A Scalable Index for Top-k Subtree Similarity Queries

Daniel Kocher
University of Salzburg
Salzburg, Austria
dkocher@cs.sbg.ac.at

Nikolaus Augsten
University of Salzburg
Salzburg, Austria
Nikolaus.Augsten@sbg.ac.at

ABSTRACT

Given a query tree Q, the *top-k subtree similarity query* retrieves the k subtrees in a large document tree T that are closest to Q in terms of tree edit distance. The classical solution scans the entire document, which is slow. The state-of-the-art approach precomputes an index to reduce the query time. However, the index is large (quadratic in the document size), building the index is expensive, updates are not supported, and data-specific tuning is required.

We present a scalable solution for the top-k subtree similarity problem that does not assume specific data types, nor does it require any tuning. The key idea is to process promising subtrees first. A subtree is promising if it shares many labels with the query. We develop a new technique based on inverted lists that efficiently retrieves subtrees in the required order and supports incremental updates of the document. To achieve linear space, we avoid full list materialization but build relevant parts of a list on the fly.

In an extensive empirical evaluation on synthetic and real-world data, our technique consistently outperforms the state-of-the-art index w.r.t. memory usage, indexing time, and the number of candidates that must be verified. In terms of query time, we clearly outperform the state of the art and achieve runtime improvements of up to four orders of magnitude.

ACM Reference Format:
Daniel Kocher and Nikolaus Augsten. 2019. A Scalable Index for Top-k Subtree Similarity Queries. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319892

1 INTRODUCTION

Data with hierarchical structure are naturally represented as trees. A tree stores data values in node labels and encodes the relation between the values in the structure (e.g., text values and element nesting in XML). We consider applications that, given an example tree (the query), are interested in subtrees of a large document tree that are similar to the query. An example is the abstract syntax tree of a large software project [1, 14]: In order to avoid code duplication or detect code moves, software engineers are interested in finding all code fragments (i.e., subtrees of the abstract syntax tree) that are similar to a given example fragment. In RNA secondary structures (which are represented as ordered, labeled trees [3, 17]), biologists search for similar foldings of RNA subsequences. To automatically extract product information from the web, the similarity of substructures in web pages are leveraged [28]. Production engineers retrieve components with similar building plans from bills of materials, which form trees that may consists of millions of nodes [15, 19].

We study *top-k subtree similarity queries*: given a large document tree T and a (small) query tree Q, find the k most similar subtrees in T w.r.t. Q. Two trees are similar if their *edit distance* [31], a common tree similarity measure, is small. The edit distance between two ordered labeled trees is defined as the minimum number of node edit operations (insertion, deletion, rename) that transform one tree into the other.

Previous solutions for top-k subtree similarity queries fall into two categories: index-based and index-free algorithms. TASM-Postorder [4] is the fastest index-free algorithm and runs in small memory. Unfortunately, TASM-Postorder must scan the entire document to answer a top-k query, which is slow. StructureSearch [9] addresses this issue and leverages a precomputed index to retrieve candidate subtrees. The candidates must be verified using the edit distance.

StructureSearch runs faster than TASM-Postorder but suffers from the following issues: (1) The index size is quadratic in the document size n for deep trees; note that the document is the database over which we answer the top-k query. (2) Despite the index, StructureSearch must retrieve and verify many subtrees, which leads to high runtimes also for small values of k. (3) While StructureSearch can be generalized to generic tree data, the solution is tailored to XML documents, which have many repeating labels in the inner nodes (element tags) and infrequent labels in the leaves (text values). Further, XML trees are typically flat. Flat trees are in favor of StructureSearch since the index grows larger for deep trees. (4) The index is not updatable.

Our solution is based on the idea of a *candidate score*. The candidate score ranks all subtrees of a document. The score is high if the query and the subtree share many labels. Intuitively, subtrees with a high score are more likely to be close to the query in terms of edit distance. By processing subtrees in candidate score order, we (a) find good candidates quickly and (b) can stop early when the ranking is good enough, i.e., all remaining subtrees cannot improve the ranking. Stopping early is possible since the candidate score implies a lower bound on the edit distance. The candidate score is very effective. In many settings, we verify orders of magnitude fewer candidate subtrees than StructureSearch; in some settings we only verify k candidates, which is optimal.

The challenge is to efficiently generate candidates in score order. The query is not known upfront, thus the order must be established at query time and cannot be precomputed. It is clearly not feasible to enumerate all subtrees and sort them by their score. We introduce a new technique that is based on an inverted list index over the document node labels. The inverted list of a label stores all subtrees that contain that label. We split the lists into partitions of subtrees with the same size and show how to leverage the list partitions to processes the subtrees in score order. The partitions are accessed in the order of the best candidate score that may be found in that partition. Only relevant partitions need to be accessed, e.g., there is only a single partition that may contain subtrees of the highest score.

The catch is that the label inverted list index is quadratic in the document size n for tress with depth $O(n)$; for such trees, also the index of the state-of-the-art algorithm, StructureSearch, is quadratic. We propose a new algorithm, SlimCone, which uses an incrementally updatable, linear-space index structure to build the relevant partitions of the inverted lists on the fly at query time. SlimCone verifies the subtrees in non-decreasing candidate score order. We show how to generate partitions efficiently such that the performance penalty of generating the partitions on the fly is small.

Summarizing, our contributions are the following.

- We propose SlimCone, a new, index-based algorithm for the top-k subtree similarity problem. SlimCone verifies subtrees in decreasing candidate score order, i.e., more promising subtrees are processed first. SlimCone does not require any parameters and is not tailored to a specific data type.
- The state-of-the-art algorithm uses a quadratic-size index. We propose the first linear-space index for top-k subtree similarity queries. Our index groups subtrees into partitions. All subtrees in a partition have the same guarantee w.r.t. to the candidate score such that we find promising subtrees efficiently.
- We propose an extension of SlimCone that supports incremental index updates. Previous work must recompute the index from scratch when the document tree is updated.
- We empirically evaluate our solution on large synthetic and real-world data sets. Our technique clearly outperforms the state of the art w.r.t. memory usage, indexing time, number of verified candidates, and query runtime, often by orders of magnitude.

The remaining paper is organized as follows. Section 2 provides background material and introduces the problem statement. Section 3 discusses the candidate scores. In Sections 4-6 we present our index structures and algorithms. Section 7 describes how to make our index incrementally updatable. We discuss related work in Section 8. Before we conclude in Section 10, we provide empirical evidence of the scalability and efficiency of our solution in Section 9. Appendix A provides proofs to all lemmata and theorems, and the pseudo codes for all algorithms described in Sections 4-6.

2 NOTATION, BACKGROUND, AND PROBLEM STATEMENT

Trees. We assume *rooted, ordered, labeled trees.* A *tree* T is a directed, acyclic, connected graph with nodes $V(T)$ and directed edges $E(T) \subseteq V(T) \times V(T)$. Each node has at most one incoming edge, the node with no incoming edge is the *root node.* The size of a tree, $|T| = |V(T)|$, is the number of its nodes. In an edge $(u, v) \in E(T)$, u is the *parent* of v, denoted $par(v)$, and v is the *child* of u. Two nodes are *siblings* if they have the same parent. A *leaf node* has no children. Each node u has a *label*, $\lambda(u)$, which is not necessarily unique. The multiset of all labels in T is $\mathcal{L}(T)$. The *postorder* (*preorder*) *identifier* of node u, $post(u)$ ($pre(u)$), is the postorder (preorder) position of u in the tree (1-based numbering). The trees are *ordered*, i.e., the sibling order matters. If node u is on the path from the root to node v, $u \neq v$, then v is a *descendant* of u, and u is a *ancestor* of v. A *subtree* T_u of T is a tree that consists of node u, all descendants of u, and all edges in $E(T)$ connecting these nodes.

Tree Edit Distance. The edit distance, $\delta(S, T)$, between two trees, S, T, is the minimum number of node edit operations that transforms S into T. We assume the standard node operations [31]: *Rename* changes the label of a node. *Delete* removes a node u and connects the children of the deleted node to its parent, starting at the sibling position of u and maintaining the sibling order. *Insert* adds a new node u as the i-th child of an existing node p, replacing a (possibly empty) sequence $C = (c_i, c_{i+1}, \ldots, c_j)$ of p's children; the child sequence C is connected under the new node u. Insert and delete are reverse operations. The fastest algorithms for the tree edit distance run in $O(|T|^3)$ time and $O(|T|^2)$

space [22], i.e., computing the edit distance is expensive and should be avoided.

Lower Bounds. A lower bound for the tree edit distance may underestimate the true distance, but never overestimates it. A number of edit distance lower bounds have been defined [21]. Lower bounds are typically computed much faster than the edit distance. We leverage the *label lower bound*,

$$llb(S,T) = max\{|S|, |T|\} - |\mathcal{L}(S) \Cap \mathcal{L}(T)| \leq \delta(S,T)^{1}, \quad (1)$$

and the *size lower bound*,

$$slb(S,T) = ||S| - |T|| \leq \delta(S,T) \quad (2)$$

Definition 2.1 (Top-k Subtree Similarity Query). Given a query tree Q, a document tree T, $k \leq |T|$. The *top-k subtree similarity query* returns a top-k ranking R, where R is the sequence of the k most similar subtrees of document T w.r.t. query Q such that $\forall T_j \notin R, T_i \in R. \delta(Q,T_i) \leq \delta(Q,T_j)$. The subtrees in $R = [T^1, T^2, \ldots, T^k]$ are sorted by their edit distance to Q, i.e., $\forall 1 \leq i < j \leq k. \delta(Q,T^i) \leq \delta(Q,T^j)$.

Problem Statement. Our goal is a time- and space-efficient solution for the top-k subtree similarity query that scales to large document trees.

A naive solution computes the edit distance $\delta(Q,T_i)$ for all subtrees $T_i \in T$, sorts them by $\delta(Q,T_i)$, and returns the first k subtrees in ascending sort order. Obviously, this approach does not scale to large documents [4]. Efficient techniques prune irrelevant subtrees and compute the edit distance only for candidate subtrees that cannot be filtered. Well known filter techniques include the following.

Size Filter. Augsten et al. [4] show that only subtrees of a maximum size $\tau = 2|Q| + k$ need to be considered, thus subtrees T_i, $|T_i| > \tau$, can be pruned.

Ranking Filter. Once an intermediate ranking R' of size k is obtained, the edit distance $\delta(Q,R'[k])$ ($\delta(R'[k])$ for short) between the query Q and the last tree $R'[k]$ in the ranking serves as a filter: A subtree $T_i \notin R'$ improves the final ranking R' iff $\delta(Q,T_i) < \delta(R'[k])$ [4]. Together with a lower bound, $lb(Q,T_i)$, a subtree can be safely pruned if $lb(Q,T_i) \geq \delta(R'[k])$.

The better the ranking, the more effective is the ranking filter. Thus, to reduce the number of verifications it is important to find good subtrees early in the process.

Table 1 provides an overview of our notation.

3 EFFECTIVE CANDIDATE GENERATION

The key idea of our approach is to prioritize promising subtrees. If we fill the ranking with good subtrees, the ranking filter (cf. Section 2) is effective and we can terminate early.

^{1}$A \Cap B$ denotes the intersection between two multisets, A and B.

Table 1: Notation overview.

Notation	Description
T/Q	document / query tree
R/R'	final / intermediate top-k ranking
k	results size, $k = \|R\|$
$R[j]$	j-th entry in R
T_i	a subtree $T_i \in T$
$par(u)$	parent of node u
$pre(u)/post(u)$	preorder / postorder identifier of node u
$\lambda(u)$	label of node u
$\mathcal{L}(T_i)$	label multiset of tree T_i
$\delta(Q,T_i)$	edit distance btw. Q and T_i
$\delta(R[j])$	edit distance btw. Q and j-th entry in R
$slb(Q,T_i)$	size lower bound btw. Q and T_i
$llb(Q,T_i)$	label lower bound btw. Q and T_i
τ $(= 2\|Q\| + k)$	maximum relevant subtree size [4]

In this section we define the *candidate score* to rank subtrees. In the following sections we discuss how to retrieve subtrees in the order of their candidate score.

Definition 3.1 (Candidate Score). Given query Q and document T, the candidate score of a subtree T_i of T is

$$score(T_i) = \frac{1}{1 + llb(Q,T_i)},$$

where $llb(Q,T_i)$ is the label lower bound between Q and T_i.

