the ordering of characters in the string and it allows non-trivial alignment. These properties enable edit distance to capture typographical errors for text documents, and to capture similarities for Homologous proteins or genes [29], [43].

The problem of string similarity search studied in this paper is to retrieve all strings in a string database whose edit distance to a query string is within a given threshold. This is a challenging problem, because edit distance computation is costly and a scan-based approach that computes the edit distance for each string in the database would incur a prohibiting $O(N \cdot n^2)$ cost for a large database, where N is the number of strings in the database and n is the average length of a

The associate editor coordinating the review of this manuscript and approving it for publication was Qichun Zhang.

string. To address the performance challenge, there has been a rich literature on this problem [3], [4], [6], [7], [16], [18], [20]–[23], [29], [39], [41], [46], [48].

All existing techniques adopt a filtering-and-verification framework, with a main focus on the filtering phase to reduce the overhead of edit distance computation in the verification phase. To effectively generate candidate strings by filtering out strings dissimilar to a query string, they utilize signatures extracted from data strings. The most widely used signature scheme is q-gram, which is a substring of a string with length q. Given two strings with an edit distance threshold, a necessary condition to meet the threshold is established on the minimum number of q-grams contained in both strings. To efficiently generate candidate strings using the q-gram signature scheme, all existing algorithms utilize an inverted index built on data strings. They extract q-grams from data strings, and make an inverted list on each q-gram, which is a list of ids of strings that contain the q-gram. Then, they build an index that maps each q-gram to its corresponding inverted list. Early work (e.g., [20], [21]) extracts overlapping q-grams from a query string, and using an inverted index, generates those data string that shares enough number of q-grams with the query string. Later work (e.g., [14], [17]) selects non-overlapping q-grams from a query string to reduce the number of candidate strings.

A drawback of the q-gram signature scheme is that there can be many strings that share a q-gram, because q is usually chosen to be a small value (e.g. from 2 to 4) to support various thresholds. As a result, a large number of candidates can be generated in the filtering phase. To overcome the limitation, the partition signature scheme has been proposed [22], [23]. The partition signature scheme establishes a filtering condition using the pigeonhole principle as follows. Given two strings r and s with a threshold τ, if we decompose r into $\tau + c$ partitions, i.e., disjoint substrings, at least c partitions should be contained in s to meet the threshold. Since we can use large signatures (especially when $c = 1$), the partition signature scheme has been found to be much more efficient than the q-gram scheme. However, an offline index cannot be built on partitions because the number of partitions is determined by the threshold, which is specified when a query is issued. Therefore, it is not suitable to the string similarity search problem. This scheme is proposed to solve the string similarity join problem, where an index is built on-the-fly during join processing.

Recently, a hierarchical index structure, named HSTree, has been proposed to apply the partition signature scheme to string similarity search problem [39], [48]. The HSTree is a full binary tree. At the i^{th} level of the tree, each data string is decomposed into 2^i partitions, where the j^{th} partition is indexed in the j^{th} node of the i^{th} level (see Section II-C for the details of the HSTree index). Given a query string with a threshold τ, it selects the lowest level having at least $\tau + 1$ nodes (or partitions) to use the pigeonhole principle. As HSTree can use the partition signature scheme in the search problem, it exhibits good performance. It is also easily used to support top-k similarity queries.

Although we can use the partition signature scheme with HSTree, this approach has the following limitation. We only use the tree at a specific level, which is determined by a threshold. In partition-based approach, we need at least $\tau + 1$ partitions to use the pigeonhole principle. Since HSTree selects nodes in a specific level, the number of partitions used for a query is not consistently determined.

Example 1: Consider we have a string "SIMILARITY" in our string database. Figure 1 depicts how the string is partitioned into each HSTree node. For a query with a threshold $\tau = 2$, the tree nodes at the second level is selected and $\tau + 2 = 4$ partitions of the string is used to check if at least $c = 2$ partitions are contained in the query string. When $\tau = 4$, the tree nodes at the third level is selected and $\tau + 4 = 8$ partitions of the string is used to check if at least $c = 4$ partitions are contained in the query string.

In Example 1, we have to select $\tau + 2$ partitions for $\tau = 2$, while we should select $\tau + 4$ partitions for $\tau = 4$. When using the pigeonhole principle with $\tau + c$ partitions, there is a trade-off between filtering and verification costs for different c values (see Section III-A for the details), but HSTree cannot balance the trade-off because c value is determined by the threshold τ as shown in the example above.

To address the limitation, we propose a partition selection algorithm that selects $\tau + c$ partitions, i.e., HSTree nodes, for a fixed value c regardless of τ. We observe that we can select partitions from different levels of HSTree. For example, consider we are given a fixed $c = 1$. When $\tau = 2$, we can select $\tau + c = 3$ partitions "SIMIL" at level 1, and "AR" and "ITY" at level 2 in Figure 1. If $\tau = 4$, we can select $\tau + c = 5$ partitions "SI", "MIL", and "AR" at level 2 and "I" and "TY" at level 3. Interestingly, there are many alternative ways to select a given number of partitions. When $\tau = 2$, for example, we can select alternative combinations of $\tau + c = 3$ partitions: {"SIMIL", "AR", "ITY"} or {"SIM", "IL", "ARITY"}. Based on the observation, we propose a novel dynamic programming algorithm that selects an optimal combination of a given number of partitions that generates the minimum number of candidates.

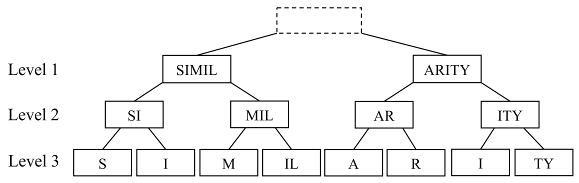


FIGURE 1. "SIMILARITY" partitioned into HSTree nodes.

The following summarize the contributions of the paper

- We show that there are many alternative combinations of HSTree nodes to evaluate a query, and develop a novel dynamic programming algorithm that select an optimal combination of nodes.

- We propose an enhanced HSTree, named a threaded HSTree, that connects inverted lists of an HSTree node to inverted lists of its child nodes. With a threaded HSTree, we can reduce the overhead of index lookups in HSTree nodes while selecting optimal tree nodes.
- We implement the proposed algorithm and conduct extensive experiments on real datasets. From our experimental results, we show that the proposed algorithm significantly outperforms the existing algorithm that uses the HSTree.

The rest of this paper is organized as follows. In Section II, we formulate the problem of string similarity search, describe the candidate generation method based on the partition signature scheme and review the HSTree index structure. In Section III, we propose a novel dynamic programming algorithm to select an optimal combination of tree nodes. In Section IV, we enhance an HSTree to reduce the overhead of index lookups. In Section V, we report our experimental results on real datasets. We brief related work in Section VI and conclude the paper in Section VII.

II. PROBLEM FORMULATION AND PRIOR WORK

A. PROBLEM FORMULATION

The edit distance between two strings r and s, denoted by $\mathsf{ed}(r, s)$, is the minimum number of edit operations to transform r into s or vice versa. An edit operation is insertion, deletion, or substitution of a single character. For example, ed(“`string`”, “`strong`”) is 1 because “`string`” can be transformed into “`strong`” by substituting one character ‘`i`’ with ‘`o`’.

Definition 1: Given a string database $\mathcal{D}$, and a query string q with an edit distance threshold τ, the problem of string similarity search is to retrieve from $\mathcal{D}$ all strings s such that $\mathsf{ed}(q, s) \leq \tau$.

Example 2: For the strings in Table 1, consider a string database $\mathcal{D} = \{s_1, s_2, \ldots, s_8\}$. Given a query string q = “`string`” with a threshold 1, the result of the similarity search is $\{s_1, s_2, s_3\}$.

B. DISJOINT SIGNATURE-BASED APPROACH FOR GENERATING CANDIDATES

We can establish a necessary condition between two strings to meet a threshold using the pigeonhole principle. The following definition and lemma formally state the necessary condition.

Definition 2: Given a string s, consider two substrings $s[p_1, l_1]$ and $s[p_2, l_2]$ of s, where $s[p, l]$ denotes a substring of s starting at position p with length l. Without loss of generality, we assume that $p_1 < p_2$. The substrings $s[p_1, l_1]$ and $s[p_2, l_2]$ are disjoint, if and only if $p_1 + l_1 \leq p_2$.

Disjoint substrings in the definition above does not share any character in a common position. For a string “`abcde`”, for example, “`abc`” and “`de`” are disjoint, but “`abc`” and “`cd`” are not disjoint.

Lemma 1: Given two strings r and s and a threshold τ, if we extract $\tau + c$ disjoint segments from r, where c is a constant, s should contain at least c segments of r to meet the threshold.

A disjoint segment of r contained in s is called a *matching segment*. The lemma above states that we need at least c matching segments to meet the threshold. The intuition behind the lemma is that a mismatching segment causes at least one edit operation, and edit operations caused by different segments are independent. If the number of matching segments is less than c, the number of mismatching segments is greater than τ, and thus r and s cannot meet the threshold τ.

Example 3: Consider two strings s_5 and s_6 in Table 1. Given a threshold $\tau = 2$, if we extract $\tau + 2$ disjoint segments, “`al`”, “`on`”, “`ene`”, and “`ss`” from s_5, only one of the segments, i.e., “`al`”, is contained in s_6. Hence, s_5 and s_6 cannot meet the threshold by Lemma 1

TABLE 1. An example string collection.

