Efficient Top-k Approximate Subtree Matching in Small Memory

Nikolaus Augsten, Denilson Barbosa, Michael Böhlen, Themis Palpanas

Abstract—We consider the *Top-k Approximate Subtree Matching* (TASM) problem: finding the k best matches of a small query tree within a large document tree using the canonical tree edit distance as a similarity measure between subtrees. Evaluating the tree edit distance for large XML trees is difficult: the best known algorithms have cubic runtime and quadratic space complexity, and, thus, do not scale. Our solution is TASM-postorder, a memory-efficient and scalable TASM algorithm. We prove an upper bound for the maximum subtree size for which the tree edit distance needs to be evaluated. The upper bound depends on the query and is independent of the document size and structure. A core problem is to efficiently prune subtrees that are above this size threshold. We develop an algorithm based on the prefix ring buffer that allows us to prune all subtrees above the threshold in a single postorder scan of the document. The size of the prefix ring buffer is linear in the threshold. As a result, the space complexity of TASM-postorder depends only on k and the query size, and the runtime of TASM-postorder is linear in the size of the document. Our experimental evaluation on large synthetic and real XML documents confirms our analytic results.

Index Terms—Approximate Subtree Matching, Tree Edit Distance, Top-k Queries, XML, Subtree Pruning, Similarity Search

1 Introduction

Repositories of XML documents have become popular and widespread. Along with this development has come the need for efficient techniques to *approximately match* XML trees based on their similarity according to a given distance metric. Approximate matching is used for integrating heterogeneous repositories [1], [2], [3], [4], cleaning such integrated data [5], as well as for answering *similarity queries* [6], [7]. In this paper we consider the *Top-k Approximate Subtree Matching* problem (TASM), i.e., the problem of ranking the k best approximate matches of a small query tree in a large document tree. More precisely, given two ordered labeled trees, a query Q of size m and a document T of size n, we want to produce a *ranking* $(T_{i_1}, T_{i_2}, \dots, T_{i_k})$ of k subtrees of T (consisting of nodes of T with their descendants) that are closest to Q with respect to a given metric. We use the canonical tree edit distance to determine the ranking [8], [9].

The naive solution to TASM computes the distance between the query Q and every subtree in the document T, thus requiring n distance computations. Using the well-established tree edit distance as a metric, the naive solution to TASM requires $O(m^2n^2)$ time and $O(mn)$ space. An $O(n)$ improvement in time leverages the dynamic programing formulation of tree edit distance algorithms: compute the distance between Q and T, and rank all subtrees of T by visiting the resulting memoization table. Still, for large documents with millions of nodes, the $O(mn)$ space complexity is prohibitive.

We develop and evaluate an efficient algorithm for TASM based on a prefix ring buffer that performs a single scan of the large document. The size of the prefix ring buffer is independent of the document size. Our contributions are:

- We prove an upper bound τ on the size of the subtrees that must be considered for solving TASM. This threshold is independent of document size and structure.
- We introduce the prefix ring buffer to prune subtrees larger than τ in $O(\tau)$ space, during a single postorder scan of the document.
- We develop TASM-postorder, an efficient and scalable algorithm for solving TASM. The space complexity is independent of the document size and the time complexity is linear in the document size.

The rest of this paper is organized as follows. Section 2 gives the problem definition and Section 3 discusses related work. Section 4 revisits the tree edit distance and discusses the state-of-the-art in TASM. Section 5 introduces the prefix ring buffer and discusses our pruning strategy, which is the basis of our solution for TASM, given in Section 6 and thoroughly evaluated in Section 7. We conclude in Section 8.