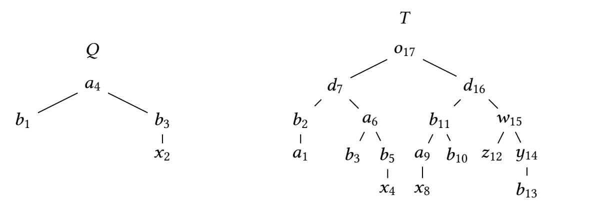
The candidate score is in the interval $(0, 1]$, more promising subtrees score higher. The candidate score imposes a total order on the subtrees of document T, which we call *candidate score order*: Given two subtrees $T_i, T_j \in T$, $T_i > T_j$ iff $score(T_i) > score(T_j)$.

A subtree T_i is processed by computing the tree edit distance between T_i and the query Q, and by inserting T_i into the ranking if $\delta(Q,T_i) < \delta(R[k])$. If we process the subtrees in candidate score order, we can stop after m subtrees if the following stopping condition holds.

Lemma 3.2 (Early Termination). *Let T^i be the i-th subtree of document T in candidate score order w.r.t. query Q (breaking ties arbitrarily), R' a top-k ranking of the subtrees $T^1, T^2, \ldots, T^m$, $k \leq m < |T|$. If $\delta(R'[k]) \leq llb(Q,T^{m+1})$, then R' is a valid top-k ranking for all subtrees $T^i \in T$.*

Simple Algorithm. A simple top-k subtree similarity algorithm, Simple, that uses Lemma 3.2 and the size filter (cf. Section 2) proceeds as follows: compute the score for each subtree $T_i \in T$, $1 \leq |T_i| \leq \tau$, and sort all subtrees by score, process the subtrees in sort order, and stop when the early termination condition holds.

Running Example. Figure 1 shows an example document T, an example query Q, and the edit distance (δ) for all subtrees $T_i \in T$ w.r.t. Q. Each node is represented by its label and the postorder identifier (subscript number). In the examples, we refer to the subtree rooted in the i-th node of T in postorder as T_i.



T_i	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}	T_{16}	T_{17}
δ	3	3	3	3	2	0	3	3	2	3	4	4	3	3	4	9	13

Figure 1: Running example.

Example 3.3. We compute the top-k ranking, $k = 3$, for Q in T using Simple (cf. Figure 1). Due to the size filter, the maximum subtree size that must be considered is $\tau = 2|Q| + k = 11$. We compute the label lower bound for all subtrees T_i, $|T_i| \leq \tau$ and rank them by candidate score. For example, the label lower bound for T_9 is $llb(Q, T_9) = max\{|Q|, |T_9|\} - |\mathcal{L}(Q) \Cap \mathcal{L}(T_9)| = 2$, where $\mathcal{L}(Q) = \{\{a, b, b, x\}\}$ and $\mathcal{L}(T_9) = \{\{a, x\}\}$; the candidate score of T_9 is $score(T_9) = 1/3$. The result is shown in Table 2; we order subtrees by postorder position in the case of ties; T_{17} is not listed since $|T_{17}| > \tau$.

Table 2: Example subtrees ordered by candidate score.

$llb(Q, T_i)$	$score(T_i)$	Subtrees
0	1	T_6, T_{11}
2	1/3	T_2, T_5, T_9
3	1/4	T_1, T_3, T_4, T_7, T_8, T_{10}, T_{13}, T_{14}, T_{15}
4	1/5	T_{12}
5	1/6	T_{16}

Simple first processes T_6, T_{11}, and T_2 in this order and computes $\delta(Q, T_6) = 0$, $\delta(Q, T_{11}) = 4$, and $\delta(Q, T_2) = 3$, resulting in the intermediate ranking $R' = [T_6, T_2, T_{11}]$. Since $\delta(R'[k]) = 4$ and $llb(Q, T') = 2$ for the next unprocessed subtree $T' = T_5$, we continue and verify T_5 and T_9. $\delta(Q, T_5) = 2$, $\delta(Q, T_9) = 2$, resulting in $R' = [T_6, T_5, T_9]$. Now, $\delta(R'[k]) = 2 \leq llb(Q, T') = 3$ for the next subtree $T' = T_1$, and we can terminate.

4 INDEX AND MERGEALL ALGORITHM

We introduce the *candidate index*, which enables us to efficiently retrieve candidates in score order, and propose MergeAll, a baseline algorithm that solves top-k subtree similarity queries using our index.

4.1 Candidate Index

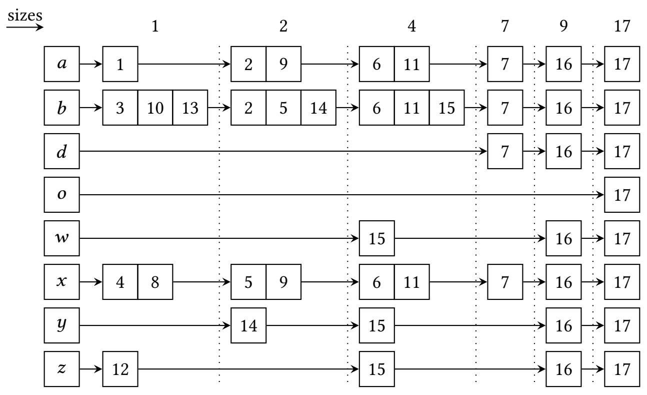
The candidate index, $\mathcal{I}$, is built over a document tree, T, and stores the following data structures:

(1) An *inverted list index* over the document labels.
(2) The *node index*, a compact representation of T.

Our index supports the following operations:

- $\mathcal{I}.list(\lambda)$ retrieves the inverted list l_λ for a label λ and returns *nil* if that list does not exist.
- $\mathcal{I}.sizes()$ retrieves all distinct subtree sizes in T.

Inverted List Index. We build an inverted list index on the document labels. For each *distinct* label $\lambda \in \mathcal{L}(T)$, we maintain a list l_λ of all subtrees that contain a node labeled λ. The inverted list entries are lexicographically sorted by subtree size and postorder identifier (ascending order). Figure 2a shows the inverted lists for our example document. A list entry is a subtree T_i, represented by the postorder identifier of its root node, i. The lists are partitioned by subtree sizes (shown above the lists).



(a) Inverted list index of T.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
lbl	*a*	*b*	*b*	*x*	*b*	*a*	*d*	*x*	*a*	*b*	*b*	*z*	*b*	*y*	*w*	*d*	*o*
size	1	2	1	1	2	4	7	1	2	1	4	1	1	2	4	9	17

(b) Node index of T.

Figure 2: Baseline index structure for document T of our running example (cf. Figure 1).

Node Index. We store the document T in an array of size $|T|$. The i-th field in the array (1-based counting) is a pair $(\lambda_i, |T_i|)$, where λ_i is the label of the i-th node of T in postorder and $|T_i|$ is the size of the subtree rooted in that node. Figure 2b shows the node index for our example document.

The node index is a lossless and compact representation of the document tree. We do not need any other representation of the document for our algorithms. Conveniently, each subtree T_i in the node index is a connected subsequence starting at position $i - |T_i| + 1$ ($|T_i|$ is accessed in constant time in the node index) and ending at position i. The subtree part of the node index is a valid tree representation by itself.

The node index is built in a single scan of the document using a SAX parser and is stored in main memory. While parsing, we build a dictionary that maps string labels to unique integers. In our indexes and algorithms (including the edit distance computation), we use integer labels (in our examples, however, we show the original string labels).

Index Size. The size of the candidate index is $O(n^2)$, $n = |T|$. Consider the tree in Figure 3 with root label λ_1, a single leaf λ_n, and pairwise distinct labels, $\lambda_i \neq \lambda_j$ unless $i = j$. The inverted list of λ_i has i entries, e.g., λ_n appears in n subtrees. The overall number of entries is $\sum_{i=1}^{n} i$, which is quadratic.

For the tree in Figure 3, also the index of StructureSearch [9] requires quadratic space. We introduce a linear-space index in Section 6.

$$\lambda_1 \text{ ——— } \lambda_2 \text{ ——— } \lambda_3 \text{ ——— } \cdots \text{ ——— } \lambda_{n-1} \text{ ——— } \lambda_n$$

Figure 3: Worst-case document for the inverted list index (root to the left, leaf to the right).

4.2 MergeAll Algorithm

MergeAll uses the candidate index and processes the subtrees T_i in the order of non-decreasing size lower bound, $slb(T_i, Q) = ||T_i| - |Q||$ (cf. Section 2), w.r.t. the query Q.

We (conceptually) split the inverted lists into vertical *stripes* as illustrated in Figure 4. A stripe S_j consists of all subtrees T_i in all lists that have size $|T_i| = |Q| + j$, e.g., S_2 and S_{-2} are the blue stripes in the figure. A *partition* consists of all subtrees of a stripe in a single inverted list, e.g., the subtrees in stripe S_0 of list l_{λ_4} form a partition (marked in the figure). Stripes and partitions may be empty.

Overview. MergeAll processes the subtrees stripe by stripe. The current stripe number is j. We leverage the fact that the size lower bound for all subtrees T_i in S_j and S_{-j} is $slb(T_i, Q) = j$. By incrementing the stripe number we process the subtrees in ascending size lower bound order.

The goal, however, is to retrieve the subtrees in non-increasing candidate score order, which is equivalent to the non-decreasing label lower bound order. We maintain a *lower bound cache* (*lbc*) that stores subtrees in buckets. A subtree T_i in stripe S_j or S_{-j} with label lower bound $lb = llb(Q, T_i)$ is cached in bucket $lbc[lb]$ for later verification if $lb > j$.

We only process lists of labels that exist in Q, $\lambda \in \mathcal{L}(Q)$, therefore we have at most $|Q|$ lists. We start at stripe $j = 0$ and proceed in four steps:

(1) Verify all subtrees in lower bound bucket $lbc[j]$.

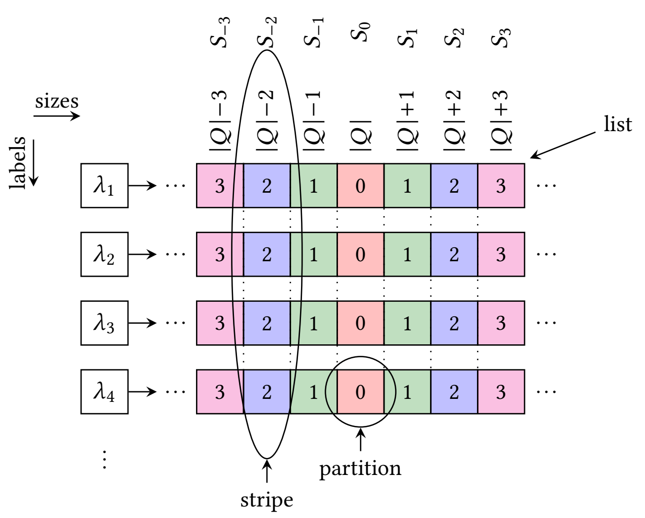


Figure 4: Stripes and partitions w.r.t. query Q.

(2) For each candidate $T_i \in S_j \cup S_{-j}$ compute $lb = llb(Q, T_i)$.
  (a) If $lb = j$, then verify T_i;
  (b) otherwise, cache T_i in lower bound bucket $lbc[lb]$.
(3) Increment to next stripe: $j \leftarrow j + 1$
(4) Continue at step (1).

Whenever we verify a subtree T_i, we also update the ranking R. Since the current stripe number j is a size lower bound for all subtrees in S_j, we can terminate if $|R| = k$ and $j \geq \delta(R[k])$. We give the pseudo code for MergeAll in the appendix (Algorithm 1).

Overlap computation. We maintain two pointers, l and r, in each list. r is initialized to the first subtree T_i (subtree with the smallest postorder identifier) of stripe S_0, l starts at position $r - 1$. If not clear from the context, we refer to the pointers of a list l_λ by $l_\lambda.l$ and $l_\lambda.r$.

We move the pointers in an n-way merge fashion to compute the label overlap with the query. We stop moving a pointer when it points to the next stripe. We first move the l pointers and maintain a counter $ol[T_i]$ for each subtree T_i that we encounter; then we move the r pointers in a similar way. After all pointers stop, the counter $ol[T_i]$ stores the overlap $|\mathcal{L}(Q) \Cap \mathcal{L}(T_i)|$. This works because our index structure sorts elements within a stripe consistently. With the overlap, we compute the label lower bound, $llb(Q, T_i) = max\{|Q|, |T_i|\} - |\mathcal{L}(Q) \Cap \mathcal{L}(T_i)| \leq \delta(Q, T_i)$.