ID	String	Length
s_1	`spring`	6
s_2	`strong`	6
s_3	`strung`	6
s_4	`strike`	6
s_5	`aloneness`	9
s_6	`alinement`	9
s_7	`apartment`	9
s_8	`amusement`	9

In the q-gram signature scheme, existing techniques select non-overlapping q-grams to generate candidates using Lemma 1. However, those techniques cannot utilize string segments between the selected q-grams, and filtering power is limited because the signature size (i.e., q) is small. PassJoin [22], [23] has been proposed to find similar strings using partition-based signatures for Lemma 1. PassJoin decomposes a string s into $\tau + c$ disjoint segments[1] $w_1, w_2, \ldots, w_{\tau+c}$, such that $s = w_1 \cdot w_2 \cdot \ldots \cdot w_{\tau+c}$, where $w_i \cdot w_{i+1}$ denotes the concatenation of w_i and w_{i+1}. We call w_i a partition of s. By using a partition-based signature, which is longer than a q-gram signature, the number of candidates can be reduced since the longer a signature is, the less strings that share the signature. In the remaining of this subsection, we briefly introduce the technique in PassJoin.

Given a string database $\mathcal{D}$ and a threshold τ, an index is built on strings in $\mathcal{D}_l = \{s \mid s \in \mathcal{D} \wedge |s| = l\}$ as follows. Each string in $\mathcal{D}_l$ is partitioned into $\tau + c$ segments[2]. For the j^{th} segments of the strings in $\mathcal{D}_l$, we make an inverted index $\mathcal{L}_l^j$ that maps each distinct j^{th} segment w to $\mathcal{L}_l^j(w)$, which is a list of ids of strings that have w in their

[1]We note that PassJoin [22], [23] uses $\tau + 1$ signatures. In our discussion here, we generalize it to $\tau + c$ signatures based on Lemma 1 for the easy of description in the following section.

[2]There are many ways to partition a string into $\tau + c$ segments. PassJoin uses an even-partition scheme, where each segment should have nearly the same length. Refer to [22], [23] for the details.

j^{th} segments. In this way, an index $\mathcal{I}_l = \{\mathcal{L}_l^j \mid 1 \leq j \leq \tau + c\}$ is built for the strings in $\mathcal{D}_l$. The index for $\mathcal{D}$ is $\mathcal{I} = \{\mathcal{I}_l \mid l$ is a distinct length of strings in $\mathcal{D}\}$.

Example 4: For the string collection $\mathcal{D}$ in Table 1, consider that $\tau = 1$ and $c = 1$ are given. To make $\mathcal{I}_6$, we decompose each string in $\mathcal{D}_6 = \{s_1, s_2, s_3, s_4\}$ into two partitioned segments with the same length 3. We can construct $\mathcal{I}_9$ for $\mathcal{D}_9$ in a similar way. Figure 2 shows $\mathcal{I}_6 = \{\mathcal{L}_6^1, \mathcal{L}_6^2\}$ and $\mathcal{I}_9 = \{\mathcal{L}_9^1, \mathcal{L}_9^2\}$. Therefore, the index for $\mathcal{D}$ is $\mathcal{I} = \{\mathcal{I}_6, \mathcal{I}_9\}$.

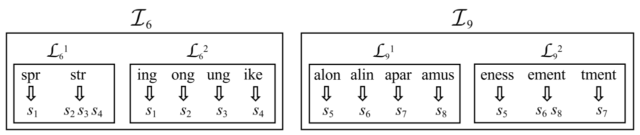


FIGURE 2. Example inverted indices $\mathcal{I}_6$ and $\mathcal{I}_9$ for the string collection in Table 1.

Before we describe how to evaluate a query using the index, we present an obvious necessary condition between two strings to meet a threshold. The following lemma states a condition on the size difference of two strings.

Lemma 2: Given two strings r and s with a threshold τ, if $\mathsf{ed}(r, s) \leq \tau$, then $||r| - |s|| \leq \tau$.

To evaluate a query string q, we need to search the index from $\mathcal{I}_{|q|-\tau}$ to $\mathcal{I}_{|q|+\tau}$ by Lemma 2. For each inverted index $\mathcal{L}_l^j$ in $\mathcal{I}_l$ ($|q| - \tau \leq l \leq |q| + \tau$, $1 \leq j \leq \tau + c$), we first compute substrings of q, denoted by $\mathcal{W}(q, \mathcal{L}_l^j)$, to look up $\mathcal{L}_l^i$. Let the length of segments indexed in $\mathcal{L}_l^j$ be ℓ_l^j. Note that all segments in $\mathcal{L}_l^j$ have the same length (e.g., $\ell_9^2 = 5$ in Example 4). To find all segments in $\mathcal{L}_l^j$ that are contained in q, we compute $\mathcal{W}(q, \mathcal{L}_l^j) = \{w \mid w$ is a substring of q of length $\ell_l^j\}$ and take union of $\mathcal{L}_l^j(w)$'s for all $w \in \mathcal{W}(q, \mathcal{L}_l^j)$. Let the result list of the union be $\mathcal{R}_l^j(q)$. The set containing all candidate strings in $\mathcal{I}_l$ can be obtained by merging $\mathcal{R}_l^j(q)$'s for all $j \in [1, \tau + c]$ and choosing those strings that are contained at least c result lists by Lemma 1.

Example 5: Given the index depicted in Figure 2, consider a query string q = "`aparment`" with a threshold $\tau = 1$, where $c = 1$. Since $|q| = 8$, we need to search $\mathcal{I}_7$, $\mathcal{I}_8$, and $\mathcal{I}_9$ by Lemma 2. Since $\mathcal{I}_7 = \mathcal{I}_8 = \emptyset$, we only look up $\mathcal{I}_9$ as follows. Since $\ell_9^1 = 4$ (for $\mathcal{L}_9^1$), each string in $\mathcal{W}(q, \mathcal{L}_9^1) = \{$`apar`, `parm`, `arme`, `rmen`, `ment`$\}$ searches $\mathcal{L}_9^1$, and $\mathcal{R}_9^1 = \{s_7\}$ is obtained since `apar` in $\mathcal{W}(q, \mathcal{L}_9^1)$ is indexed in $\mathcal{L}_9^1$. To search $\mathcal{L}_9^2$, we enumerate $\mathcal{W}(q, \mathcal{L}_9^2) = \{$`aparm`, `parme`, `armen`, `rment`$\}$ since $\ell_9^2 = 5$ (for $\mathcal{L}_9^2$). $\mathcal{R}_9^2 = \emptyset$ because no string in $\mathcal{W}(q, \mathcal{L}_9^2)$ is indexed in $\mathcal{L}_9^2$. We merge $\mathcal{R}_9^1 = \{s_7\}$ and $\mathcal{R}_9^2 = \emptyset$ and find a candidate s_7, since s_7 is contained in c (= 1) result list.

Once candidate strings are generated, each of candidate is verified by computing the edit distance to the query string. Intuitively, the smaller size $\mathcal{W}(q, \mathcal{L}_l^j)$, the smaller number of candidates. By utilizing various conditions (e.g., a condition on position difference between segments), the number of segments in $\mathcal{W}(q, \mathcal{L}_l^j)$ can be substantially reduced (see Section II-C).

C. HSTree

The partition-based approach described in Section II-B can build an index only with a given static threshold. Therefore, it is hard to apply the approach to the search problem, where a threshold can vary from query to query. To address the problem, HSTree [39], [48] has been proposed, which maintains alternative partitioning results of each data string.

For each distinct string length l, an HSTree $\mathcal{H}_l$ is built on $\mathcal{D}_l$ as follows. At the level i of the tree[3], $\mathcal{H}_l$ partitions each data string $s \in \mathcal{D}_l$ into 2^i segments, and the j^{th} segment of s is indexed in the inverted index of the j^{th} node, denoted by $\mathcal{L}_l^{i,j}$, just like the partition-based approach in the previous section. For the purpose of presentation, we use the inverted index in a node interchangeably with the node itself in the rest of the paper. Similar to PassJoin, HSTree also use an even-partition scheme. Specifically, for each segment w at the level i, w is partitioned into two disjoint segments in the $(i+1)^{th}$ level, such that the first segment is the prefix of w of length $\lfloor |w|/2 \rfloor$ and the second segment is the suffix of w of length $\lceil |w|/2 \rceil$. For example, Figure 3 shows an HSTree $\mathcal{H}_9$ for the string collection in Table 1.

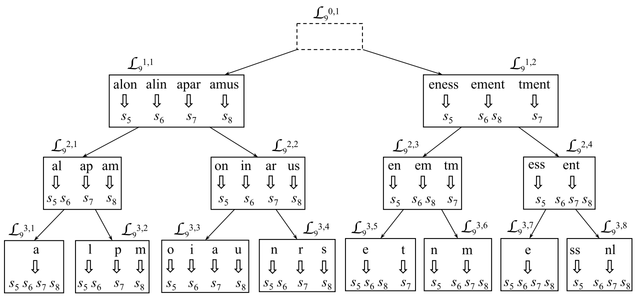


FIGURE 3. $\mathcal{H}_9$ for the string collection in Table 1.