2 Problem Definition

Definition 1: (Top-k Approximate Subtree Matching Problem). Let Q (query) and T

- *N. Augsten is with the Faculty of Computer Science, Free University of Bozen-Bolzano, Italy. E-mail: augsten@inf.unibz.it*
- *D. Barbosa is with the Department of Computing Science, University of Alberta, Canada. E-mail: denilson@cs.ualberta.ca*
- *M. Böhlen is with the Department of Informatics, University of Zurich, Switzerland. E-mail: boehlen@ifi.uzh.ch*
- *T. Palpanas is with the Department of Information Engineering and Computer Science, University of Trento, Italy. E-mail: themis@disi.unitn.eu*

(document) be ordered labeled trees, n be the number of nodes of T, T_i be the subtree of T that is rooted at node t_i and includes all its descendants, $d(.,.)$ be a distance function between ordered labeled trees, and $k \leq n$ be an integer. A sequence of subtrees, $R = (T_{i_1}, T_{i_2}, \ldots, T_{i_k})$, is a top-$k$ ranking of the subtrees of the document T with respect to the query Q iff

1) the ranking contains the k subtrees that are closest to the query: $\forall T_j \notin R : d(Q, T_{i_k}) \leq d(Q, T_j)$, and
2) the subtrees in the ranking are sorted by their distance to the query: $\forall 1 \leq j < k : d(Q, T_{i_j}) \leq d(Q, T_{i_{j+1}})$.

Top-k approximate subtree matching (TASM) is the problem of computing a top-k ranking of the subtrees of a document T with respect to a query Q.

3 Related Work

Answering top-k *queries* is an active research field [10]. Specific to XML, many authors have studied the ranking of answers to twig queries [11], [12], [13], which are XPath expressions with branches specifying predicates on nodes (e.g., restrictions on their tag names or content) and structural relationships between nodes (e.g., ancestor-descendant). Answers (resp., approximate answers) to a twig query are subtrees of the document that satisfy (resp., partially satisfy) the conditions in the query. Answers are ranked according to the restrictions in the query that they violate. Approximate answers are found by explicitly relaxing the restrictions in the query through a set of predefined rules. Relevant subtrees that are similar to the query but do not fit any rule will not be returned by these methods. The main differences among the methods above are in the relaxation rules and the scoring functions they use. In contrast, we do not restrict the set of possible answers by predefined rules. All subtrees of the document are potentially considered as an answer. Further, we do not define a new scoring function for the structural similarity, but we use the established tree edit distance [8], [9].

The goal of XML *keyword search* [7], [14], [15] is to find the top-k subtrees of a document given a set of keywords. Answers are subtrees that contain at least one such keyword. Because two keywords may appear in different branches of the XML tree (and thus be far from each other in terms of structure), candidate answers are ranked based on a *content score* (indicating how well a subtree covers the keywords) and a *structural score* (indicating how concise a subtree is). These are combined into a single ranking. Kaushik et al. [16] study TA-style [17] algorithms to combine content and structural scores. TASM differs from keyword search: instead of keywords, queries are entire trees; instead of using text similarity, subtrees are ranked based on the well-understood tree edit distance.

XFinder [6] ranks the top-k approximate matches of a small query tree in a large document tree. Both the query and the document are transformed to strings using Prüfer sequences, and the tree edit distance is approximated by the longest subsequence distance between the resulting strings. The edit model used to compute distances in XFinder does not handle renaming operations. Also, in [6] no runtime analysis is given and the experiments reported use documents of up to 5MB. In contrast, we provide and validate tight analytical bounds, solve the problem with the unrestricted tree edit distance and efficiently apply our solution to documents of 1.6GB.

We use the tree edit distance [8] to compute the similarity between the query and the subtrees of the document. For ordered trees like XML this problem can be solved with elegant dynamic programming formulations. Zhang and Shasha [9] present an $O(n^2 \log^2 n)$ time and $O(n^2)$ space algorithm for trees with n nodes and height $O(\log n)$. Their worst case complexity is $O(n^4)$. Demaine et al. [18] use a different tree decomposition strategy to improved the time complexity to $O(n^3)$ in the worst case. This is not a concern in practice since XML documents tend to be shallow and wide [19]. This is also true for the real documents in our tests: the DBLP bibliography[1] (26M nodes, 476MB, height 6), and the PSD7003 protein dataset[2] (37M nodes, 683MB, height 7). Thus we use the classical algorithm of Zhang and Shasha [9].