We next discuss two special cases. (1) *Duplicate query labels.* When the query Q has duplicate labels, the list l_λ is retrieved x times if Q has x nodes with label λ. Then, for a subtree T_i we get an overlap $ol[T_i] > |\mathcal{L}(Q) \Cap \mathcal{L}(T_i)|$ if T_i has fewer than x nodes with label λ. The top-k result is still correct, but T_i may be processed too early w.r.t. to the candidate score order. To avoid this situation, we can collect all subtrees and compute their label overlap using our node index. In practice, the small violations of the candidate order have little effect, and we suggest using the merge approach.

(2) *Lists without query label.* After processing all lists of the labels in Q, one of the following situations may happen. (a) $|R| < k$, i.e., we did not find k subtrees that have a common label with Q; (b) $\delta(R[k]) > |Q|$, i.e., there may be subtrees that do not share a label with Q but should be in the ranking. In this case, we need to consider lists of labels that do not exist in Q. For all subtrees in lists of non-query labels the minimum edit distance is $|Q|$. We merge the lists stripe by stripe and use the stopping condition to terminate. This corner case rarely appears in practice.

The following theorem considers MergeAll with the fix for duplicate query labels.

THEOREM 4.1. *MergeAll solves the top-k subtree similarity problem and verifies subtrees in candidate score order.*

Example 4.2. Figure 5 illustrates MergeAll for our running example, $k = 3$. We retrieve the lists of the labels in Q: a, b (twice since b is a duplicate label), and x. We start with stripe S_0 (red stripe). Pointer r is initialized to the first subtree in S_0 (T_6 in all lists), l starts on the last subtree in the green partition. We compute the overlap by moving the pointers and merging the lists. l cannot be moved; r merges the partitions in S_0 and computes the overlaps of T_6 (4), T_{11} (4), and T_{15} (2). Note that the true overlap of T_{15} is 1; we overestimate due to the duplicate query label b. From the overlaps, we get the label lower bounds $llb(Q, T_6) = llb(Q, T_{11}) = 0$ and $llb(Q, T_{15}) = 2$. Hence, T_6 and T_{11} are verified, whereas T_{15} is cached in bucket $lbc[2]$; $R' = [T_6, T_{11}]$. For the next stripe, $j = 1$, there is nothing to do since $lbc[1]$, S_1, and S_{-1} are all empty. For $j = 2$, we first verify T_{15} in $lbc[2]$ and get the ranking $R' = [T_6, T_{11}, T_{15}]$; next we process the subtrees in stripe S_{-2} (green); S_2 is empty. The overlaps (2 for T_{14}, T_9, and 3 for T_5, T_2) are computed while l is decremented. T_{14}, T_9 are verified immediately. After T_5 is verified in the next round $j = 3$, $R = [T_6, T_5, T_9]$, and we terminate since $\delta(R[k]) \leq j$. Figure 5 illustrates the pointers after processing T_5.

5 CONE: PARTITION-BASED TRAVERSAL

MergeAll processes one stripe per round and computes the label lower bound for all subtrees in a stripe. The stripes may be large, leading to slow execution times. We observe, however, that the size of the partitions within a stripe may vary greatly. The inverted lists of frequent labels are very long (e.g., the list of the "article" tag in the DBLP bibliography), leading to large partitions. Then, the runtime is dominated by processing the partitions of long lists.

In this section we present Cone, an algorithm that addresses this issue. Cone processes only a subset of the partitions in each stripe. The inverted lists are sorted and short

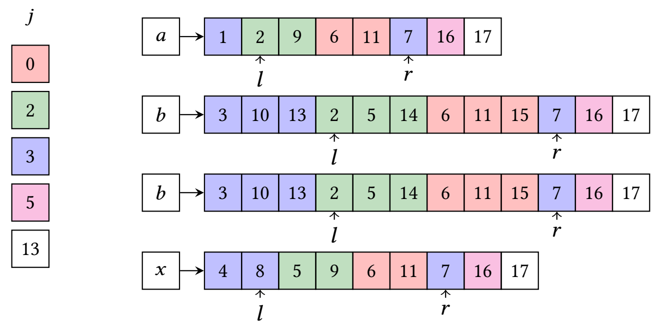


Figure 5: MergeAll after processing stripes j = 2.

lists are accessed first. Therefore, the algorithm may terminate before considering any of the large partitions. Cone uses an edit distance bound $\mathcal{B}$, which is zero initially and is incremented in each round. Only partitions that possibly contain a subtree T_i at distance $\mathcal{B}$ from query Q are considered.

Assume we know that there are $nml(T_i)$ labels in Q that do not exist in subtree T_i. We call $nml(T_i) = |\mathcal{L}(Q) \setminus \mathcal{L}(T_i)|$ the *number of missing* labels in T_i w.r.t. Q. Then we can draw conclusions on the size of T_i that is required to achieve edit distance $\mathcal{B}$.

THEOREM 5.1 (SIZE INTERVAL). *Let T_i be a subtree of document T, Q be the query tree, $nml(T_i)$ be the number of missing labels in T_i w.r.t. Q, and $\mathcal{B} \geq 0$ an edit distance bound. If $\delta(Q, T_i) \leq \mathcal{B}$, then $|T_i|$ is in the* size interval

$$si(\mathcal{B}, Q, T_i) = [|Q| - \mathcal{B}; |Q| + \mathcal{B} - nml(T_i)] \quad (3)$$

For a given edit distance bound, $\mathcal{B}$, the subtrees within the size interval are called *pre-candidates.* The Cone algorithm proceeds in rounds. In every round some additional partitions are processed. Every round examines one additional list until all lists are initialized. We call a list *initialized* if we have already processed a partition in that list.

In the first round, $\mathcal{B} = 0$, and we process the partition of subtree size $|Q|$ in the first list (cf. Theorem 5.1). The subtrees in this partition can achieve an edit distance of 0 since their size matches the query size and all labels may match (no label mismatch found so far). Notably, these are the only subtrees that can achieve edit distance 0. Subtrees in other lists have at least one missing label w.r.t. Q, and subtrees in another partition of the first list are either smaller or larger than $|Q|$.

In every round $\mathcal{B}$ is incremented and an additional list is considered (if non-initialized lists are left). For the j-th list that we process, $nml(T_i) \geq j - 1$: any new subtree T_i that we find in the j-th list has at least $j - 1$ missing labels since we have processed all subtrees of size $|T_i|$ in the previous $j - 1$ lists and did not see T_i.

We process only a subset of the partitions in a given list and round, namely the partitions that satisfy the size interval of the current round. Figure 6 illustrates this partition-based traversal.

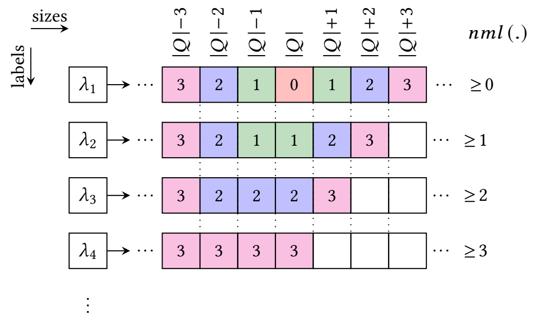


Figure 6: Cone traversal of the inverted list index in candidate score order.

The Cone algorithm distinguishes between pre-candidates and candidates. We use our index structure to generate pre-candidates. In the i-th round, $\mathcal{B} = i - 1$, and we only need to consider the first i lists in the index. Similar to MergeAll, we maintain two pointers, l and r, for each list, initialized to the partition of the subtree size closest to $|Q|$. The pointers are used to generate pre-candidates from a partition. Some pre-candidates may be promoted to candidates. A pre-candidate T_i gets promoted whenever its label lower bound is equal to $\mathcal{B}$. Candidates are verified immediately, whereas the remaining pre-candidates are stored in the lower bound cache (lbc) for verification in a later round (cf. Section 4.2).

Inverted List Ordering. Since Cone examines lists one by one, the list order is important. Different pre-candidates may be reported for different list orders, resulting in earlier/later termination as well as fewer/more label lower bound computations and verifications. Consider, for example, the lists in Figure 7 in reversed order $[l_b, l_x, l_a]$. Then, $llb\,(Q, T_{15}) = 3$ is computed in round 1 and T_{15} is cached for the round with $\mathcal{B} = 3$. Since the list length corresponds to the label frequency (a long list implies many subtrees with this label), we order the lists in ascending order by their length.

Like in MergeAll, we may not be able to produce enough candidates from the lists that share a label with the query. In this rare case, we fall back to MergeAll on all remaining lists to derive a correct ranking (cf. Section 4). The pseudo code of Cone is given in Algorithm 4 in the appendix.

Theorem 5.2. *Cone solves the top-k subtree similarity problem and verifies subtrees in candidate score order.*

Example 5.3. Figure 7 shows Cone applied on our running example, $k = 3$. The first round, $\mathcal{B} = 0$, retrieves and initializes l_a since l_a is the shortest list among all lists of the query labels. Pointer l is initialized to T_9, pointer r to T_6. Then, pre-candidates T_6, T_{11} are generated from partition 0 of l_a. Subtrees T_6 and T_{11} may match Q exactly since there is no label mismatch so far, and $|T_6| = |T_{11}| = |Q|$. Next, we compute the true label lower bounds using the node index; $llb\,(Q, T_6) = llb\,(Q, T_{11}) = 0$. Both are verified and round 1 concludes; $R' = [T_6, T_{11}]$ and $\mathcal{B}$ is incremented (lower bound cache lbc is empty). In round 2, we first process l_a again. The next partition of l_a contains subtrees of size 2 (T_9, T_2) and 7 (T_7), hence no pre-candidates are reported from l_a. Then, we initialize and process list l_b (l points to T_9, r to T_6), which does not provide us with new pre-candidates (T_6, T_{11} were already processed, indicated by the gray/green boxes). In round 3, $\mathcal{B} = 2$, T_9 and T_2 are reported from l_a, and T_5 is reported from l_b. All pre-candidates are promoted since $llb\,(Q, T_9) = llb\,(Q, T_2) = llb\,(Q, T_5) = 2$, resulting in $R' = [T_6, T_5, T_9]$. Since $\mathcal{B} \geq \delta\,(R\,[k])$, we terminate; $R = [T_6, T_5, T_9]$. Figure 7 depicts the processed list entries.

Compared to MergeAll, Cone processes only 2 lists (instead of 4) and computes only 4 label lower bounds. Notice how the presence of T_{15} in list l_b does not impose any overhead because we terminate before it is processed.

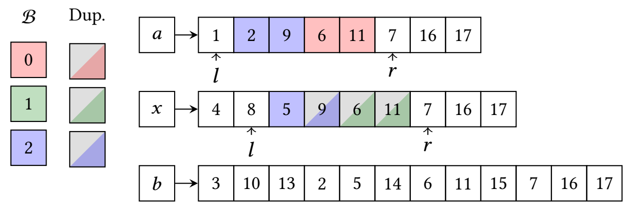


Figure 7: Processed subtrees of Cone.

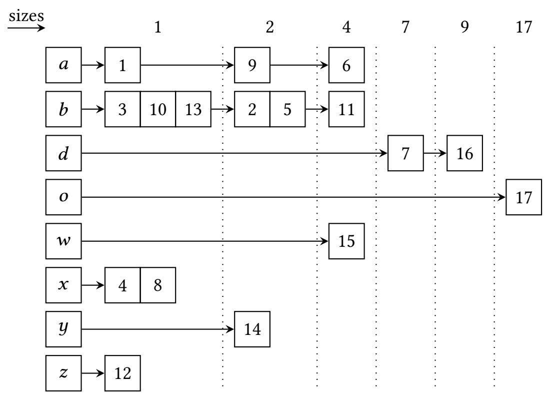
6 LINEAR SPACE INDEX AND SLIMCONE

Cone, presented in the previous section, is effective at producing candidates in score order. Unfortunately, Cone relies on an inverted list index that requires quadratic memory (in the worst case, cf. Section 4). In this section, we introduce the *slim inverted list* index, which requires only linear space, and the SlimCone algorithm that operates on the new index. SlimCone mimics Cone, but instead of scanning materialized inverted lists, relevant list parts are generated on the fly.

6.1 Indexing in Linear Space

In the worst case, the inverted list index requires quadratic space. To avoid the full materialization of the inverted lists, we introduce an implicit and lossless list representation that requires only linear space.