Given a query q with a threshold τ, we can evaluate the query using the HSTree between $\mathcal{H}_{|q|-\tau}$ and $\mathcal{H}_{|q|+\tau}$ by Lemma 2. To generate candidate strings, we need at least $\tau + 1$ partitions of strings by Lemma 1. For each $\mathcal{H}_l$ ($|q| - \tau \leq l \leq |q| + \tau$), we select the lowest level having at least $\tau + 1$ nodes. Therefore, $i = \lceil \log_2(\tau + 1) \rceil$. Given 2^i nodes, i.e., $\mathcal{L}_l^{i,j}$ ($1 \leq j \leq 2^i$), the query can be evaluated as the similar way introduced in the previous section. A segment set of the query string q for searching an index $\mathcal{L}_l^{i,j}$, denoted by $\mathcal{W}(q, \mathcal{L}_l^{i,j})$, is computed using the following position lower bound and upper bound (see PassJoin [22], [23] for the details).

$$\begin{aligned} \mathsf{LB_T} = \max(1,\ & p(\mathcal{L}_l^{i,j}) - (j-1), \\ & p(\mathcal{L}_l^{i,j}) + \Delta - (\tau + c - j)), \end{aligned} \tag{1}$$

$$\begin{aligned} \mathsf{UB_T} = \min(&|q| - \ell(\mathcal{L}_l^{i,j}) + 1, p(\mathcal{L}_l^{i,j}) + (j-1), \\ & p(\mathcal{L}_l^{i,j}) + \Delta + (\tau + c - j)), \end{aligned} \tag{2}$$

where $\ell(\mathcal{L}_l^{i,j})$ and $p(\mathcal{L}_l^{i,j})$ denote the length and the position of the segments in $\mathcal{L}_l^{i,j}$, respectively, and $\Delta = |q| - l$. In the

[3]The root node of an HSTree is at the level 0.

formulas above, $j-1$ and $\tau+c-j$ indicate the relative locations[4] of a partition from the left-side and the right-side perspectives, respectively. Given the $\mathsf{LB_T}$ and $\mathsf{UB_T}$,

$$\mathcal{W}(q, \mathcal{L}_l^{i,j}) = \{q[p, \ell(\mathcal{L}_l^{i,j})] \mid p \in [\mathsf{LB_T}, \mathsf{UB_T}]\}, \tag{3}$$

where $q[p, \ell(\mathcal{L}_l^{i,j})]$ denotes a substring of q starting at position p with length $\ell(\mathcal{L}_l^{i,j})$. Since the level i has 2^i nodes, $\tau + c = 2^i$ and $c = 2^i - \tau$ in Lemma 1. Hence, we generate those strings whose segments are selected from at least $2^i - \tau$ nodes at the level i, while searching the index with $\mathcal{W}(q, \mathcal{L}_l^{i,j})$ for $1 \leq j \leq 2^i$.

Example 6: Consider a query string q = "`alignment`" with a threshold $\tau = 1$ for the string collection in Table 1. Since $|q| = 9$, we need to search $\mathcal{H}_8$, $\mathcal{H}_9$, and $\mathcal{H}_{10}$. As $\mathcal{H}_8 = \mathcal{H}_{10} = \emptyset$, we search $\mathcal{H}_9$ depicted in Figure 3. In $\mathcal{H}_9$, we select the level $\lceil \log_2 \tau + 1 \rceil = 1$ and search $\mathcal{L}_9^{1,1}$ and $\mathcal{L}_9^{1,2}$. To search $\mathcal{L}_9^{1,1}$, we compute $\mathcal{W}(q, \mathcal{L}_9^{1,1}) = \{\texttt{alig}\}$ ($\mathsf{LB_T} = 1$ and $\mathsf{UB_T} = 1$ because $p(\mathcal{L}_9^{1,1}) = 1$, $\Delta = 0$, and $\ell(\mathcal{L}_9^{1,1}) = 4$). Since $\mathcal{L}_9^{1,1}$ does not contain `alig`, we generate no candidate string in this node. Next, we search $\mathcal{L}_9^{1,2}$ with $\mathcal{W}(q, \mathcal{L}_9^{1,2}) = \{\texttt{nment}\}$ ($\mathsf{LB_T} = 5$ and $\mathsf{UB_T} = 5$ because $p(\mathcal{L}_9^{1,2}) = 5$, $\Delta = 0$, and $\ell(\mathcal{L}_9^{1,2}) = 5$). $\mathcal{L}_9^{1,2}$ does not contain `nment`, and we generate no candidate in $\mathcal{L}_9^{1,2}$ either. Hence, the result is $\emptyset$.

III. OPTIMIZED HSTree NODE SELECTION

A. MOTIVATION OF OUR WORK

In general, the quality of a partition signature for generating candidates is assumed to be proportional to the size of the signature. By choosing $c = 1$ in Lemma 1, we can maximize the size of each partitioned segment, and thus expect that the number of candidates generated from each partition is minimized. For this reason, PassJoin [22], [23] uses $\tau + 1$ scheme. If we use a larger c value, the size of each partition signature is reduced, and thus the inverted list for the signature becomes longer. Nevertheless, we have a stricter filtering condition, since a candidate requires to have at least c partitioned segments contained in a query. To generate candidates, however, we need to scan and merge more and longer inverted lists, which can degrade the performance of the search. Therefore, c in Lemma 1 can be also used as a tunable parameter (e.g. [14], [17], [18]) to balance filtering and verification costs.

In HSTree, the c value is dependent on τ, that is, $c = 2^i - \tau$ where $i = \lceil \log_2(\tau + 1) \rceil$. For example, we have to use $\tau + 1$ scheme for $\tau = 3$, while we should use $\tau + 4$ scheme for $\tau = 4$. This is because we select nodes in a specific level of the tree based on a given threshold. Since the c value is determined by τ, we cannot expect consistent performance for different τ values, and we have no chance for improving performance by adjusting c. To address the problem, for a given fixed c value, we propose a novel technique that selects optimal $\tau + c$ disjoint nodes across different levels in an HSTree, where any two nodes are disjoint if and only if they do not lie on the same root-to-leaf path of the tree. Note that if two HSTree nodes are disjoint, the substrings of a data string corresponding the nodes are also disjoint. Given $\tau + c$ disjoint nodes, therefore, we can apply Lemma 1 to generate candidate strings.

[4]The position of a segment starts from 1, while the relative location of a partition starts from 0.

Example 7: Consider a query string $q = \texttt{alignment}$ for $\tau = 4$ for $\mathcal{H}_9$ in Figure 3. In the original HSTree technique, we select the level 3 and c is determined to $2^3 - 4 = 4$. For a given $c = 1$, however, we can consider tree nodes from all levels to select $\tau + 1 = 5$ disjoint nodes. For example, we can select $\{\mathcal{L}_9^{1,1}, \mathcal{L}_9^{3,5}, \mathcal{L}_9^{3,6}, \mathcal{L}_9^{3,7}, \mathcal{L}_9^{3,8}\}$ at the levels 1 and 3. Alternatively, we can also select $\{\mathcal{L}_9^{2,1}, \mathcal{L}_9^{3,3}, \mathcal{L}_9^{3,4}, \mathcal{L}_9^{2,3}, \mathcal{L}_9^{2,4}\}$ at the level 2 and 3. There are many other combinations of $\tau + 1$ disjoint nodes in this case.

As shown in the example above, there can be multiple combinations of $\tau + c$ disjoint nodes. Among all possible combinations, in this paper, we develop a novel dynamic programming algorithm that selects an optimal combination that minimizes the number of candidates. We remark that any combination of $\tau + c$ disjoint nodes generates candidate strings containing all result strings by Lemma 1. Thus, our optimization technique does not affect the accuracy of similarity search, i.e., it does not miss any result string. In the following subsections, we use $c = 1$ for simplicity, and we will discuss the effect of different c values by treating c as a tunable parameter in Section V-B.

B. OPTIMIZED NODE SELECTION ALGORITHM

Given a query string q with a threshold τ, we search from $\mathcal{H}_{|q|-\tau}$ to $\mathcal{H}_{|q|+\tau}$ to generate candidates as we discussed earlier. Because we independently search each HSTree, and each tree generates candidates independently, in this section, we restrict our discussion to those strings of length l and consider optimized node selection for the HSTree $\mathcal{H}_l$. Given $\tau + 1$ nodes $\{N_1, N_2, \ldots, N_{\tau+1}\}$ of $\mathcal{H}_l$, candidate strings are generated as follows.

$$\mathcal{C} = \bigcup_{i=1}^{\tau+1} \bigcup_{w \in \mathcal{W}(q, N_i)} N_i(w), \tag{4}$$

where q is the query string and w is each segment in $\mathcal{W}(q, N_i)$. Note that N_i also denotes the inverted index of the i^{th} node. Like all other string similarity search techniques that utilize signatures to generate candidates, we assume each partition signature independently generates candidate strings. Therefore, the number of candidates can be estimated as:

$$\mathcal{N}_{\mathcal{C}} = \sum_{i=1}^{\tau+1} \sum_{w \in \mathcal{W}(q, N_i)} |N_i(w)|, \tag{5}$$

where $|N_i(w)|$ denotes the size of the inverted list $N_i(w)$.

An optimal combination of $\tau + 1$ nodes can be obtained by enumerating all possible $\tau + 1$ disjoint nodes, computing the number of candidates generated by each combination, and selecting a combination having the minimum number of candidates. The following lemma states that we can prune certain combinations of $\tau + 1$ disjoint nodes.

Lemma 3: Given a combination of $\tau + 1$ disjoint nodes $\mathcal{S} = \{N_1, \ldots, N_{\tau+1}\}$ of $\mathcal{H}_l$, if $\sum_{k=1}^{\tau+1} \ell(N_k) < l$, there exists another combination that generates candidates no more than the initial combination $\mathcal{S}$, where $\ell(N_k)$ denotes the length of segments indexed in N_k.

Proof: If $\sum_{k=1}^{\tau+1} \ell(N_k) < l$, there should be at least one node $N \in \mathcal{S}$ such that no nodes in the subtree rooted by the sibling of N is included in $\mathcal{S}$. By replacing N with its parent, we can have another combination that generates candidates no more than $\mathcal{S}$, because candidate generated by the parent is obviously a subset of that generated by N. □

Example 8: In Example 7, we may select $\tau + 1$ nodes $\{\mathcal{L}_9^{2,1}, \mathcal{L}_9^{3,3}, \mathcal{L}_9^{3,4}, \mathcal{L}_9^{3,6}, \mathcal{L}_9^{2,4}\}$. In this case, we can replace $\mathcal{L}_9^{3,6}$ with its parent $\mathcal{L}_9^{2,3}$, and candidates generated by $\mathcal{L}_9^{2,3}$ is a subset of candidates generated by $\mathcal{L}_9^{3,6}$.