Guha et al. [1] match pairs of XML trees from heterogeneous repositories whose tree edit distance falls within a threshold. They give upper and lower bounds for the tree edit distance that can be computed in $O(n^2)$ time as a pruning strategy to avoid comparing all pairs of trees from the repositories. Yang et al. [20] and Augsten et al. [21] provide lower bounds for the tree edit distance that can be computed in $O(n \log n)$ time. In contrast, we compute (once for each query) an upper bound on the *size* of the subtrees that may be in the answer, i.e., among the top-k.

Approximate substructure matching has also been studied in the context of graphs [22], [23]. TALE [23] is a tool that supports approximate graph queries against large graph databases. TALE is based on an indexing method that scales linearly to the number of nodes of the graph database. Unlike our work, TALE uses heuristic techniques and does not guarantee that the final answer will include the best matches or that all possible matches will be considered.

We define the postorder queue to abstract from the underlying XML storage model. The postorder queue uses the postorder position and the subtree size of a node to uniquely define the XML structure. The interval encoding [24], which stores XML in relations, is based on similar ideas.

[1] http://dblp.uni-trier.de/xml

[2] http://www.cs.washington.edu/research/xmldatasets

4 Preliminaries and Background

The tree edit distance has emerged as the standard measure to capture the similarity between ordered labeled trees. Given a cost model, it sums up the cost of the least costly sequence of edit operations that transforms one tree into the other.

4.1 Trees

A *tree* T is a directed, acyclic, connected graph with nodes $V(T)$ and edges $E(T)$, where each node has at most one incoming edge. A *node*, $t_i \in V(T)$, is an (identifier, label) pair. The identifier is unique within the tree. The *label*, $\lambda(t_i) \in \Sigma$, is a symbol of a finite alphabet Σ. The *empty node* ϵ does not appear in a tree. $V_\epsilon(T) = V(T) \cup \{\epsilon\}$ denotes the set of all nodes of T extended with the empty node ϵ. By $|T| = |V(T)|$ we denote the size of T. An *edge* is an ordered pair (t_p, t_c), where $t_p, t_c \in V(T)$ are nodes, and t_p is the *parent* of t_c. Nodes with the same parent are *siblings*.

The nodes of a tree are strictly and totally ordered. Node t_c is the i-th *child* of t_p iff t_p is the parent of t_c and $i = |\{t_x \in V(T) : (t_p, t_x) \in E(T), t_x \leq t_c\}|$. Any child node t_c *precedes* its parent node t_p in the node order, written $t_c < t_p$. The tree traversal that visits all nodes in ascending order is the *postorder* traversal.

The number of t_p's children is its *fanout* f_{t_p}. The node with no parent is the *root* node, $\mathrm{root}(T)$, and a node without children is a *leaf*. An *ancestor* of t_i is a node t_a in the path from the root node to t_i, $t_a \neq t_i$. With $\mathrm{anc}(t_d)$ we denote the set of all ancestors of a node t_d. Node t_d is a *descendant* of t_i iff $t_i \in \mathrm{anc}(t_d)$. A node t_i is *to the left* of a node t_j iff $t_i < t_j$ and t_i is not a descendant of t_j.

T_i is the *subtree rooted in node* t_i of T iff $V(T_i) = \{t_x \mid t_x = t_i \text{ or } t_x \text{ is a descendant of } t_i \text{ in } T\}$ and $E(T_i) \subseteq E(T)$ is the projection of $E(T)$ w.r.t. $V(T_i)$, thus retaining the original node ordering. By $lml(t_i)$ we denote the *leftmost leaf* of T_i, i.e., the smallest descendant of node t_i. A *subforest* of a tree T is a graph with nodes $V' \subseteq V(T)$ and edges $E' = \{(t_i, t_j) \mid (t_i, t_j) \in E(T), t_i \in V', t_j \in V'\}$.