For a label $\lambda \in \mathcal{L}\,(T)$, the inverted list index stores every subtree that *contains* label λ. In other words, a list stores all nodes on every path from a node labeled λ up to the document root, and the paths are traversed at index build time. We propose *slim inverted lists* to avoid full list materialization and traverse paths during candidate generation. A slim inverted list (slim list) stores only nodes *labeled* λ (i.e., the start of a path). For the path traversals (upwards, towards the root node), we extend the node index (cf. Section 4.1) with parent pointers. This information enables us to reconstruct paths on the fly. Figure 8 depicts the slim inverted list index and the slim-extended node index of our running example.



(a) Slim inverted list index.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
lbl	*a*	*b*	*b*	*x*	*b*	*a*	*d*	*x*	*a*	*b*	*b*	*z*	*b*	*y*	*w*	*d*	*o*
size	1	2	1	1	2	4	7	1	2	1	4	1	1	2	4	9	17
par	2	7	6	5	6	7	17	9	11	11	16	15	14	15	16	17	0

(b) Slim-extended node index.

Figure 8: Linear-space index for example document T.

6.2 The SlimCone Algorithm

We propose a new algorithm, SlimCone, that generates candidates in score order from slim inverted lists. Since we push the path traversals into the candidate generation phase, SlimCone needs to walk up paths at query time using the slim-extended node index. SlimCone is also round-based ($\mathcal{B}$ is incremented in each round, starting with 0) and implements the Cone traversal on top of our slim inverted list index.

Cone can perform a binary search on the inverted lists to find the starting partitions. With slim lists, this approach would consider only nodes labeled λ, but there may be larger subtrees on the respective paths to the root. Slim lists do not store these subtrees explicitly. To generate correct pre-candidates, we need to traverse the respective paths for each entry of a slim list that represents a subtree smaller than Q. Notably, we may not need to traverse the paths completely, but only until we encounter a subtree T_i with size $|T_i| \geq |Q|$.

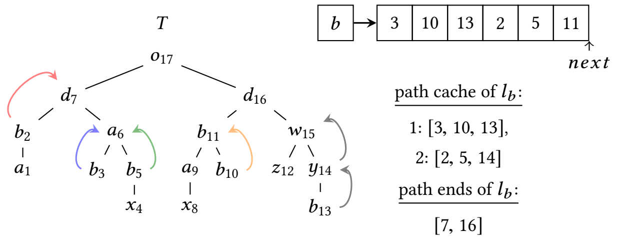


Figure 9: Finding the starting point of a slim list.

For the path traversal, we retrieve all node identifiers from the slim list at which a subtree T_i with $|T_i| < |Q|$ is rooted. For each identifier, we look up its parent in the node index and follow the path until the parent's subtree size is greater than or equal to $|Q|$. If the parent's subtree size is $|Q|$, then the parent is in the first partition; we immediately compute the label lower bound w.r.t. Q and verify the subtree if the label lower bound matches $\mathcal{B}$.

We keep track of the *path ends* (*pe*) for each slim list since we may need to continue the upward traversal in a later round. If the last node on the path roots a subtree that was verified, we store its parent in the path ends. Furthermore, we maintain a *path cache* (*pc*) for each slim list that stores all node identifiers on a path with the size of the subtree they root. This avoids redundant traversals of the same path. Details on path cache and path ends are given below.

We may also need to examine additional list entries. Therefore, we store a single pointer for each list, *next*, which points to the next unprocessed list entry and is advanced whenever subtrees larger than Q are examined.

Note that the paths of *all* nodes that root a subtree T_i with $|T_i| < |Q|$ need to be traversed to generate all pre-candidates. While our algorithm climbs up all paths, it visits all nodes that root subtrees that are part of the corresponding full inverted list. Since we stop the traversal when we find a subtree root i s.t. $|T_i| \geq |Q|$, we construct the corresponding inverted list only partially. Figure 9 exemplifies this concept for example list l_b.

We discuss the main concepts used by SlimCone to generate candidates in non-increasing score order.

Path Caching. The path cache (*pc*) stores a bucket for each subtree size that we encounter during the path traversals. In bucket b, we collect all roots of subtrees T_i s.t. $|T_i| = b$. This is necessary due to the vertical list expansion. Without the path cache, we would need to traverse the path downwards again. Hence, we reuse the path information in later rounds. If we need to consider smaller subtrees, we do a lookup in the path cache. This provides us with a (possibly empty) set of subtree roots, which contains all nodes that belong to a certain partition.

Path Ends. We need to book-keep information about path ends (*pe*) for each slim list. After successfully climbing up a path to the first node at which a subtree T_i with $|T_i| \geq |Q|$ is rooted, we need to store the last node identifier on the path. This is due to the list expansion towards larger subtrees (w.r.t. $|Q|$). Therefore, for each slim list, we maintain a sequence of node identifiers, each of which represents the current end of a path. By storing these node identifiers, we can continue the upward path traversal in later rounds, if necessary.

List Ordering. To be consistent with the list ordering of Cone, we order the lists in SlimCone like in Cone, i.e., by increasing length. For each slim list, we compute and store the length of the corresponding full inverted lists. We refer to this value as *full list length*.

Similar to Cone, we use MergeAll on all lists of labels that are not in $\mathcal{L}(Q)$ to derive a correct ranking for the case that SlimCone produces too few results. Algorithm 6 in the appendix shows the pseudo code of SlimCone.

Theorem 6.1. *SlimCone solves the top-k subtree similarity problem and verifies subtrees in candidate score order.*

Example 6.2. In Figure 10, we illustrate SlimCone for our running example, $k = 3$. Similar to Cone, we retrieve slim list l_a since its the shortest w.r.t. the full list lengths. The initialization for l_a now differs from Cone: we climb up the paths of all entries of l_a since the subtree sizes are smaller than or equal to $|Q|$. This results in the path cache and path ends of l_a shown in Figure 10. During the traversal, we find T_{11} and T_6 (in this order) having $|T_6| = |T_{11}| = |Q|$. Consequently, we compute $llb(Q, T_6) = llb(Q, T_{11}) = 0$ and verify both, $\delta(Q, T_6) = 0$, $\delta(Q, T_{11}) = 4$. This results in $R' = [T_6, T_{11}]$. Note that after examining T_6 and T_{11}, we traverse to their respective parents. Therefore, T_{16} is added to the path ends of l_a. No new pre-candidates are processed in round 2. However, list l_x is retrieved and initialized, resulting in the path cache and path ends of l_x depicted in Figure 10. Since we have already stored the node identifiers 9, 7, and 16 during initialization of l_a, neither 9 is added to the path cache nor 7, 16 are added to the path ends of l_x. In round 3, $\mathcal{B} = 2$, we process bucket 2 of the path cache of l_a, generating the pre-candidates T_2 and T_9. We compute $llb(Q, T_2) = llb(Q, T_9) = 2$, verify T_2 and T_9, and update $R' = [T_6, T_9, T_2]$. Analogously, we process T_5 from the path cache of l_x. This results in $R = [T_6, T_5, T_9]$ and we terminate.

7 EFFICIENT INDEX UPDATES

We extend the slim index to support incremental updates. We support the standard node edit operations as defined in Section 2: rename, delete, and insert. Updates affect the inverted list index and the node index.

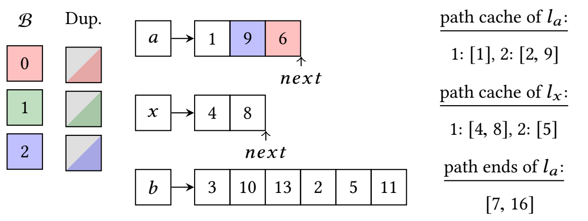


Figure 10: Slim lists, path caches, and path ends.

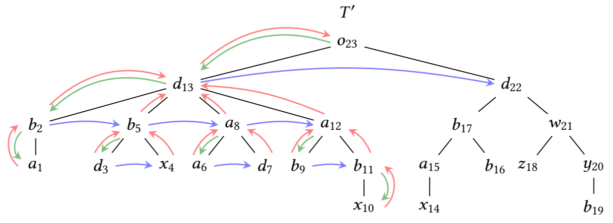
Updating the Inverted List Index. The position of a node in the inverted list index is determined by its label and the size of its subtree. The rename operation changes the label of a node, which requires us to remove the node from the list of the old label and insert it into the list of the new label. Insert (delete) changes the subtree size of all ancestors of the inserted (deleted) node, and we must add the new node into the respective list (remove the deleted node). We implement a slim inverted list as a balanced search tree (ordered by subtree size), thus requiring only $O(\log l)$ time to find, insert, or delete a node from a list of length l. No further changes are required to the slim inverted list index in response to node edit operations on the document. Space and runtime complexity at query time are not affected.

Dynamic Node Index. The node index encodes the labels and the structure of the document T. At query time, we need to efficiently support the following operations: (1) access a node by its identifier, (2) reconstruct a subtree (or the label set of a subtree) given its root node. A subtree is reconstructed by traversing all its nodes in postorder (cf. Section 4.1).

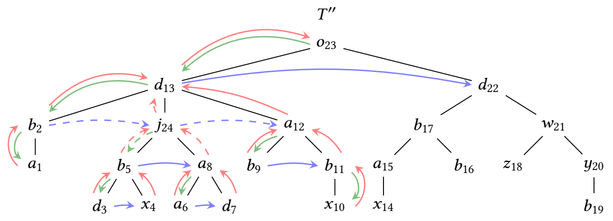
(1) In the static node index, we identify a node by its postorder position. In our dynamic version of the slim-extended node index, we allow arbitrary node identifiers. The node index is stored in an array and the identifier of a node matches the array position, thus a node is accessed in constant time. To ensure a compact representation, we use consecutive identifiers and maintain a free list to reuse array positions after node deletions.

(2) To reconstruct a subtree given its root node, we store two additional fields for each node v: first child, $c1(v)$; next (right) sibling, $sib(v)$. These fields also allow us to efficiently traverse all nodes of a subtree in postorder.

We discuss the effect of updates on the dynamic node index. *Rename:* The node is accessed via its identifier and the label is changed in constant time. *Insert:* The insert operation adds a new node u as the i-th child of an existing node p, replacing a (possibly empty) sequence $C = (c_i, c_{i+1}, \ldots, c_j)$ of p's children, and the child sequence is connected under the new node u. We need to insert a new node into the index; the identifier of the new node matches its position in the node index array (new nodes are appended or fill a gap



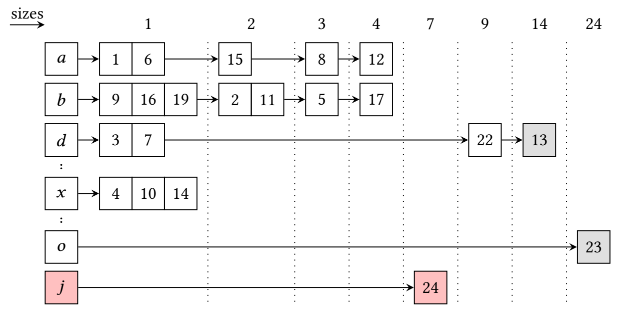
(a) Example tree before node insertion (T').



(b) Example tree after inserting node j_{24} (T'').

	1	2	3	4	5	6	7	8	9	10	11	12	13		23	24
lbl	*a*	*b*	*d*	*x*	*c*	*u*	*p*	*v*	*b*	*x*	*b*	*a*	*d*		*o*	*j*
size	1	2	1	1	3	1	1	3	1	1	2	4	14		24	7
par	2	13	5	5	24	8	8	24	12	11	12	13	23	··	0	13
*c*1	0	1	0	0	3	0	0	6	0	0	10	9	2		13	5
sib	0	24	4	0	8	7	0	0	11	0	0	0	22		0	12

(c) Dynamic node index of example tree T''.



(d) Slim inverted list index of example tree T''.

Figure 11: Index update example.

resulting from an earlier deletion). The following existing nodes must be updated. (a) Ancestors of the inserted node u: The subtree sizes are incremented by one; if u is the first child of its parent p, the first child pointer, $c1(p)$, is updated. (b) (Former) children of p: The parent pointer of all nodes in C is updated; the next sibling pointers of c_j and (if $i > 1$) c_{i-1} are updated. To insert a new root node, we assume a virtual node with identifier zero, which is treated as the parent of the actual root node. *Delete* is the reverse of insert. The positions of deleted nodes are registered in the free list.

Figure 11 illustrates the slim inverted lists and the dynamic node index after inserting a new node into an example tree T'. Only the dashed pointers in T'' (colored fields in slim lists and dynamic node index) need to be updated. Red, green, and blue pointers (fields) denote the parent, first child, and next sibling pointers (fields), respectively. Changes to subtree sizes are highlighted in gray. Overall, the complexity of updating the node index is $O(d + f)$, where d is the depth and f the maximum fanout of a node in the document. In many real datasets, f and d are small compared to the document size. We show the efficiency of updates in our experiments.