The following recurrence calculates the minimum number of candidates when we select n disjoint nodes from a subtree rooted by $\mathcal{L}_l^{i,j}$, which is the j^{th} node at the level i of $\mathcal{H}_l$ for those data strings of length l.

if $n = 1$:

$$\mathcal{N}_{\mathcal{C}}(\mathcal{L}_l^{i,j}, n) = \sum_{w \in \mathcal{W}(q, \mathcal{L}_l^{i,j})} |\mathcal{L}_l^{i,j}(w)|, \tag{6}$$

otherwise :

$$\mathcal{N}_{\mathcal{C}}(\mathcal{L}_l^{i,j}, n) = \min_k \{\mathcal{N}_{\mathcal{C}}(\mathcal{L}_l^{i+1,2j-1}, k) + \mathcal{N}_{\mathcal{C}}(\mathcal{L}_l^{i+1,2j}, n-k)\} \tag{7}$$

In case $n = 1$, $\mathcal{L}_l^{i,j}$ generates the minimum number of candidates, thus we select $\mathcal{L}_l^{i,j}$. When $n > 1$, we select k disjoint nodes from the subtree rooted by the left child $\mathcal{L}_l^{i+1,2j-1}$, and remaining $n-k$ disjoint nodes from the subtree rooted by the right child $\mathcal{L}_l^{i+1,2j}$. Among all possible k values, which are discussed in Lemma 4, we select an optimal combination of n disjoint nodes that has the minimum number of candidates. Note that the recurrence considers only those combinations of disjoint nodes $\{N_1, \ldots N_n\}$ such that $\sum_{k=1}^{n} \ell(N_k) = \ell(\mathcal{L}_l^{i,j})$ by the following lemma.

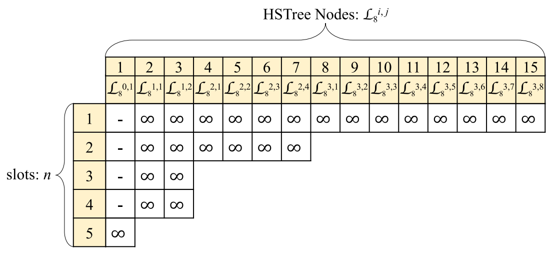
Lemma 4: In the recurrence above, the range of all possible k values is as follows.

$$\max(1, n - 2^{\mathsf{maxL}-(i+1)}) \leq k \leq \min(n-1, 2^{\mathsf{maxL}-(i+1)}),$$

where maxL denotes the maximum level of the tree, i.e., the leaf level $\lfloor \log_2 l \rfloor$.

Proof: It is obvious that $1 \leq k \leq n-1$. Since an HSTree is a full binary tree, the maximum number of disjoint nodes (i.e., the number of nodes in the leaf level) in the subtree rooted by $\mathcal{L}_l^{i+1,2j-1}$ (or $\mathcal{L}_l^{i+1,2j}$) is $2^{\mathsf{maxL}-(i+1)}$. Therefore, $k \leq 2^{\mathsf{maxL}-(i+1)}$ and $n - k \leq 2^{\mathsf{maxL}-(i+1)}$, that is, $n - 2^{\mathsf{maxL}-(i+1)} \leq k \leq 2^{\mathsf{maxL}-(i+1)}$. □

Theorem 1: $\mathcal{N}_{\mathcal{C}}(\mathcal{L}_l^{i,j}, n)$ correctly computes the minimum number of candidates.



HSTree Nodes: $\mathcal{L}_8^{i,j}$

slots: n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	$\mathcal{L}_8^{0,1}$	$\mathcal{L}_8^{1,1}$	$\mathcal{L}_8^{1,2}$	$\mathcal{L}_8^{2,1}$	$\mathcal{L}_8^{2,2}$	$\mathcal{L}_8^{2,3}$	$\mathcal{L}_8^{2,4}$	$\mathcal{L}_8^{3,1}$	$\mathcal{L}_8^{3,2}$	$\mathcal{L}_8^{3,3}$	$\mathcal{L}_8^{3,4}$	$\mathcal{L}_8^{3,5}$	$\mathcal{L}_8^{3,6}$	$\mathcal{L}_8^{3,7}$	$\mathcal{L}_8^{3,8}$
1	-	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
2	-	∞	∞	∞	∞	∞	∞								
3	-	∞	∞												
4	-	∞	∞												
5	∞														

FIGURE 4. **An initial DP array for an HSTree having 15 nodes (i.e., depth = 3) when $\tau = 4$.**

Proof: It considers k disjoint nodes in the left subtree of $\mathcal{L}_l^{i,j}$ and $n-k$ disjoint nodes in the right subtree of $\mathcal{L}_l^{i,j}$. Given any k value, assume that it correctly selects the minimum number of candidates in the left and right subtrees, respectively. Since it considers all possible range of k value by Lemma 4, it computes the minimum number of candidates for the subtree rooted by $\mathcal{L}_l^{i,j}$ by the assumption. When $\mathcal{L}_l^{i,j}$ is a leaf node, it clearly returns the correct number of candidates by Equation (7). Therefore, by induction, $\mathcal{N}_{\mathcal{C}}(\mathcal{L}_l^{i,j}, n)$ correctly computes the minimum number of candidates. □

A minor difficulty with the recurrence is that we can identify relative locations of partitions after selecting optimal partitions, while $\mathsf{LB_T}$ and $\mathsf{UB_T}$ for $\mathcal{W}(q, \mathcal{L}_l^{i,j})$ require the relative location of the partition $\mathcal{L}_l^{i,j}$. Notice that j in $\mathcal{L}_l^{i,j}$ is no more the relative location of the partition $\mathcal{L}_l^{i,j}$, since we select partitions in different levels of the tree. We solve the problem by using the following $\mathsf{LB_L}$ and $\mathsf{UB_L}$ for $\mathcal{W}(q, \mathcal{L}_l^{i,j})$ in our recurrence (see PassJoin [22], [23] for the details of $\mathsf{LB_L}$ and $\mathsf{UB_L}$).

$$\mathsf{LB_L} = \max(1,\ p(\mathcal{L}_l^{i,j}) - \lfloor \frac{\tau - \Delta}{2} \rfloor), \tag{8}$$

$$\mathsf{UB_L} = \min(|q| - \ell(\mathcal{L}_l^{i,j}) + 1, p(\mathcal{L}_l^{i,j}) + \lfloor \frac{\tau + \Delta}{2} \rfloor). \tag{9}$$

After selecting optimal partitions, we use $\mathsf{LB_T}$ and $\mathsf{UB_T}$ to generate candidates with the selected partitions. It is worth noting that we do not need to lookup indices for the inverted lists used in generating candidates, but we can select the required inverted lists from those inverted lists retrieved during partition selection because $[\mathsf{LB_T}, \mathsf{UB_T}] \subseteq [\mathsf{LB_L}, \mathsf{UB_L}]$ [22], [23].

It is obviously inefficient to compute the minimum number of candidates by recursively enumerating all possible combinations of $\tau+1$ disjoint nodes. Instead, we develop a dynamic programming algorithm based on the recurrence above as follows. There are $|\mathcal{H}_l| = 2^{\mathsf{maxL}+1} - 1$ nodes in $\mathcal{H}_l$, where maxL is the leaf level. We make an array A having $|\mathcal{H}_l|$ elements, where the node $\mathcal{L}_l^{i,j}$ corresponds to $A[2^i - 1 + j]$. For the purpose of presentation, we use $A[\mathcal{L}_l^{i,j}]$ to denote $A[2^i - 1 + j]$. The subtree rooted by $\mathcal{L}_l^{i,j}$ has $2^{\mathsf{maxL}-i}$ leaf nodes. Hence, we make $\min(\tau + 1, 2^{\mathsf{maxL}-i})$ slots for $A[\mathcal{L}_l^{i,j}]$ to keep $\mathcal{N}_{\mathcal{C}}(\mathcal{L}_l^{i,j}, n)$ in the n^{th} slot of $A[\mathcal{L}_l^{i,j}]$, i.e., $A[\mathcal{L}_l^{i,j}][n] = \mathcal{N}_{\mathcal{C}}(\mathcal{L}_l^{i,j}, n)$. We initialize each slot of the array to ∞.

For example, Figure 4 shows an initial DP array for an HSTree with depth 3 when $\tau = 4$.

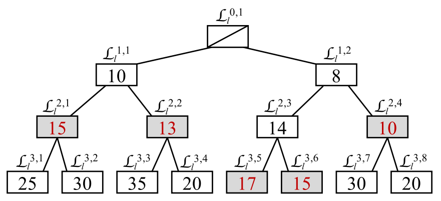
Given an initialized DP array A, Algorithm 1 outlines our dynamic programming algorithm that computes the minimum number of candidates in $\mathcal{H}_l$ for a query string q with a threshold τ. We assume that q, τ and A are globally visible in the algorithm. In the algorithm, each slot of the DP array A has three values: n_cands is the minimum number of candidates, and left and right are links to its children cells, which are used to keep track of the optimal combination of nodes. The algorithm computes the minimum number of candidates only when it is not already computed (Line 1). If the number of nodes to be selected is 1, then it saves the sum of the sizes of the inverted lists for $\mathcal{W}(q, \mathcal{L}_l^{i,j})$ of the current node (Line 4). In this case, the child links are set to nil, which indicates that this node is a terminal node (Line 5). If the number of nodes to be selected is greater than 2, the algorithm selects an optimal combination of nodes in the subtree rooted by current node based on the recurrence (**for** loop in Line 7). It saves the minimum number of candidates in $A[\mathcal{L}_l^{i,j}][n]$.n_cands (Line 10). It also keeps, in $A[\mathcal{L}_l^{i,j}][n]$.left and $A[\mathcal{L}_l^{i,j}][n]$.right, the slot numbers of the children of the current node (Lines 11–12). Note that the algorithm only needs to keep the slot numbers, because the children of current node can be easily located as follows. The left and right children of $A[\mathcal{L}_l^{i,j}]$ are $A[\mathcal{L}_l^{i+1,2j-1}]$ and $A[\mathcal{L}_l^{i+1,2j}]$, respectively.