4.2 Postorder Queues

A postorder queue is a sequence of $(label, size)$ pairs of the tree nodes in postorder, where $label$ is the node label and $size$ is the size of the subtree rooted in the respective node. A postorder queue uniquely defines an ordered labeled tree. The only operation allowed on a postorder queue is dequeue, which removes and returns the first element of the sequence.

Definition 2 (Postorder Queue): Given a tree T with $n = |T|$ nodes, the *postorder queue*, *post*(T), of T is a sequence of pairs $((l_1, s_1), (l_2, s_2), \ldots, (l_n, s_n))$, where $l_i = \lambda(t_i)$, $s_i = |T_i|$, with t_i being the i-th node of T in postorder. The dequeue operation on a postorder queue $p = (p_1, p_2, \ldots, p_n)$ is defined as $\mathrm{dequeue}(p) = ((p_2, p_3, \ldots, p_n), p_1)$.

4.3 Edit Operations and Edit Mapping

An edit operation transforms a tree Q into a tree T. We use the standard edit operations on trees [8], [9]: *delete* a node and connect its children to its parent maintaining the sibling order; *insert* a new node between an existing node, t_p, and a subsequence of consecutive children of t_p; and *rename* the label of a node. We define the edit operations in terms of edit mappings [8], [9].

Definition 3: (Edit Mapping and Node Alignment). Let Q and T be ordered labeled trees. $M \subseteq V_\epsilon(Q) \times V_\epsilon(T)$ is an *edit mapping between* Q *and* T iff

1) every node is mapped:
   a) $\forall q_i (q_i \in V(Q) \Leftrightarrow \exists t_j ((q_i, t_j) \in M))$
   b) $\forall t_i (t_i \in V(T) \Leftrightarrow \exists q_j ((q_j, t_i) \in M))$
   c) $(\epsilon, \epsilon) \notin M$
2) all pairs of non-empty nodes $(q_i, t_j), (q_k, t_l) \in M$ satisfy the following conditions:
   a) $q_i = q_k \Leftrightarrow t_j = t_l$ (one-to-one condition)
   b) q_i is an ancestor of $q_k \Leftrightarrow t_j$ is an ancestor of t_l (ancestor condition)
   c) q_i is to the left of $q_k \Leftrightarrow t_j$ is to the left of t_l (order condition)

A pair $(q_i, t_j) \in M$ is a *node alignment.*

Non-empty nodes that are mapped to other non-empty nodes are either renamed or not modified when Q is transformed into T. Nodes of Q that are mapped to the empty node are deleted from Q, and nodes of T that are mapped to the empty node are inserted into T.

4.4 Tree Edit Distance

In order to determine the distance between trees a cost model must be defined. We assign a cost to each node alignment of an edit mapping. This cost is proportional to the costs of the nodes.

Definition 4 (Cost of Node Alignment): Let Q and T be ordered labeled trees, let $cst(x) \geq 1$ be a cost assigned to a node x, $q_i \in V_\epsilon(Q)$, $t_j \in V_\epsilon(T)$. The *cost of a node alignment*, $\gamma(q_i, t_j)$, is defined as:

$$\gamma(q_i, t_j) = \begin{cases} cst(q_i) \quad \text{if } q_i \neq \epsilon \wedge t_j = \epsilon & \text{(delete)} \\ cst(t_j) \quad \text{if } q_i = \epsilon \wedge t_j \neq \epsilon & \text{(insert)} \\ (cst(q_i) + cst(t_j))/2 & \text{(rename)} \\ \qquad \text{if } q_i \neq \epsilon \wedge t_j \neq \epsilon \wedge \lambda(q_i) \neq \lambda(t_j) & \\ 0 & \text{(no change)} \\ \qquad \text{if } q_i \neq \epsilon \wedge t_j \neq \epsilon \wedge \lambda(q_i) = \lambda(t_j) & \end{cases}$$