8 RELATED WORK

Top-k Subtree Similarity Queries. TASM-Dynamic [4, 31], a simple solution for the top-k subtree similarity problem, computes the edit distance between the query Q and the entire document T using dynamic programming. As a side product, the edit distances between the query and all subtrees of the document are computed. This approach requires $O(|Q|^2 |T|)$ time and $O(|Q| |T|)$ space [11]. Augsten et al. [4, 5] show that the maximum subtree size that must be considered is $\tau = 2|Q| + k$. They develop the TASM-Postorder algorithm that runs in $O(|Q|^2 + |Q| k)$ space, i.e., the memory is independent of the document size. TASM-Postorder does not use an index and must scan the document for each query. We empirically compare our solution to TASM-Postorder.

Cohen [9] proposes StructureSearch, the first index-based method for top-k subtree similarity queries. The index identifies repeating subtree patterns to reduce the number of redundant edit distance computations. StructureSearch does not need to scan the document at query time and outperforms TASM-Postorder in terms of runtime. However, StructureSearch requires a large index, which can be quadratic in the document size. The document is the database, which may be large (e.g., SwissProt has $|T| = 479$M nodes). Our SlimCone algorithm requires only a linear-size index. We empirically compare StructureSearch to SlimCone. Our solution builds a smaller index, building the index is faster, and in most settings we outperform StructureSearch in terms of query response time, often by orders of magnitude.

XML Indexing Techniques. Inverted lists and data structures similar to our node index have also been used to index XML documents [16, 20]. These works solve a different problem (answering resp. ranking XPath queries) and do not consider the tree edit distance. Further, our index access methods are different: we access the inverted lists partition by partition based on an edit distance bound and build the partitions on the fly while accessing them.

Tree Edit Distance. The classical tree edit distance algorithm by Zhang and Shasha [31] runs in $O(n^4)$ time and $O(n^2)$ space for trees with n nodes; for flat trees of depth $O(\log n)$ the algorithm runs efficiently in $O(n^2 \log^2 n)$ time. Bille [7] surveys classical edit distance algorithms. Newer developments include the algorithm by Demaine et al. [11], which reduces the runtime to $O(n^3)$, and AP-TED$^+$ by Pawlik and Augsten [22]. AP-TED$^+$ analyzes the input trees and dynamically computes the optimal evaluation strategy. While the runtime complexity remains cubic, this worst case can often be avoided. Despite all efforts, computing the edit distance remains expensive. We introduce the candidate score to rank subtrees, verify promising candidates first, and thus reduce the number of expensive edit distance computations.

Related Problem Definitions. Related but different problems include, for example, XML duplicate detection [8, 23], approximations of the tree edit distance [6, 30], tree similarity joins [24], top-k similarity joins for sets [29], and top-k queries over relational data [18].

Cohen et al. [10] introduce a top-k algorithm that works for both ordered and unordered trees. In near linear time, the algorithm retrieves the top-k subtrees in a document w.r.t. a so-called profile distance function. A profile distance function projects tree features to a multiset and evaluates the distance between feature multisets; examples include pq-grams [6] and binary branches [30]. The algorithm by Cohen et al. [10] solves a related but different problem and does not provide edit distance guarantees on the ranking.

In their TA algorithm, Fagin et al. [12, 13] process ranked lists of items sorted by some local score. The global score of an item is computed based on the respective local scores. The goal is to find the k items with the highest global score. Akbarinia et al. [2] improve the efficiency of TA by minimizing the number of list accesses. Theobald et al. propose TA-based solutions to answer probabilistic top-k queries [27], efficiently expand queries [26], and build an efficient top-k query processing system for semi-structured data [25]. We introduce the candidate score on subtrees, but we do not merge ranked lists. The challenge in our setting is to rank candidates efficiently and produce the head of the ranked list without generating the tail.

9 EMPIRICAL EVALUATION

We empirically compare our solutions to two state-of-the-art algorithms on both synthetic and real-world data. We vary document size, query size, and k, and measure query time, indexing time, main memory, and the number of verifications.

9.1 Setup & Data Sets

Setup. All experiments were conducted on a 64-bit machine with 8 Intel(R) Xeon(R) CPUs E5-2630 v3, 2.40GHz, 20MB L3 cache (shared), 256KB L2 cache (per core), and 96GB of RAM, running Debian 8.11, kernel 3.16.0-6-amd64. We compile our code with `clang` (ver. 3.5.0-10) at maximum optimization level (`-O3`). Although we have multiple cores, we run all experiments single-threaded with no other load on the machine. We measure the runtime with `getrusage`[2] (sum of user and system CPU time). Each runtime measurement is an average over five runs. We measure main memory as the heap peak value provided by the `libmemusage.so` library[3] (preloaded using the `LD_PRELOAD` environment variable).

Data Sets and Queries. We use the XMark benchmark to generate synthetic data sets of five different sizes. Additionally, we run experiments on three real-world data sets: TreeBank[4] (TB), DBLP[5], and SwissProt[6] (SP). Important data set characteristics are summarized in Table 3. XMark, DBLP, and SwissProt were also used in previous work [9], although only small subsets of DBLP and SwissProt were used; we process the full data sets. From each of the data sets (documents, T), we randomly extract four different queries, Q, with 4, 8, 16, 32, 64 nodes, respectively. We also vary the result size, k.

Table 3: Data set characteristics.

Name	Size T [MB]	Size [Nodes] $\|T\|$	Size [Nodes] avg. $\|T_i\|$	# diff. labels
XMark1	112	$3.6 \cdot 10^6$	6.2	$510 \cdot 10^3$
XMark2	223	$7.2 \cdot 10^6$	6.2	$822 \cdot 10^3$
XMark4	447	$14.4 \cdot 10^6$	6.2	$1.3 \cdot 10^6$
XMark8	895	$28.9 \cdot 10^6$	6.2	$1.9 \cdot 10^6$
XMark16	1,790	$57.8 \cdot 10^6$	6.2	$2.9 \cdot 10^6$
TreeBank	83	$3.8 \cdot 10^6$	8.4	$1.4 \cdot 10^6$
DBLP	2,161	$126.5 \cdot 10^6$	3.4	$21.6 \cdot 10^6$
SwissProt	6,137	$479.3 \cdot 10^6$	5.1	$11.4 \cdot 10^6$

Algorithms. We compare our algorithms Merge, Cone, Slim (cf. Sections 4–6) to the state-of-the-art algorithms TASMPostorder [4, 5] (Tasm, fastest index-free algorithm) and StructureSearch [9] (Struct, fastest algorithm with precomputed index). Slim-Dyn refers to the version of Slim with incremental update support (cf. Section 7). All algorithms were implemented in C++11. We maintain the node labels in a dictionary and replace string labels by integers. All indexes reside in main memory. For computing the tree edit distance, we use the algorithm by Zhang & Shasha [31], which is efficient for flat trees (depth $O(\log n)$, as is typically the case in XML).

[2]http://man7.org/linux/man-pages/man2/getrusage.2.html
[3]http://man7.org/linux/man-pages/man1/memusage.1.html
[4]https://www.seas.upenn.edu/~pdtb/
[5]https://dblp.uni-trier.de/xml/dblp.xml.gz
[6]ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/knowledgebase/complete/uniprot_sprot.xml.gz

Space-efficient StructureSearch. Cohen [9] implements the StructureSearch algorithm using uncompressed Dewey labels (Struct-Dewey in our experiments), which leads to large indexes of about 10 times the document size (in MB); in the worst case, the index size is quadratic in the document size since each Dewey label may be of linear size. Cohen suggests to compress the Dewey labels to improve space performance. We take a different approach and use preorder, postorder, and parent (preorder) identifiers to (1) verify ancestor relationships and (2) to generate the ancestor path of a node. Given the pre- and postorder identifiers of two nodes u and v, u is an ancestor of v if and only if $pre(v) > pre(u) \land post(v) < post(u)$. We efficiently generate the path between a node v and its ancestor u using the parent pointers. Thus, the node identifiers in our space-efficient implementation (Struct) have constant size and we need not deal with compressed Dewey labels to verify node relationships or generate ancestor paths. In our experiments, we show that we substantially reduce the index size w.r.t. the original implementation.

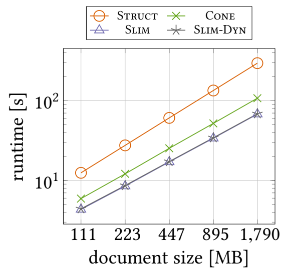
Struct assumes XML and distinguishes common and uncommon labels. Inner nodes and the x most frequent leaf nodes of an XML document are considered common. Further, Struct has a maximum edit bound y. The top-k ranking of Struct contains only subtrees with a maximum edit distance of y. All data sets in our tests are available in XML, thus we configure Struct as suggested by Cohen [9] and set $x = 1000$, $y = |Q|$. Our algorithms do not require any parameters.

9.2 Indexing

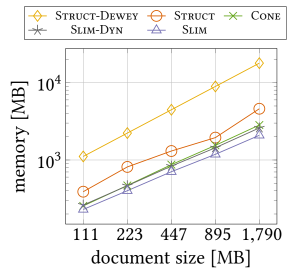
We compare the indexes of Struct, Cone, Slim, and Slim-Dyn in terms of size (all memory-resident index structures including the document) and runtime to build the index. With Struct-Dewey we refer to the original implementation of Struct by Cohen [9], which uses uncompressed Dewey labels. The index of Merge is identical to the index of Cone and is not shown separately; Tasm does not build an index.

The results are shown in Figure 12. For Struct-Dewey, we estimate the index size based on the instructions of Cohen [9] (index size is about 10 times the document size). Our space-efficient implementation of Cohen's algorithm (Struct) substantially improves the memory and requires only about 2–5 times the document size (except for TB). Cone and Slim clearly outperform Struct both in terms of index size and runtime for building the index. Even the space-efficient implementation of Struct requires at least 1.5–3 times more memory than Slim. Except for DBLP and TB, the index size of Slim is within two times the document size. Among our algorithms, Slim is faster and builds a smaller index. This is expected since Slim indexes each node once, while Cone may index each node multiple times. In the worst case, when the depth of the document grows linearly with its size, the index of Cone grows quadratically; this is not the case for the documents in our test. The size of Slim-Dyn (which supports incremental updates) is similar to the size of the space-efficient implementation of Struct (which does not support updates), but builds much faster.

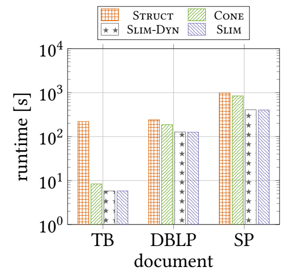
Incremental Updates. We compare the time to incrementally update the slim index to the time of building the static slim index from scratch (Slim-From-Scratch). Figure 12e and Figure 12f show the results for the XMark8 and the DBLP data sets, respectively. We randomly rename or delete nodes in the document. Insertion is similar to deletion in that it reverses the index updates of a delete operation. The update time is linear in the number of updates for both rename and deletion. As expected, deletion takes slightly more time than rename since all ancestors and children of the deleted node must be updated. The break even point for building the index from scratch is at about 10^5 deletions / $5 \cdot 10^5$ renames for XMark8 and 10^4 deletions / 10^5 renames for DBLP.



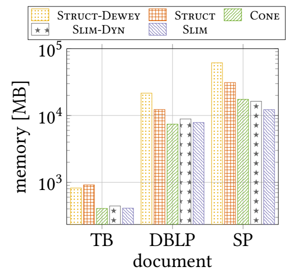
(a) Build time, XMark.



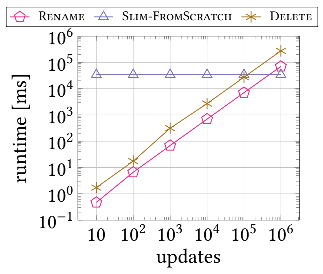
(b) Index size, XMark.



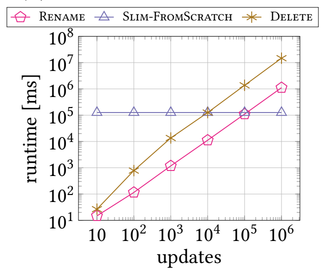
(c) Build time, real world.



(d) Index size, real world.



(e) Update time, XMark8.



(f) Update time, DBLP.

Figure 12: Build time, index size, and update time.