Algorithm 1: DPSelect($\mathcal{L}_l^{i,j}$, n)

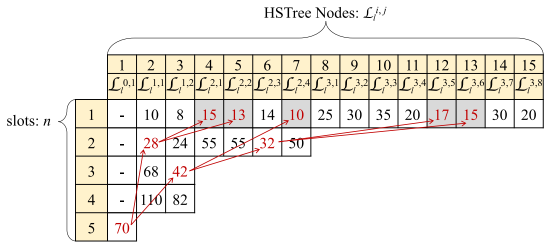
```
input  : $\mathcal{L}_l^{i,j}$ is a node of $\mathcal{H}_l$, and $n$ is the number of
         disjoint nodes to select.
output: $A[\mathcal{L}_l^{i,j}][n]$.n_cands – minimum number of
         candidates of $n$ disjoint nodes in the subtree
         rooted by $\mathcal{L}_l^{i,j}$

1  if $A[\mathcal{L}_l^{i,j}][n]$.n_cands $\neq \infty$ then
2      return $A[\mathcal{L}_l^{i,j}][n]$.n_cands;

3  if $n = 1$ then
4      $A[\mathcal{L}_l^{i,j}][n]$.n_cands $\leftarrow \sum_{w \in \mathcal{W}(q, \mathcal{L}_l^{i,j})} |\mathcal{L}_l^{i,j}(w)|$;
5      $A[\mathcal{L}_l^{i,j}][n]$.left $\leftarrow A[\mathcal{L}_l^{i,j}][n]$.right $\leftarrow$ nil;
6  else
7      for $k \leftarrow \max(1, n - 2^{\text{maxL}-(i+1)})$ to
               $\min(n - 1, 2^{\text{maxL}-(i+1)})$ do
8          $\mathcal{N}_\mathcal{C} \leftarrow$ DPSelect($\mathcal{L}_l^{i+1,2j-1}$, $k$) +
                   DPSelect($\mathcal{L}_l^{i+1,2j}$, $n - k$);
9          if $A[\mathcal{L}_l^{i,j}][n] > \mathcal{N}_\mathcal{C}$ then
10             $A[\mathcal{L}_l^{i,j}][n]$.n_cands $\leftarrow \mathcal{N}_\mathcal{C}$;
11             $A[\mathcal{L}_l^{i,j}][n]$.left $\leftarrow k$;
12             $A[\mathcal{L}_l^{i,j}][n]$.right $\leftarrow n - k$;

13 return $A[\mathcal{L}_l^{i,j}][n]$.n_cands;
```

Recall the location of $\mathcal{L}_l^{i,j}$ in A is $2^i - 1 + j$. Let the location be x. The locations of the left and right children are $2x = 2^{i+1} - 1 + 2j - 1$ and $2x + 1 = 2^{i+1} - 1 + 2j$, respectively. The algorithm finally returns the number of minimum candidates (Line 13).

Example 9: Given a query string q with a threshold $\tau = 4$, consider an HSTree $\mathcal{H}_l$ shown in Figure 5(a). In the figure, the number in each node denotes the sum of the lengths of the inverted lists selected by segments of q (i.e., $\sum_{w \in \mathcal{W}(q, \mathcal{L}_l^{i,j})} |\mathcal{L}_l^{i,j}(w)|$). We can find an optimal combination of disjoint nodes by calling DPSelect($\mathcal{L}_l^{0,1}$, $\tau + 1 = 5$). To find an optimal combination, DPSelect construct a DP array depicted in Figure 5(b). It recursively fills each slot in the array while keeping links to its children slots. Once it fills $A[\mathcal{L}_l^{0,1}][5]$, we can find an optimal combination by following the links of $A[\mathcal{L}_l^{0,1}][5]$, which are depicted in red lines in the figure. The optimal combination of disjoint nodes selected by the algorithm is indicated by the grayed slots.



(a) $\sum_{w \in \mathcal{W}(q, \mathcal{L}_l^{i,j})} |\mathcal{L}_l^{i,j}(w)|$ for each node $\mathcal{L}_l^{i,j}$ in an HSTree $\mathcal{H}_l$


HSTree Nodes: $\mathcal{L}_l^{i,j}$
slots: n

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	$\mathcal{L}_l^{0,1}$	$\mathcal{L}_l^{1,1}$	$\mathcal{L}_l^{1,2}$	$\mathcal{L}_l^{2,1}$	$\mathcal{L}_l^{2,2}$	$\mathcal{L}_l^{2,3}$	$\mathcal{L}_l^{2,4}$	$\mathcal{L}_l^{3,1}$	$\mathcal{L}_l^{3,2}$	$\mathcal{L}_l^{3,3}$	$\mathcal{L}_l^{3,4}$	$\mathcal{L}_l^{3,5}$	$\mathcal{L}_l^{3,6}$	$\mathcal{L}_l^{3,7}$	$\mathcal{L}_l^{3,8}$
1	-	10	8	15	13	14	10	25	30	35	20	17	15	30	20
2	-	28	24	55	55	32	50								
3	-	68	42												
4	-	110	82												
5	70														

(b) DP array for $\mathcal{H}_l$ when $\tau = 4$

FIGURE 5. A running example of the DPSelect **algorithm.**

Lemma 5: The time complexity of the algorithm is $O(l \cdot \tau \cdot C_I)$, where l is the length of strings indexed in an HSTree $\mathcal{H}_l$, τ is a threshold for a query, and C_I is the average cost for index lookups.

Proof: It can be seen that the number of segments in $\mathcal{W}(q, \mathcal{L}_l^{i,j})$ is at most 2τ. The algorithm requires $O(l \cdot \tau \cdot C_I)$ to fill the first row of the DP array, i.e., $A[1][*]$, since there are at most l slots in the first row. In the n^{th} row of the DP array, there are at most $\frac{l}{2^{\lceil log_2 n \rceil}} \leq \frac{l}{n}$ slots. A slot in the n^{th} row requires at most $2(n - 1)$ lookups of the DP array slots (see Line 6 of the algorithm). Hence, the algorithm requires $O(l)$ to fill the n^{th} row ($n > 1$), and it requires $O(l \cdot \tau)$ to fill all the rows except the first row. Consequently, filling the first row dominates the time complexity of the algorithm, which is $O(l \cdot \tau \cdot C_I)$. □

C. REDUCING INDEX LOOKUP OVERHEAD

A drawback of the proposed algorithm is that it looks up indices in all tree nodes. To reduce the overhead of index lookups, we limit the maximum level maxL to $\lceil \log_2(\tau+1) \rceil$, i.e., the minimum level having at least $\tau + 1$ nodes. An interesting observation is that we can ignore HSTree nodes below a certain level to select $\tau + 1$ disjoint nodes for a query. Lemma 6 formally states the observation.

Lemma 6: Given a threshold τ and a maximum level maxL, the minimum level where we can select a node is

$$\mathsf{minL} = \lceil \log_2(2^{\mathsf{maxL}}/(2^{\mathsf{maxL}} - \tau)) \rceil.$$

Proof: The maximum number of possible disjoint nodes (i.e., the number of nodes in the leaf level) in an HSTree is 2^{maxL}. Consider we select a node at minL. Then, we cannot use any nodes in the subtree rooted by the selected node (the subtree has $2^{\mathsf{maxL}-\mathsf{minL}}$ nodes at the level maxL). Hence, the maximum number of remaining disjoint nodes is $2^{\mathsf{maxL}} - 2^{\mathsf{maxL}-\mathsf{minL}}$. To select $\tau + 1$ disjoint nodes, we need to be able to select τ nodes in the remaining disjoint nodes. Hence, we have the inequality, $\tau \leq 2^{\mathsf{maxL}} - 2^{\mathsf{maxL}-\mathsf{minL}}$. The inequality gives us the minimum level $\mathsf{minL} = \lceil \log_2(2^{\mathsf{maxL}}/(2^{\mathsf{maxL}} - \tau)) \rceil$. ☐

Example 10: For the HSTree in Figure 3, if $\tau = 5$, $\mathsf{maxL} = \lceil \log_2(5 + 1) \rceil = 3$ and $\mathsf{minL} = \lceil \log_2(2^3/(2^3 - 5) \rceil = 2$.

The observation can be generalized to $\tau + c$ scheme as follows. Since we need to select $\tau + c$ disjoint nodes, maxL becomes $\lceil \log_2(\tau + c) \rceil$, and minL in Lemma 6 becomes $\lceil \log_2(2^{\mathsf{maxL}}/(2^{\mathsf{maxL}} - (\tau + c - 1))) \rceil$. We remark that Algorithm 1 does not look up the inverted indices of the nodes below minL by Lemma 4 (i.e, the condition in Line 3 is always false for the nodes below minL).

IV. THREADED HSTree

To further reduce the index lookup cost, in this section, we develop a threaded HSTree structure. Consider a segment w indexed in an HSTree node N and let the inverted list for w be I_w. The first half of w is indexed in the left child of N and the second half of w is indexed in the right child of N. Let the inverted lists of the first and second halves be I_{w_1} and I_{w_2}, respectively. We connect I_w with I_{w_1} and I_{w_2} with pointers, which are called *threads*. If we look up the inverted index in N to find I_w for w, we can directly locate I_{w_1} and I_{w_2} by following the threads. Figure 6 shows this modification of the HSTree in Figure 3, where the dashed blue lines denote the threads that connect inverted lists.