9.3 Effectiveness and Query Time

We evaluate our algorithms that process the subtrees in candidate score order (MERGE, CONE, SLIM, SLIM-DYN, cf. Figure 13). Since SLIM and SLIM-DYN are the same algorithms operating on different indexes, we only discuss SLIM. Supporting updates only marginally affects the query time for varying query, document, and result size (cf. SLIM-DYN in Figures 13–16). MERGE needs to verify many more candidates and is consistently slower than its competitors. This confirms the effectiveness of the clever list traversal used by CONE and SLIM. In some cases, CONE is faster than SLIM since SLIM must build the lists on the fly; we measure the largest difference for DBLP, where SLIM must traverse many paths to initialize the inverted lists. The number of verifications is the same for both algorithm. Overall, the runtime difference is small in most cases, thus SLIM pays a low price for reducing the memory complexity from quadratic to linear.

Next, we compare SLIM to the two state-of-the-art approaches with precomputed index (STRUCT) resp. without index (TASM). The query time increases with the document size for all solutions except SLIM (cf. Figure 14). The runtime of SLIM may even decrease with the document size. Larger documents have more subtrees, therefore there is a better chance to fill the ranking with good matches and terminate early. For example, the number of verifications decreases between XMark1 and XMark2. SLIM builds and traverses only the relevant parts of the lists and is therefore efficient for large documents. Also for STRUCT, the number of verifications is independent of the document size, but in absolut numbers SLIM verifies between two and three orders of magnitude fewer candidates. Further, the runtime of STRUCT substantially increases with the document size. Overall, SLIM is up to three orders of magnitude faster than STRUCT. Notably, SLIM is beneficial for a single query even without precomputed index (cf. SLIM-NOINDEX in Figures 14a and 14c).

SLIM outperforms its competitors also when the query size increases (Figure 15). Note the small number of verifications in Figure 15b: for $|Q| = 8$, SLIM verifies only k candidates, which is optimal (only subtrees that appear in the final ranking are verified). This confirms the effectiveness of the score order and the clever list traversal in SLIM. STRUCT resp. TASM must verify at least two resp. three orders of magnitude more candidates (except for TB, $|Q| = 64$). The runtime of SLIM on XMark8 is always below $1s$ ($|Q| = 4$ and $|Q| = 8$: below $1ms$), whereas the best competitor, STRUCT, runs for at least $1s$ and up to $8s$. The results on our real-world data sets lead to similar conclusions; only on DBLP STRUCT is slightly faster.

In Figure 16, we vary the result size k. All algorithms produce more candidates since the lower bound computed from the top-k ranking is looser when k is larger (and thus the subtree at position k in the ranking is less similar to the query). SLIM benefits from the small candidate set for small values of k and achieves runtimes between $0.1ms$ and $1s$ in the range $k = 1$ to $k = 100$. STRUCT must verify many more candidates than SlimCone. Although in STRUCT the number of verifications for $k = 1$ is by orders of magnitude smaller than for $k = 100$, the runtime improves only marginally. STRUCT retrieves many subtrees from the index that are filtered before they are verified; the number of retrieved subtrees does not depend on k and may be much larger than the number of verifications. SLIM does not incur this overhead: candidates are processed partition by partition, and more promising partitions are processed first. Except for DBLP, SLIM outperforms STRUCT on all k values except $k = 10000$. For $k = 10000$, both algorithms must verify many subtrees. STRUCT groups subtrees into equivalence classes of subtrees and verifies only one representative in each class, thus saving edit distance computations. This verification technique is orthogonal to the candidate generation and could also be adopted in SLIM.

10 CONCLUSION

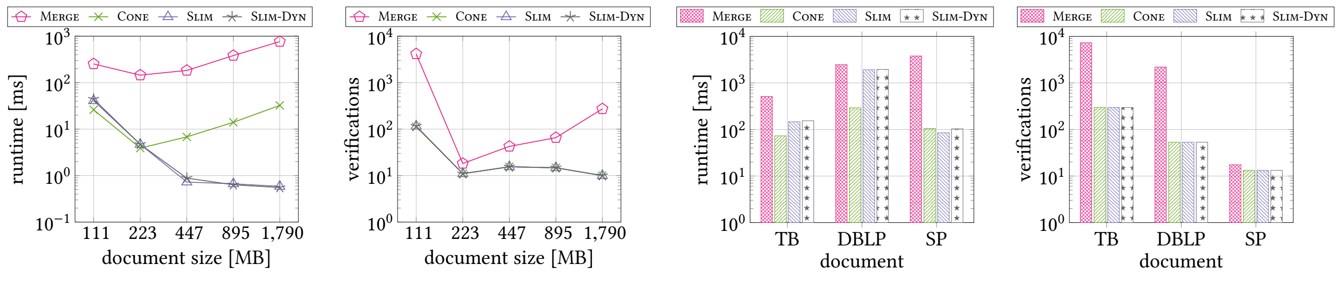
In this paper, we introduced a novel indexing technique for top-k subtree similarity queries, which retrieve the k most similar subtrees in a document T w.r.t. a query tree Q. We proposed the first incrementally updatable, linear-space index to solve this problem. Previous solutions either scan the entire document, or they build an index, but the index is static (not updatable) and large (quadratic in the document size in the worst case). The document is the database on which we compute the top-k query. Computing a quadratic-size index is not feasible for large documents.

We proposed the candidate score, which sorts subtrees such that more promising subtrees appear earlier in the sort order. We could show that processing subtrees in candidate score order substantially reduces the number of items that must be processed and verified. We developed SlimCone, a novel algorithm that leverages our linear-size index to efficiently retrieve candidates in non-increasing candidate score order. SlimCone is not tailored to XML (like previous work) and does not require any tuning parameters.

Our experiments confirmed the effectiveness of our techniques. SlimCone outperformed the state of the art in almost all scenarios w.r.t. memory consumption, number of verifications, indexing time, and query time, often by multiple orders of magnitude.

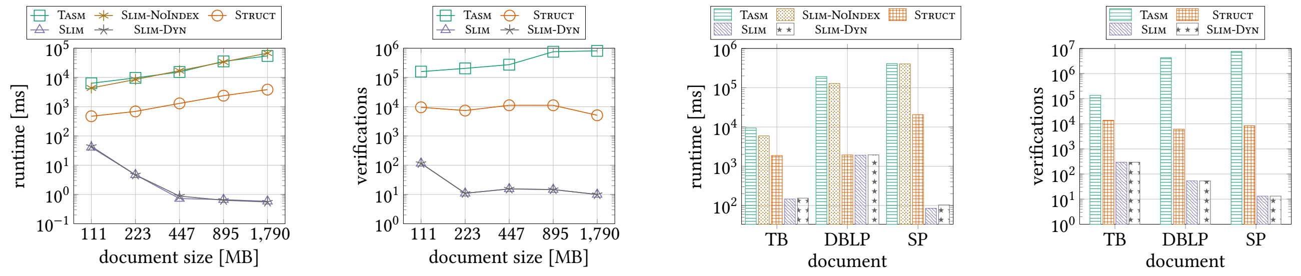
ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments and Mateusz Pawlik, Thomas Hütter, and Willi Mann for valuable discussions. This work was supported by Austrian Science Fund (FWF): P 29859.



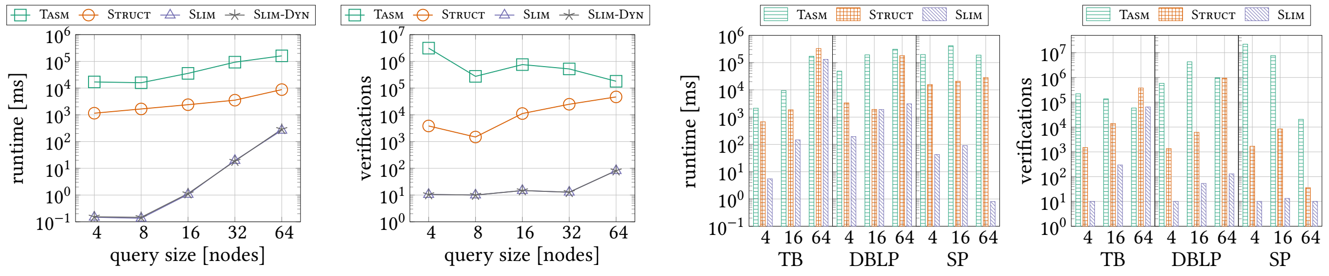
(a) Query time, XMark. (b) Verifications, XMark. (c) Query time, real world. (d) Verifications, real world.

Figure 13: Merge, Cone, Slim: Query time and number of verifications over document size, k=10, |Q|=16.



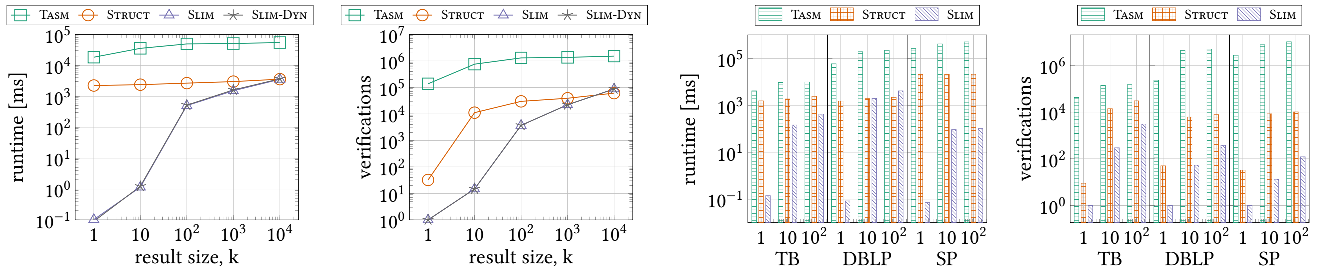
(a) Query time, XMark. (b) Verifications, XMark. (c) Query time, real world. (d) Verifications, real world.

Figure 14: State of the art vs. Slim: Query time and number of verifications over document size, k=10, |Q|=16.



(a) Query time, XMark8. (b) Verifications, XMark8. (c) Query time, real world. (d) Verifications, real world.

Figure 15: State of the art vs. Slim: Query time and number of verifications over query size |Q|, k=10.



(a) Query time, XMark8. (b) Verifications, XMark8. (c) Query time, real world. (d) Verifications, real world.

Figure 16: State of the art vs. Slim: Query time and number of verifications over varying result size k, |Q|=16.

REFERENCES

[1] Simian - Similarity Analyzer | Duplicate Code Detection for the Enterprise. https://www.harukizaemon.com/simian/. Accessed: 2019-02-11, 02:29 PM.

[2] R. Akbarinia, E. Pacitti, and P. Valduriez. Best position algorithms for top-k queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 495–506. VLDB Endowment, 2007.

[3] T. Akutsu. Tree Edit Distance Problems: Algorithms and Applications to Bioinformatics. *IEICE Transactions on Information and Systems*, E93.D(2):208–218, 2010.

[4] N. Augsten, D. Barbosa, M. Böhlen, and T. Palpanas. TASM: Top-k Approximate Subtree Matching. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 353–364, March 2010.

[5] N. Augsten, D. Barbosa, M. Bohlen, and T. Palpanas. Efficient Top-k Approximate Subtree Matching in Small Memory. *IEEE Transactions on Knowledge and Data Engineering*, 23(8):1123–1137, Aug 2011.

[6] N. Augsten, M. Böhlen, and J. Gamper. Approximate Matching of Hierarchical Data Using pq-grams. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 301–312. VLDB Endowment, 2005.

[7] P. Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1-3):217–239, 2005.

[8] P. Calado, M. Herschel, and L. Leitão. An overview of XML duplicate detection algorithms. In *Soft Computing in XML Data Management - Intelligent Systems from Decision Making to Data Mining, Web Intelligence and Computer Vision*, volume 255 of *Studies in Fuzziness and Soft Computing*, pages 193–224. Springer, 2010.

[9] S. Cohen. Indexing for Subtree Similarity-Search Using Edit Distance. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 49–60, New York, NY, USA, 2013. ACM.

[10] S. Cohen and N. Or. A General Algorithm for Subtree Similarity-Search. In *2014 IEEE 30th International Conference on Data Engineering*, pages 928–939, March 2014.

[11] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Transactions on Algorithms*, 6(1), 2009.

[12] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, pages 102–113, New York, NY, USA, 2001. ACM.

[13] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, June 2003.

[14] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM.

[15] J. Finis, R. Brunel, A. Kemper, T. Neumann, F. Färber, and N. May. DeltaNI: An Efficient Labeling Scheme for Versioned Hierarchical Data. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 905–916, New York, NY, USA, 2013. ACM.

[16] T. Grust. Accelerating xpath location steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 109–120, New York, NY, USA, 2002. ACM.

[17] C. Herrbach, A. Denise, and S. Dulucq. Average complexity of the jiang-wang-zhang pairwise tree alignment algorithm and of a rna secondary structure alignment algorithm. *Theoretical Computer Science*, 411(26):2423 – 2432, 2010.

[18] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4):11:1–11:58, 2008.

[19] M. Kashkoush and H. ElMaraghy. Matching bills of materials using tree reconciliation. *Procedia CIRP*, 7:169 – 174, 2013. Forty Sixth CIRP Conference on Manufacturing Systems 2013.

[20] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 779–790, New York, NY, USA, 2004. ACM.

[21] F. Li, H. Wang, J. Li, and H. Gao. A Survey on Tree Edit Distance Lower Bound Estimation Techniques for Similarity Join on XML Data. *SIGMOD Rec.*, 42(4):29–39, Feb. 2014.

[22] M. Pawlik and N. Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157 – 173, 2016.

[23] S. Puhlmann, M. Weis, and F. Naumann. XML duplicate detection using sorted neighborhoods. In *International Conference on Extending Database Technology (EDBT)*, volume 3896 of *Lecture Notes in Computer Science*, pages 773–791. Springer, 2006.

[24] Y. Tang, Y. Cai, and N. Mamoulis. Scaling Similarity Joins over Tree-structured Data. *Proc. VLDB Endow.*, 8(11):1130–1141, July 2015.

[25] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum. Topx: efficient and versatile top-k query processing for semistructured data. *The VLDB Journal*, 17(1):81–115, Jan 2008.

[26] M. Theobald, R. Schenkel, and G. Weikum. Efficient and Self-tuning Incremental Query Expansion for Top-k Query Processing. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '05, pages 242–249, New York, NY, USA, 2005. ACM.

[27] M. Theobald, G. Weikum, and R. Schenkel. Top-k Query Evaluation with Probabilistic Guarantees. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 648–659. VLDB Endowment, 2004.

[28] M. L. A. Vidal, A. S. da Silva, E. S. de Moura, and J. a. M. B. Cavalcanti. Structure-driven crawler generation by example. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '06, pages 292–299, New York, NY, USA, 2006. ACM.

[29] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k Set Similarity Joins. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 916–927, Washington, DC, USA, 2009. IEEE Computer Society.

[30] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity Evaluation on Tree-structured Data. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 754–765, New York, NY, USA, 2005. ACM.

[31] K. Zhang and D. Shasha. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.*, 18(6):1245–1262, Dec. 1989.

A APPENDIX

A.1 Proofs

Lemma 3.2 (Early Termination). *Let T^i be the i-th subtree of document T in candidate score order w.r.t. query Q (breaking ties arbitrarily), R' a top-k ranking of the subtrees $T^1, T^2, \ldots, T^m$, $k \leq m < |T|$. If $\delta(R'[k]) \leq llb(Q, T^{m+1})$, then R' is a valid top-k ranking for all subtrees $T^i \in T$.*

Proof. Due to the candidate score order and Def. 3.1, $\delta(R'[k]) \leq llb(Q, T^j)$ for all $j > m$; since $llb(Q, T^j) \leq \delta(Q, T^j)$, no subtrees T^j can improve the ranking. □

Theorem 4.1. *MergeAll solves the top-k subtree similarity problem and verifies subtrees in candidate score order.*

Proof. *Correctness*: The stopping condition, $|R| = k \wedge j \geq \delta(R[k])$, is correct since all subtrees T_i in partitions that are not processed have size lower bound $slb(Q, T_i) \geq j$ and can therefore not improve the ranking. *Candidate score order*: We increment stripe number j, starting with $j = 0$. For a given j, we perform two steps: (1) We postpone the verification of subtrees $T_i \in S_j \cup S_{-j}$ for which $x = llb(Q, T_i) > j$ and cache them in $lbc[x]$. (2) We verify (a) all subtrees $T_i \in S_j \cup S_{-j}$ for which $llb(Q, T_i) = j$ *and* (b) all subtrees $T_i \in lbc[j]$ (cached subtrees from previous stripes, $j' < j$). Thus, all subtrees T_i of the stripes $j' < j$ with $llb(Q, T_i) = j$ are verified when we process stripe j. There exists no subtree T_i with $llb(Q, T_i) = j$ in some stripe $S_{j''}$, $j'' > j$, since $llb(Q, T_i) \geq slb(Q, T_i) = j'' > j$ for all subtrees in $S_{j''}$. □

Theorem 5.1 (Size Interval). *Let T_i be a subtree of document T, Q be the query tree, $nml(T_i)$ be the number of missing labels in T_i w.r.t. Q, and $\mathcal{B} \geq 0$ an edit distance bound. If $\delta(Q, T_i) \leq \mathcal{B}$, then $|T_i|$ is in the* size interval

$$si(\mathcal{B}, Q, T_i) = [|Q| - \mathcal{B}; |Q| + \mathcal{B} - nml(T_i)] \qquad (3)$$

Proof. Recall that the number of missing labels in T_i w.r.t. Q is defined as $nml(T_i) = |\mathcal{L}(Q) \setminus \mathcal{L}(T_i)|$; $nml(T_i) \leq \mathcal{B}$ due to $\delta(Q, T_i) \leq \mathcal{B}$. We prove the correctness of the size interval by contradiction.

Case A: Assume a subtree T_i with $\delta(Q, T_i) \leq \mathcal{B}$ and $|T_i| \leq |Q| - \mathcal{B} - 1$. The minimum number of edit operations that transform any instance of T_i to some instance of Q consists of $|Q| - |T_i|$ insert operations. Note that we can decrease $nml(T_i)$ *and* the size difference $|Q| - |T_i|$ by inserting a new node with a label from $\mathcal{L}(Q) \setminus \mathcal{L}(T_i)$ into T_i. Thus, in the best case, we perform exactly $|Q| - |T_i|$ insertions, i.e., $\delta(Q, T_i) = |Q| - |T_i|$. Our assumption yields $|Q| - |T_i| \geq \mathcal{B} + 1$, hence $\delta(Q, T_i) \geq \mathcal{B} + 1$, which contradicts our assumption.

Case B: Assume a subtree T_i with $\delta(Q, T_i) \leq \mathcal{B}$ and $|T_i| \geq |Q| + \mathcal{B} - nml(T_i) + 1$. In this case, a delete operation can decrease the size difference $|T_i| - |Q|$ but cannot decrease $nml(T_i)$: to align the labels, we additionally need $nml(T_i)$ rename operations. Hence, the minimum number of edit operations that transform any instance of T_i to some instance of Q consists of (1) $nml(T_i)$ rename and (2) $|T_i| - |Q|$ delete operations, i.e., $\delta(Q, T_i) = nml(T_i) + |T_i| - |Q|$. Our assumption implies that $|T_i| - |Q| \geq \mathcal{B} - nml(T_i) + 1$. Therefore, $\delta(Q, T_i) \geq nml(T_i) + \mathcal{B} - nml(T_i) + 1 = \mathcal{B} + 1$, which contradicts our assumption.

Since the edit distance is symmetric, we do not need to consider the transformations of Q into T_i. □

Theorem 5.2. *Cone solves the top-k subtree similarity problem and verifies subtrees in candidate score order.*

Proof. *Correctness*: The stopping condition, $|R| = k \wedge \mathcal{B} \geq \delta(R[k])$, holds since all subtrees T_i in unprocessed partitions have size $|T_i| \notin si(\mathcal{B}, Q, T_i)$ and therefore $\delta(Q, T_i) > \mathcal{B}$ (cf. Theorem 5.1). Hence, these subtrees do not improve the ranking. If Cone does not produce enough candidates from lists that share a label with Q, we fall back to MergeAll on all remaining lists to derive a correct ranking. *Candidate score order*: We increment the edit bound $\mathcal{B}$, starting with $\mathcal{B} = 0$. For a given list x (starting with 0), we process all unprocessed partitions that contain subtrees in the size range $si(\mathcal{B}, Q, T_i)$. Let $P_{\mathcal{B}}$ denote the set of all subtrees T_i in these new partitions of the lists $x \leq \mathcal{B}$ that we have not seen before.

Similar to MergeAll, we perform two steps for a given $\mathcal{B}$: (1) We postpone the verification of subtrees $T_i \in P_{\mathcal{B}}$ for which $x = llb(Q, T_i) > \mathcal{B}$ and cache them in $lbc[x]$. (2) We verify (a) all subtrees $T_i \in P_{\mathcal{B}}$ for which $llb(Q, T_i) = \mathcal{B}$ and (b) all subtrees $T_i \in lbc[\mathcal{B}]$ (cached from previous sets $P_{\mathcal{B}'}$, $\mathcal{B}' < \mathcal{B}$). Hence, all subtrees T_i of the sets $P_{\mathcal{B}'}$, $\mathcal{B}' < \mathcal{B}$, with $llb(Q, T_i) = \mathcal{B}$ are verified when we process set $P_{\mathcal{B}}$. There is no subtree T_i with $llb(Q, T_i) \leq \mathcal{B}$ in some set $P_{\mathcal{B}''}$, $\mathcal{B}'' > \mathcal{B}$, i.e. $llb(Q, T_i) \leq \mathcal{B} \implies |T_i| \in si(\mathcal{B}, Q, T_i)$. Analogous to the proof of Theorem 5.1, we show this by contradiction. Recall that $llb(Q, T_i) = max\{|Q|, |T_i|\} - |\mathcal{L}(Q) \Cap \mathcal{L}(T_i)|$.

Case A: Assume a subtree T_i with $llb(Q, T_i) \leq \mathcal{B}$ and $|T_i| \leq |Q| - \mathcal{B} - 1$. Then, $max\{|Q|, |T_i|\} = |Q|$ implies that $llb(Q, T_i) = |Q| - |\mathcal{L}(Q) \Cap \mathcal{L}(T_i)|$. Our assumption yields $|\mathcal{L}(T_i)| \leq |Q| - \mathcal{B} - 1$, and $|\mathcal{L}(Q)| = |Q|$. Hence, $|\mathcal{L}(Q) \Cap \mathcal{L}(T_i)| \leq |Q| - \mathcal{B} - 1$ and therefore $llb(Q, T_i) \geq \mathcal{B} + 1$, which contradicts our assumption.

Case B: Assume a subtree T_i with $llb(Q, T_i) \leq \mathcal{B}$ and $|T_i| \geq |Q| + \mathcal{B} - nml(T_i) + 1$. Since $nml(T_i) \leq \mathcal{B}$, $max\{|Q|, |T_i|\} = |T_i| \implies llb(Q, T_i) = |T_i| - |\mathcal{L}(Q) \Cap \mathcal{L}(T_i)|$. Since $nml(T_i)$ labels of Q are not in T_i, $|\mathcal{L}(Q) \Cap \mathcal{L}(T_i)| = |Q| - nml(T_i)$ and therefore $llb(Q, T_i) = |T_i| - (|Q| - nml(T_i)) = |T_i| - |Q| + nml(T_i)$. Our assumption yields $|T_i| - |Q| \geq \mathcal{B} - nml(T_i) + 1$, hence $llb(Q, T_i) \geq \mathcal{B} + 1$, which contradicts our assumption.

In the fallback case, MergeAll guarantees score order. □

Theorem 6.1. *SlimCone solves the top-k subtree similarity problem and verifies subtrees in candidate score order.*

Proof. *Correctness* and *candidate score order* follow directly from Theorem 5.2 if we prove that SlimCone's partition traversal is identical to the partition traversal of Cone. Note that an identical partition traversal is sufficient, i.e., subtrees within the partitions need not be traversed in the same order. *Identical partition traversal*: SlimCone's list ordering is identical to the list ordering of Cone (cf. Section 6), hence lists (i.e., labels) are processed in the same order. We distinguish (1) uninitialized and (2) initialized lists: (1) Uninitialized lists: For each list entry i (rooting a subtree T_i) s.t. $|T_i| < |Q|$, the path in T is traversed upwards until $i' \neq i$ and $|T_{i'}| \geq |Q|$ holds. All traversed nodes (excl. i') are cached in the path cache pc. For $T_{i'}$, there are two cases: (a) $|T_{i'}| > |Q|$: i' is stored in the path ends pe. (b) $|T_{i'}| = |Q|$: $llb(Q, T_{i'})$ is computed. If $llb(Q, T_{i'}) = \mathcal{B}$, $T_{i'}$ is verified. Otherwise, we postpone the verification of $T_{i'}$ to round $\mathcal{B}'$. (2) Initialized lists: For a given list we process (a) all list entries i s.t. $|Q| + \mathcal{B} - nml(T_i) \geq |T_i| > |Q|$, (b) all entries in the path cache pc, and (c) all entries in the path ends pe. Due to (b) we process all subtrees T_i smaller than Q, $|Q| > T_i \geq |Q| - \mathcal{B}$; due to (c) we process all subtrees T_i larger than Q, $|Q| + \mathcal{B} - nml(T_i) \geq T_i > |Q|$.