We use a threaded HSTree with our algorithm as follows. Given a query, we first look up inverted indices of the nodes at the minimum level minL, and we keep the retrieved inverted lists. For each node N in the next level, we first locate its parent node, and follow the threads of the inverted lists kept in the parent node. It can be easily seen that the inverted lists identified by parent's threads are a subset of the inverted lists required in N. Hence, we look up the inverted index in N to retrieve unidentified inverted lists only. In this way, we can

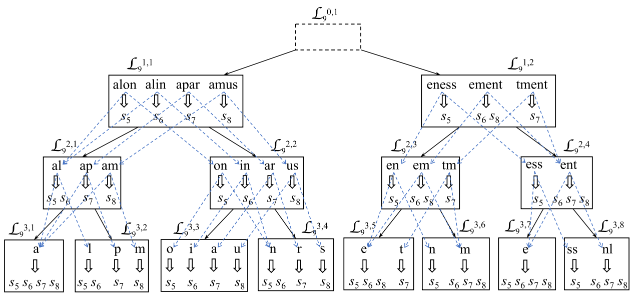


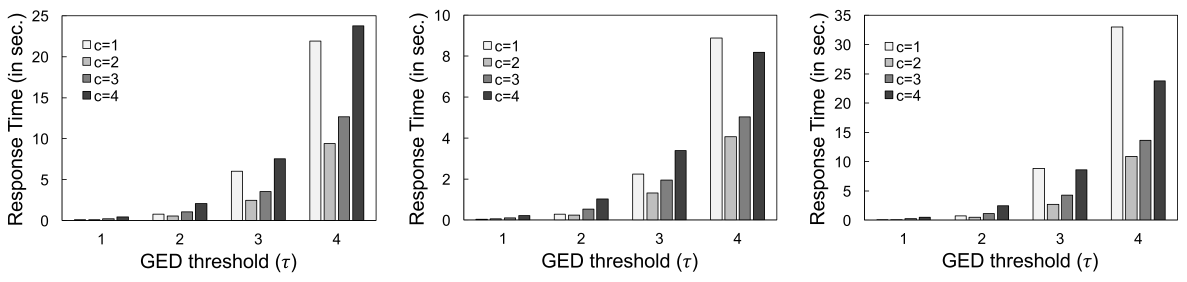
FIGURE 6. Threaded HSTree for the HSTree in Figure 3.

retrieve all inverted lists for the nodes at the levels between minL and maxL. Then, we fill the first row of the DP array using the sizes of retrieved inverted lists. Finally, we apply our algorithm to compute remaining slots of the DP array. We remark that the condition Line 3 of the algorithm is always false, since we initialize the first row of the DP array before calling the algorithm.

Algorithm 2 encapsulates the initialization of the DP array using a threaded HSTree. We assume that q, τ, the HSTree $\mathcal{H}_l$, and the DP array A are visible in the algorithm. The algorithm first initialize an array of sets of inverted lists S_L, which keeps inverted lists corresponding to $\mathcal{W}(q, \mathcal{L}_l^{i,j})$ for each node $L_l^{i,j}$ (Line 1). Then, it retrieve inverted lists for the query in each node at the level minL (Line 2). Recall that an HSTree node N also denotes the inverted index in N. For simplicity, we use $S_L[N]$ to denote the element of S_L corresponding to the node N. After looking up inverted indices in the nodes at the level minL and retrieving inverted lists for the query, it uses the retrieved inverted lists to construct inverted lists of the nodes at the levels above

Algorithm 2: InitializeDPArray

```
1  $S_L \leftarrow$ an array of empty sets;
2  foreach node $N$ at level $\mathsf{minL}$ do
3      $S_L[N] = \bigcup_{w \in \mathcal{W}(q,N)} N(w)$;
4  for $lv \leftarrow \mathsf{minL} + 1$ to $\mathsf{maxL}$ do
5      foreach node $N$ at level $lv$ do
6          $N_P \leftarrow$ parent of $N$;
7          foreach list $l \in S_L[N_P]$ do
8              if $N$ is the left child of $N_P$ then
9                  $S_L[N] = S_L[N] \cup l.\mathsf{left\_thread}$;
10             else $S_L[N] = S_L[N] \cup l.\mathsf{right\_thread}$;
11         foreach $w \in \mathcal{W}(q, N)$ do
12             if $N(w) \notin S_L[N]$ then
13                 $S_L[N] = S_L[N] \cup N(w)$
14 for $lv \leftarrow \mathsf{minL}$ to $\mathsf{maxL}$ do
15     foreach node $N$ at level $lv$ do
16         $A[N][1] \leftarrow \sum_{l \in S_L[N]} |l|$;
```



(a) Actor (b) Movie (c) Corpus

FIGURE 7. **Query time for different c values.**

TABLE 2. **Datasets used in experiments.**

	number of strings	average string length	τ
IMDB Actor	1,213,391	16	1∼4
IMDB Movie	1,568,893	19	1∼4
Web Corpus	10,000,000	21	1∼4

minL (i.e., levels $lv >$ minL) (Line 4). For each node N at the level lv, it first locates the parent node N_P of N (Line 6), and obtains inverted lists for the query in N using the inverted lists of the parent nodes, i.e., $S_L[N_p]$ as follows. If N is the left child of N_p, it follows the left thread of each inverted list in $S_L[N_p]$ (Line 8). Otherwise, it follows the right thread of each inverted list in $S_L[N_p]$ (Line 9). To retrieve those inverted lists that are not identified from $S_L[N_P]$, it finally lookup the inverted index in N (Line 11). For each segment $w \in \mathcal{W}(q, N)$, if the inverted list of w is not identified from $S_L[N_P]$ (Line 12), we lookup the inverted index of N to retrieve the inverted index (Line 13). In Line 12, we need to check if w is contained in the segments corresponding to the inverted lists in $S_L[N]$. This membership test can be easily done by merging the positions of the segments in $\mathcal{W}(q, N)$ and the positions of the segments corresponding to the inverted lists in $S_L[N]$.

V. EXPERIMENTS

A. EXPERIMENTAL SETTINGS

In experiments, we used four widely used real-world datasets, IMDB Actor and Movie (http://www.imdb.com), and Web Corpus (http://www.ldc.upenn.edu/Catalog, number LDC2006T13). Some important statistics of the datasets are presented in Table 2. We chose the datasets to compare performance for short and long strings. Our algorithm was implemented in GNU C++ and compiled with -O3 option. All experiments were conducted on a machine with 32GB main memory and Intel i7 CPU running an Ubuntu operating system. Datasets and indices were held in main memory.

We randomly extracted 2000 query strings from each data set, ran queries. We evaluated the proposed technique in terms of query processing time. The reported results in this section are aggregated response times from 2000 queries. Since the HSTree technique consistently outperformed other existing techniques as reported in [39], [48], we compared our technique with the original HSTree technique [39], [48]. In the remaining sections, we denote our algorithm by OptSearch and the original HSTree search algorithm by HSSearch.

B. EXPERIMENTS ON DIFFERENT C VALUES

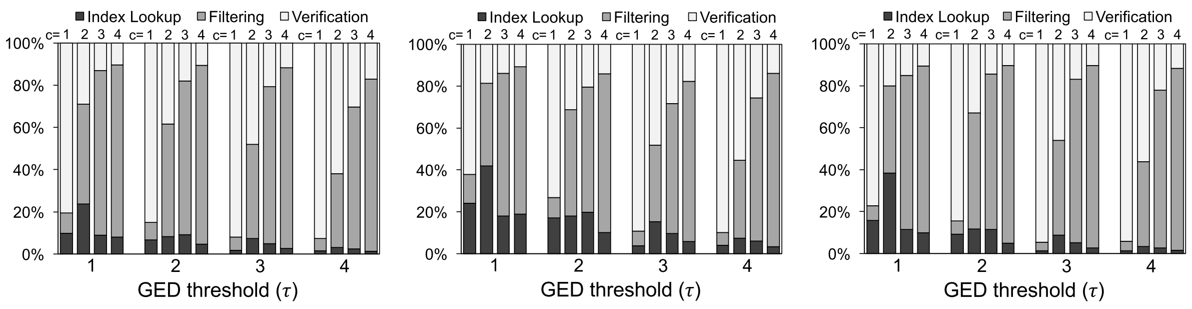
In this subsection, we evaluate our algorithm varying c values. Even when $c > 1$, as shown in [17][5], we can still estimate an upper bound of the number of candidates with the sum of the sizes of inverted lists for a query string. Therefore, we can still use our algorithm to obtain an optimal combination of $\tau + c$ disjoint nodes. In this case, as discussed in Section III-C, we only need to change maxL to $\lceil \log_2(\tau + c) \rceil$ and minL to $\lceil \log_2(2^{\mathsf{maxL}}/(2^{\mathsf{maxL}} - (\tau + c - 1))) \rceil$ in our algorithm.

Figure 7 shows query response times for alternative c values on three different datasets. If we use a c value larger than 1, an underflow of the number of partitions may occurs. In this case, we re-evaluated the query using $\tau + 1$ partitions. For all datasets and thresholds, we observed that $c = 2$ exhibited the best performance. It can be explained by the query time ratios shown in Figure 8. There is a trade-off between filtering (i.e., merging inverted lists) and verification (i.e., edit distance computation) times for different c values. To obtain the best performance, we need to balance the trade-off. As shown in the figure, the time differences between filtering and verification were minimized when c was 2. We remark that the cost for selecting an optimal combination is negligible, and it is included in Indexlookup in Figure 8. These results justify the motivation of our work described in Section III-A. Based on the results in this subsection, we used $c = 2$ in our algorithm in the following subsections.