(2) and (3) guarantee that (i) SlimCone's partition traversal is identical to the partition traversal of Cone *and* (ii) *all* subtrees of a partition are generated. □

A.2 Pseudo Codes

We provide the pseudo code for all our algorithms: MergeAll (Algorithm 1), Cone (Algorithm 4), and SlimCone (Algorithm 6). The other algorithms presented in this section are auxiliary functions used in the main algorithms.

Algorithm 1: *MergeAll* (Q, T, k)

```
Input: query tree $Q$, document tree $T$, result size $k$
Result: top-$k$ ranking $R$ of subtrees of $T$ w.r.t. $Q$
// $\mathcal{I}$ ... candidate index, $\mathcal{L}(Q)$ ... label multiset of $Q$
// $T_i$ ... subtree rooted at node $i$
1  foreach $\lambda \in \mathcal{L}(Q)$ do // initialize inverted lists
2      $l_\lambda \leftarrow \mathcal{I}.list(\lambda)$; // retrieve list $l_\lambda$
3      if $l_\lambda \neq nil$ then // initialize pointers $l_\lambda.r, l_\lambda.l$
4          $l_\lambda.r \leftarrow$ pos. of $i$ s.t. $||Q| - |T_i||$ is minimal;
5          $l_\lambda.l \leftarrow l_\lambda.r - 1$
6  $ol \leftarrow$ empty associative array; // overlap store
7  $lbc \leftarrow$ empty dynamic array; // lower bound cache
8  $j \leftarrow 0$; // current stripe number
9  $R \leftarrow$ empty ranking;
10 while $j \leq 2|Q|$ do // $j > 2|Q|$: we must consider all lists
11     if $verifyBucket(j)$ then return $R$; // evaluate $lbc[j]$
12     foreach node $i \in S_j \cup S_{-j}$ do // compute overlaps
13         $ol[i] \leftarrow \#$ of lists $l_\lambda$ s.t. $i \in l_\lambda$;
14         advance $l_\lambda.r$ and $l_\lambda.l$;
15     foreach key $i \in ol$ do // process subtrees (cache or verify)
16         $lb \leftarrow max\{|Q|, |T_i|\} - ol[i]$;
17         if $processSubtree(T_i, lb, j)$ then return $R$;
18     $j \leftarrow j + 1$; // proceed to next $j' > j$
       // check if we can terminate before continuing
19     if $|R| = k \wedge j \geq \delta(R[k])$ then return $R$;
20 return $R$;
```

Algorithm 2: *processSubtree* $(T_i, lb, \mathcal{B})$

```
Input: subtree $T_i$, lower bound $lb$, edit distance bound $\mathcal{B} \leq lb$
Result: true if final ranking found, false otherwise
// $lbc, R, Q$ globally accessible
1  if $lb > \mathcal{B}$ then // cache $T_i$
2      $lbc[lb] \leftarrow lbc[lb] \cup \{T_i\}$;
3      return $false$; // we cannot terminate
4  compute $\delta(T_i, Q)$ and update $R$ with $T_i$; // $lb = \mathcal{B}$; verify $T_i$
5  return $|R| = k \wedge \mathcal{B} \geq \delta(R[k])$; // indicates if we can terminate
```

Algorithm 3: *verifyBucket* $(\mathcal{B})$

```
Input: edit distance bound $\mathcal{B}$
Result: true if final ranking found, false otherwise
// $lbc, R, Q$ globally accessible
1  foreach $T_i \in lbc[\mathcal{B}]$ do // verify all subtrees in $lbc[\mathcal{B}]$
2      compute $\delta(T_i, Q)$ and update $R$ with $T_i$;
       // return as soon as we can terminate
3      if $|R| = k \wedge \mathcal{B} \geq \delta(R[k])$ then return $true$;
4  return $false$; // we cannot terminate
```

Algorithm 4: *Cone* (Q, T, k)

```
Input: query tree $Q$, document tree $T$, result size $k$
Result: top-$k$ ranking $R$ of subtrees of $T$ w.r.t. $Q$
1  $L \leftarrow$ deduplicated $\mathcal{L}(Q)$;
2  sort $L$ by increasing list length $|l_\lambda|, \lambda \in L$;
3  $lbc \leftarrow$ empty dynamic array; // lower bound cache
4  $\mathcal{B} \leftarrow 0$; // current edit distance bound
5  $R \leftarrow$ empty ranking;
6  while $\mathcal{B} \leq 2|Q|$ do // $\mathcal{B} > 2|Q|$: use MergeAll on all lists
7      if $verifyBucket(\mathcal{B})$ then return $R$; // evaluate $lbc[\mathcal{B}]$
8      foreach $init.$ $list$ $l_\lambda$ do // process initialized lists first
9          if $processList(l_\lambda, \mathcal{B})$ then return $R$;
10     if $\mathcal{B} \leq |L|$ then // initialize next list
11         $l_\lambda \leftarrow \mathcal{I}.list(L[\mathcal{B}])$; // retrieve next list
           // process list $l_\lambda$; $T_i$ ... subtree rooted at node $i$
12         if $l_\lambda \neq nil$ then // initialize pointers $l_\lambda.r, l_\lambda.l$
13             $l_\lambda.r \leftarrow$ pos. of $i$ s.t. $||Q| - |T_i||$ is minimal;
14             $l_\lambda.l \leftarrow l_\lambda.r - 1$;
15             if $processList(l_\lambda, \mathcal{B})$ then return $R$;
16     $\mathcal{B} \leftarrow \mathcal{B} + 1$; // proceed to next $\mathcal{B}' > \mathcal{B}$
       // check if we can terminate before continuing
17     if $|R| = k \wedge \mathcal{B} \geq \delta(R[k])$ then return $R$;
18 return $R$;
```

Algorithm 5: *processList* $(l_\lambda, \mathcal{B})$

```
Input: inverted list $l_\lambda$, edit distance bound $\mathcal{B}$
Result: true if final ranking found, false otherwise
// $Q$ globally accessible; $T_i$ ... subtree rooted at node $i$
1  $minsize \leftarrow |Q| - \mathcal{B}$; // min. subtree size to consider
2  $maxsize \leftarrow |Q| + \mathcal{B} - idx[l_\lambda]$; // max. subtree size to consider
3  foreach unseen node $i \in l_\lambda$ s.t. $maxsize \geq |T_i| \geq minsize$ do
       // process $T_i$ and return as soon as we can terminate
4      if $processSubtree(T_i, llb(T_i, Q), \mathcal{B})$ then return true;
5      advance $l_\lambda.r$ and $l_\lambda.l$;
6  return false; // we cannot terminate
```

Algorithm 6: *SlimCone* (Q, T, k)

```
Input: query tree $Q$, document tree $T$, result size $k$
Result: top-$k$ ranking $R$ of subtrees of $T$ w.r.t. $Q$
1  $L \leftarrow$ deduplicated $\mathcal{L}(Q)$;
2  sort $L$ by increasing full list length $|l_\lambda|$, $\lambda \in L$;
   // lower bound cache $lbc$, path cache $pc$, path ends $pe$, and
   // positions in slim lists $next$
3  $lbc, pc, pe, next \leftarrow$ empty dynamic arrays;
4  $\mathcal{B} \leftarrow 0$; // current edit distance bound
5  $R \leftarrow$ empty ranking;
6  while $\mathcal{B} \leq 2|Q|$ do // $\mathcal{B} > 2|Q|$: use MergeAll on all lists
7      if $verifyBucket(\mathcal{B})$ then return $R$; // evaluate $lbc[\mathcal{B}]$
8      foreach init. list $l_\lambda$ do // process initialized lists first
9          if $processSmaller(l_\lambda, \mathcal{B})$ then return $R$;
10         if $processLarger(l_\lambda, \mathcal{B})$ then return $R$;
11     if $\mathcal{B} \leq |L|$ then // initialize next list
12         $l_\lambda \leftarrow \mathcal{I}.list(L[\mathcal{B}])$; // retrieve next list
13         if $l_\lambda \neq nil$ then
               // initialize $l_\lambda$ buckets in $pc$ and $pe$
14             $pc[l_\lambda], pe[l_\lambda] \leftarrow$ empty dynamic Arrays;
15             $next[l_\lambda] \leftarrow 0$; // initialize $next$ pointer for $l_\lambda$
16             foreach unseen node $i \in l_\lambda$ s.t. $|T_i| < |Q|$ do
                   // climb up path; $T_q$ ... subtree rooted at $q$
17                 traverse up to first node $q$ s.t. $|T_q| \geq |Q|$;
                   // add all traversed nodes to path cache
18                 foreach traversed node $x$ (excl. $q$) do
19                     $pc[l_\lambda][|T_x|] \leftarrow pc[l_\lambda][|T_x|] \cup \{x\}$;
                   // process $T_q$ if size fits $|Q|$
20                 if $q$ unseen $\wedge |T_q| = |Q|$ then
21                     $lb \leftarrow llb(T_q, Q)$;
22                     if $processSubtree(T_q, lb, \mathcal{B})$ then
23                         return $R$;
24                     $q \leftarrow par(q)$; // traverse to parent of $q$
25                 $pe[l_\lambda] \leftarrow pe[l_\lambda] \cup \{q\}$; // add $q$ to path ends
26                 $next[l_\lambda] \leftarrow$ pos. of $i$ in $l_\lambda$; // next $l_\lambda$-entry
27             if $processSmaller(l_\lambda, \mathcal{B})$ then return $R$;
28             if $processLarger(l_\lambda, \mathcal{B})$ then return $R$;
29     $\mathcal{B} \leftarrow \mathcal{B} + 1$; // proceed to next $\mathcal{B}' > \mathcal{B}$
       // check if we can terminate before continuing
30     if $|R| = k \wedge \mathcal{B} \geq \delta(R[k])$ then return $R$;
31 return $R$
```

Algorithm 7: *processSmaller* $(l_\lambda, \mathcal{B})$

```
Input: inverted list $l_\lambda$, edit distance bound $\mathcal{B}$
Result: true if final ranking found, false otherwise
// $pc$, $Q$ globally accessible
1  $minsize \leftarrow |Q| - \mathcal{B}$; // min. subtree size to consider
   // process all path cache buckets that fit w.r.t. size
2  $s \leftarrow |Q| - 1$;
3  while $s \geq minsize$ do
4      $b \leftarrow pc[l_\lambda][s]$; // get path cache bucket
       // process all subtrees; $T_q$ ... subtree rooted at node $q$
5      foreach unseen node $q \in b$ do
           // process $T_q$ and return as soon as we can terminate
6          if $processSubtree(T_q, llb(T_q, Q), \mathcal{B})$ then
7              return true;
8      $s \leftarrow s - 1$; // proceed to next subtree size
9  return false; // we cannot terminate
```

Algorithm 8: *processLarger* $(l_\lambda, \mathcal{B})$

```
Input: inverted list $l_\lambda$, edit distance bound $\mathcal{B}$
Result: true if final ranking found, false otherwise
// $next$, $pe$, $Q$ globally accessible
1  $maxsize \leftarrow |Q| + \mathcal{B} - idx[l_\lambda]$; // max. subtree size to consider
   // add fitting list entries to path ends bucket
2  while $T_n \leftarrow$ subtree rooted at $next[l_\lambda] \wedge |T_n| \leq maxsize$ do
3      $pe[l_\lambda] \leftarrow pe[l_\lambda] \cup \{next[l_\lambda]\}$;
4      advance $next[l_\lambda]$;
5  foreach unseen node $q \in pe[l_\lambda]$ do // process path ends bucket
6      if $|T_q| \leq maxsize$ then // $T_q$ ... subtree rooted at node $q$
           // process $T_q$ and return as soon as we can terminate
7          if $processSubtree(T_q, llb(T_q, Q), \mathcal{B})$ then
8              return true;
9          $q \leftarrow par(q)$; // traverse to parent
10 return false; // we cannot terminate
```