C. EXPERIMENTS ON INDEX LOOKUP TIME

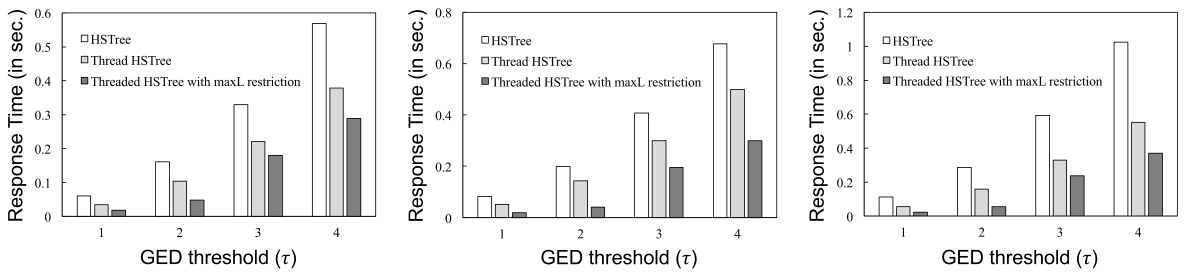
In this subsection, we evaluate the effects on index lookup times when a threaded HSTree and the restriction of the maximum level maxL are applied. Figure 9 shows the results. When we used a threaded HSTree, index lookup time was reduced by 1.5 times on average. When we applied the restriction of maxL (i.e., maxL= $\lceil \log_2(\tau + c) \rceil$) along with a threaded HSTree, index lookup time was reduced by 2 times on average. As shown in Figure 8, index lookup time did not affect the query time when a threshold was large (e.g., $\tau \geq 3$).

[5]we remark that [17] is a DNA read mapping technique that utilizes q-grams, and thus the contributions of this paper are orthogonal to that of [17].



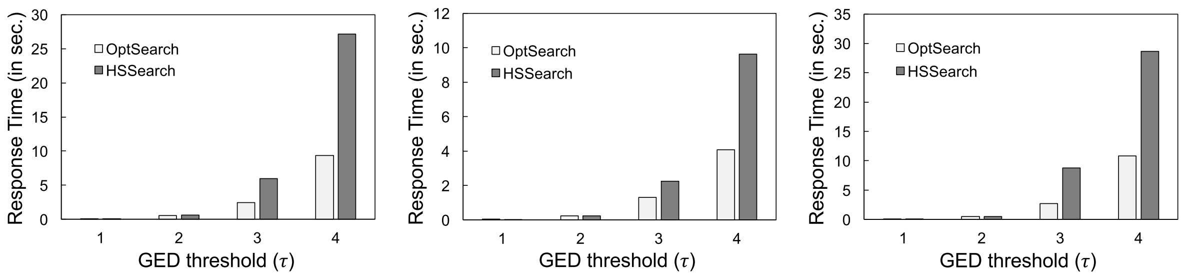
(a) Actor (b) Movie (c) Corpus

FIGURE 8. **Query time ratios for different *c* values.**



(a) Actor (b) Movie (c) Corpus

FIGURE 9. **Index lookup time comparisions.**



(a) Actor (b) Movie (c) Corpus

FIGURE 10. **Comparisons with HSSearch (Query Time).**

For a low threshold (e.g. $\tau = 1$), however, index lookup time was very important on the overall performance because merging inverted lists and verifying candidate strings were very fast. Since the proposed technique looks up inverted indices in all HSTree nodes to select an optimized combination of nodes, it is crucial to reduce index lookup time for low thresholds. As shown in the experiments in this subsection, the threaded HSTree structure and maxL restriction technique effectively reduced index lookup time.

D. COMPARISONS WITH HSSearch

In this subsection, we compare our algorithm with HSSearch. Figure 10 shows the results. On each dataset, OptSearch outperformed HSSearch by up to about 3 times. For low thresholds (e.g., $\tau \leq 2$), the performance of OptSearch was just as good as HSSearch. This is because inverted lists for partitions are very short and list merging and edit distance computation can be done very quickly on a low threshold. Since OptSearch requires more index lookups, the benefit of OptSearch is reduced by the overhead of index lookups. As a threshold increased, however, OptSearch significantly outperformed HSSearch because of the optimal partition selection and a good balance between filtering and verification. For example, OptSearch is about 3 times faster than HSSearch when $\tau = 4$ on the Actor dataset (Figure 10(a)).

Since we used the dataset, Corpus, which contains 10 millions of strings, we can see that the proposed techniques scales well to a large dataset from Figure 10(c).

VI. RELATED WORK

A. SIMILARITY MEASURES

The problem of quantifying similarity between objects has witnessed growing interest over the past decades. To measure the similarity between text data, character-based similarity functions and token set-based similarity functions are widely used. The most representative character-based similarity function is edit distance [11], [27], [28], [38], which is also known as Levenshtein distance. Since it reflects the ordering of characters in strings and it allows non-trivial alignment, it is widely adopted in many applications. For concrete examples, it is used in practical applications such as `diff` and `patch` commands in Linux OS systems, source code management for version control systems like GitHub, and DNA read mappers (e.g. [17], [18]) for analyzing genomic data. Set-based similarity functions such as Jaccard coefficient [12], Dice [9] and Cosine similarity are also used to measure the similarity between text data by tokenizing each string into a set of words or q-grams. Since set-based similarity functions considers only exact match between tokens, they might not correctly measure similarity when the granularity of a token is large. Fast-Join [40] and MF-Join [42] address this problem by considering similarity between tokens using edit distance before computing set-based similarity. LSA [19] and it variants (e.g., [24], [25]) also have been developed to measure similarity between documents through corpus analysis.

There are similarity functions that are used in structured data. SimRank [13] and many variants [26], [37], [47], [49], [50], [52] has been proposed to consider semantic similarity information between objects in information networks. The intuition behind SimRank is that similar objects are linked by similar objects. Based on the intuition, it quantifies node similarity based on the compound similarity of their neighbors. Graph edit distance [32], [34] and feature-based similarity functions [5], [36], [51], [53] has been proposed to quantify the similarity between complex objects represented by graph models. Similar to edit distance, graph edit distance measure the distance between two graphs using the minimum number of edit operations to make the graphs isomorphic. In contrast to edit distance, however, graph edit distance computation is NP-hard and many techniques have been proposed to efficiently compute graph edit distance (e.g. [15], [32]).

B. STRING SIMILARITY QUERY PROCESSING

Set similarity join is the problem of finding similar pairs of records from two collections of records, which is essential in many applications including data cleaning [8] and near duplicate detection [45]. Many algorithms are developed for the problem of set similarity join [1], [2], [6], [8], [10], [30], [35], [40], [41], [43]–[45]. Some of these algorithms (e.g. [2], [31]) solve only join problems, but most of these algorithms can be applied to the search problem in their original form or with slight modification. Many algorithms and inverted index structures have been proposed for the similarity search problem [3], [4], [6], [7], [16], [20], [21], [29], [41], [46]. The technique called VGRAM [21], [46] was proposed to use variable-length grams in an inverted index to improve similarity search performance and reduce the index size. A disk-based inverted index structure [4] was proposed for supporting similarity search on large datasets. In [16], a dataset partitioning algorithm has been proposed to reduce the number of candidates by exploiting document frequency orderings. Recent techniques exploits partitioning of data strings to establish a filtering condition based on the pigeonhole principle. PassJoin [22], [23] decomposes data strings into $\tau+1$ partitions to perform efficient string similarity join. HSTree [39], [48] proposes a hierarchical index structure that considers multiple partitionings of a string to support string similarity search using the partition-based approach of PassJoin.

VII. CONCLUSION

In this paper, we propose an optimal partition selection algorithm to improve the performance of string similarity search using the HSTree indexing technique. We observed that there can be multiple combination of $\tau + c$ disjoint nodes in an HSTree, and proposed a novel dynamic programming algorithm that selects an optimal combination of HSTree nodes. To reduce the overhead of selecting optimal combination, we also proposed a threaded HSTree, which is an enhanced HSTree structure. We evaluated the proposed technique using real datasets and showed our technique outperformed the existing technique HSSearch.

REFERENCES

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

[2] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *Proc. 16th Int. Conf. World Wide Web WWW*, 2007, pp. 131–140.

[3] A. Behm, S. Ji, C. Li, and J. Lu, "Space-constrained gram-based indexing for efficient approximate string search," in *Proc. IEEE 25th Int. Conf. Data Eng.*, Mar. 2009, pp. 604–615.

[4] A. Behm, C. Li, and M. J. Carey, "Answering approximate string queries on large data sets using external memory," in *Proc. IEEE 27th Int. Conf. Data Eng.*, Apr. 2011, pp. 888–899.

[5] H. Bunke and K. Shearer, "A graph distance metric based on the maximal common subgraph," *Pattern Recognit. Lett.*, vol. 19, nos. 3–4, pp. 255–259, Mar. 1998.

[6] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani, "Robust and efficient fuzzy match for online data cleaning," in *Proc. ACM SIGMOD Int. Conf. Manage. Data SIGMOD*, 2003, pp. 313–324.

[7] S. Chaudhuri, V. Ganti, and L. Gravano, "Selectivity estimation for string predicates: Overcoming the underestimation problem," in *Proc. 20th Int. Conf. Data Eng.*, 2004, pp. 227–238.

[8] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *Proc. 22nd Int. Conf. Data Eng. (ICDE)*, Apr. 2006, p. 5.

[9] L. R. Dice, "Measures of the amount of ecologic association between species," *Ecology*, vol. 26, no. 3, pp. 297–302, Jul. 1945.

[10] 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 *Proc. VLDB*, Sep. 2001, pp. 491–500.

[11] P. A. V. Hall and G. R. Dowling, "Approximate string matching," *ACM Comput. Surv.*, vol. 12, pp. 381–402, Dec. 1980.

[12] P. Jaccard, "Étude comparative de la distribution florale dans une portion des alpes et des jura," *Bull Soc Vaudoise Sci. Nat.*, vol. 37, pp. 547–579, 1901.

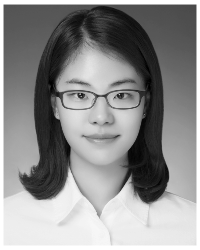
[13] G. Jeh and J. Widom, "SimRank: A measure of structural-context similarity," in *Proc. 8th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining KDD*, 2002, pp. 538–543.

[14] J. Kim, "An effective candidate generation method for improving performance of edit similarity query processing," *Inf. Syst.*, vol. 47, pp. 116–128, Jan. 2015.

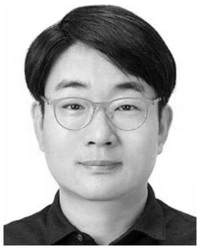
[15] J. Kim, D.-H. Choi, and C. Li, "Inves: Incremental partitioning-based verification for graph similarity search," in *Proc. EDBT*, 2019, pp. 229–240.

[16] J. Kim and H. Lee, "Efficient exact similarity searches using multiple token orderings," in *Proc. IEEE 28th Int. Conf. Data Eng.*, Apr. 2012, pp. 822–833.

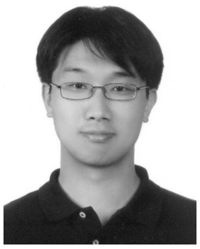
[17] J. Kim, C. Li, and X. Xie, "Improving read mapping using additional prefix grams," *BMC Bioinf.*, vol. 15, no. 1, p. 42, 2014.
[18] J. Kim, C. Li, and X. Xie, "Hobbes3: Dynamic generation of variable-length signatures for efficient approximate subsequence mappings," in *Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE)*, May 2016, pp. 169–180.
[19] T. K. Landauer and S. T. Dumais, "A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge.," *Psychol. Rev.*, vol. 104, no. 2, pp. 211–240, 1997.
[20] C. Li, J. Lu, and Y. Lu, "Efficient merging and filtering algorithms for approximate string searches," in *Proc. IEEE 24th Int. Conf. Data Eng.*, Apr. 2008, pp. 257–266.
[21] C. Li, B. Wang, and X. Yang, "VGRAM: Improving performance of approximate queries on string collections using variable-length grams," in *Proc. VLDB*, 2007, pp. 303–314.
[22] G. Li, D. Deng, and J. Feng, "Pass-join+: A partition-based method for string similarity joins with edit-distance constraints," *ACM Trans. Database Syst.*, vol. 38, no. 2, pp. 1–40, 2013.
[23] G. Li, D. Deng, J. Wang, and J. Feng, "Pass-join: A partition based method for similarity joins," *Proc. VLDB Endowment*, vol. 5, no. 3, pp. 253–264, 2011.
[24] P. Martin, S. Benno, and A. Maik, "A wikipedia- based multilingual retrieval model," in *Proc. ECIR*, 2008, pp. 522–530.
[25] I. Matveeva, G. Levow, A. Farahat, and C. Royer, "Term representation with generalized latent semantic analysis," in *Proc. RANLP*, 2005, pp. 308–315.
[26] T. Milo, A. Somech, and B. Youngmann, "Boosting simrank with semantics," in *Proc. EDBT*, 2019, pp. 1–12.
[27] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *J. Mol. Biol.*, vol. 48, no. 3, pp. 443–453, Mar. 1970.
[28] J. L. Peterson, "Computer programs for detecting and correcting spelling errors," *Commun. ACM*, vol. 23, no. 12, pp. 676–687, Dec. 1980.
[29] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin, "Efficient exact edit similarity query processing with the asymmetric signature scheme," in *Proc. Int. Conf. Manage. Data SIGMOD*, 2011, pp. 1033–1044.
[30] K. Ramasamy, J. M. Patel, R. Kaushik, and J. F. Naughton, "Set containment joins: The good, the bad and the ugly," in *Proc. VLDB*, Sep. 2000, pp. 351–362.
[31] L. A. Ribeiro and T. Härder, "Efficient set similarity joins using min-prefix," in *Proc. ADBIS*, 2009, pp. 88–102.
[32] K. Riesen, S. Fankhauser, and H. Bunke, "Speeding up graph edit distance computation with a bipartite heuristic," in *Proc. MLG*, 2007, pp. 21–24.
[33] M. Sahami and T. D. Heilman, "A Web-based kernel function for measuring the similarity of short text snippets," in *Proc. 15th Int. Conf. World Wide Web WWW*, 2006, pp. 377–386.
[34] A. Sanfeliu and K.-S. Fu, "A distance measure between attributed relational graphs for pattern recognition," *IEEE Trans. Syst., Man, Cybern.*, vols. SMC–13, no. 3, pp. 353–362, May 1983.
[35] S. Sarawagi and A. Kirpal, "Efficient set joins on similarity predicates," in *Proc. ACM SIGMOD Int. Conf. Manage. Data SIGMOD*, 2004, pp. 743–754.
[36] H. Shang, X. Lin, Y. Zhang, J. X. Yu, and W. Wang, "Connected substructure similarity search," in *Proc. Int. Conf. Manage. Data SIGMOD*, 2010, pp. 903–914.
[37] H. Tong, C. Faloutsos, and J.-Y. Pan, "Fast random walk with restart and its applications," in *Proc. 6th Int. Conf. Data Mining (ICDM)*, Dec. 2006, pp. 613–622.
[38] R. A. Wagner and M. J. Fischer, "The String-to-String correction problem," *J. ACM (JACM)*, vol. 21, no. 1, pp. 168–173, Jan. 1974.
[39] J. Wang, G. Li, D. Deng, Y. Zhang, and J. Feng, "Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search," in *Proc. IEEE 31st Int. Conf. Data Eng.*, Apr. 2015, pp. 519–530.
[40] J. Wang, G. Li, and J. Fe, "Fast-join: An efficient method for fuzzy token matching based string similarity join," in *Proc. IEEE 27th Int. Conf. Data Eng.*, Apr. 2011, pp. 458–469.
[41] J. Wang, G. Li, and J. Feng, "Can we beat the prefix filtering?: An adaptive framework for similarity join and search," in *Proc. Int. Conf. Manage. Data SIGMOD*, 2012, pp. 85–96.
[42] J. Wang, C. Lin, and C. Zaniolo, "MF-join: Efficient fuzzy string similarity join with multi-level filtering," in *Proc. IEEE 35th Int. Conf. Data Eng. (ICDE)*, Apr. 2019, pp. 386–397.
[43] C. Xiao, W. Wang, and X. Lin, "Ed-join: An efficient algorithm for similarity joins with edit distance constraints," in *Proc. PVLDB*, Aug. 2008, pp. 933–944.
[44] C. Xiao, W. Wang, X. Lin, and H. Shang, "Top-k set similarity joins," in *Proc. IEEE 25th Int. Conf. Data Eng.*, Mar. 2009, pp. 916–927.
[45] C. Xiao, W. Wang, X. Lin, and J. X. Yu, "Efficient similarity joins for near duplicate detection," in *Proc. 17th Int. Conf. World Wide Web WWW*, 2008, pp. 131–140.
[46] X. Yang, B. Wang, and C. Li, "Cost-based variable-length-Gram selection for string collections to support approximate queries efficiently," in *Proc. ACM SIGMOD Int. Conf. Manage. Data SIGMOD*, 2008, pp. 353–364.
[47] S.-H. Yoon, S.-W. Kim, and S. Park, "C-rank: A link-based similarity measure for scientific literature databases," *Inf. Sci.*, vol. 326, pp. 25–40, Jan. 2016.
[48] M. Yu, J. Wang, G. Li, Y. Zhang, D. Deng, and J. Feng, "A unified framework for string similarity search with edit-distance constraint," *VLDB J.*, vol. 26, no. 2, pp. 249–274, Apr. 2017.
[49] W. Yu, X. Lin, W. Zhang, J. Pei, and J. A. McCann, "SimRank*: Effective and scalable pairwise similarity search based on graph topology," *VLDB J.*, vol. 28, no. 3, pp. 401–426, Jun. 2019.
[50] M. Zhang, J. Wang, and W. Wang, "HeteRank: A general similarity measure in heterogeneous information networks by integrating multi-type relationships," *Inf. Sci.*, vol. 453, pp. 389–407, Jul. 2018.
[51] S. Zhang, J. Yang, and W. Jin, "SAPPER: Subgraph indexing and approximate matching in large graphs," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 1185–1194, Sep. 2010.
[52] P. Zhao, J. Han, and Y. Sun, "P-rank: A comprehensive structural similarity measure over information networks," in *Proc. 18th ACM Conf. Inf. Knowl. Manage. CIKM*, 2009, pp. 553–562.
[53] G. Zhu, X. Lin, K. Zhu, W. Zhang, and J. X. Yu, "TreeSpan: Efficiently computing similarity all-matching," in *Proc. Int. Conf. Manage. Data SIGMOD*, 2012, pp. 529–540.



TAEGYOUNG LEE received the B.S. and M.S. degrees in computer science and engineering from Jeonbuk National University, South Korea, in 2015 and 2017, respectively, and the second M.S. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 2019. She is currently pursuing the Ph.D. degree in computer science and engineering with The Hong Kong University of Science and Technology (HKUST). Her research interests include data analysis, theory of computation, and computational geometry.



TAE-SUN CHUNG received the B.S. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in February 1995, and the M.S. and Ph.D. degrees in computer science from Seoul National University, South Korea, in February 1997 and August 2002, respectively. He is currently a Professor with the Department of Software, Ajou University. His current research interests include flash memory storage, query processing in spatial databases, machine learning, similarity search, and general database systems.



JONGIK KIM received the B.S. and M.S. degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 1998 and 2000, respectively, and the Ph.D. degree in computer engineering from Seoul National University, South Korea, in 2004. He worked as a Senior Researcher at the Electronics and Telecommunications Research Institute (ETRI), from 2004 to 2007. He joined the Division of Computer Science and Engineering, Jeonbuk National University, as a Faculty Member, in 2007. He is currently a Professor with Jeonbuk National University. He was a Visiting Scholar with the University of California at Irvine (UCI) Irvine, from 2012 to 2013. He has been working in the area of semi-structured database (XML database), telematics systems, flash-memory data management, event stream processing, and similarity query processing.