Definition 5 (Cost of Edit Mapping): Let Q and T be two ordered labeled trees, $M \subseteq V_\epsilon(Q) \times V_\epsilon(T)$ be an edit mapping between Q and T, and $\gamma(q_i, t_j)$ be the cost of a node alignment. The *cost of the edit mapping* M is defined as the sum of the costs of all node alignments in the mapping:

$$\gamma^*(M) = \sum_{(q_i, t_j) \in M} \gamma(q_i, t_j)$$

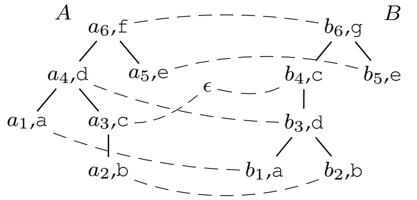


Fig. 1. Edit Mapping between Trees A and B.

The tree edit distance between two trees Q and T is the cost of the least costly edit mapping [9].

Definition 6 (Tree Edit Distance): Let Q and T be two ordered labeled trees. The *tree edit distance*, $\delta(Q,T)$, between Q and T is the cost of the least costly edit mapping, $M \subseteq V_\epsilon(Q) \times V_\epsilon(T)$, between the two trees:

$$\delta(Q,T) = \min\{\gamma^*(M) \mid \\ M \subseteq V_\epsilon(Q) \times V_\epsilon(T) \text{ is an edit mapping}\}$$

In the unit cost model all nodes have cost 1, and the unit cost tree edit distance [9] is the minimum number of edit operations that transform one tree into the other. Other cost models can be used to tune the tree edit distance to specific application needs, for example, the fanout weighted tree edit distance [21] makes edit operations that change the structure (insertions and deletions of non-leaf nodes) more expensive; in XML, the node cost can depend on the element type.

Example 1: Figure 1 illustrates an edit mapping $M = \{(a_1,b_1), (a_2,b_2), (a_3,\epsilon), (a_4,b_3), (\epsilon,b_4), (a_5,b_5), (a_6,b_6)\}$ between trees A and B. If the cost of all nodes of A and B is 1, $\gamma(a_6,b_6) = \gamma(a_3,\epsilon) = \gamma(\epsilon,b_4) = 1$; the cost of all other node alignments is zero. M is the least costly edit mapping between A and B, thus the tree edit distance is $\delta(A,B) = \gamma^*(M) = 3$ (node a_6 is renamed, a_3 is deleted, b_4 is inserted).

4.5 Computing the Tree Edit Distance

The fastest algorithms for the tree edit distance use dynamic programming. In this section we discuss the classic algorithm by Zhang and Shasha [9], which recursively decomposes the input trees into smaller units and computes the tree distance bottom-up. The decompositions do not always result in trees, but may also produce forests; in fact, the decomposition rules of Zhang and Shasha [9] assume forests. A forest is recursively decomposed by deleting the root node of the rightmost tree in the forest, deleting the rightmost tree of the forest, or keeping only the rightmost tree of the forest. Figure 4 illustrates the decomposition of the example document H in Figure 2.

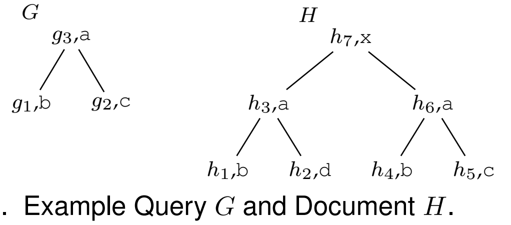


Fig. 2. Example Query G and Document H.

The decomposition of a tree results in the set of all its subtrees and all the prefixes of these subtrees. A prefix is a subforest that consists of the first i nodes of a tree in postorder.

Definition 7 (Prefix): Let T be an ordered labeled tree, and t_i be the i-th node of T in postorder. The *prefix* $\text{pfx}(T,t_i)$ of T, $1 \leq i \leq |T|$, is a forest with nodes $V' = \{t_1, t_2, \ldots, t_i\}$ and edges $E' = \{(t_k,t_l) \mid (t_k,t_l) \in E(T), t_k \in V', t_l \in V'\}$.

A tree with n nodes has n prefixes. The first line in Figure 4 shows all prefixes of example document H.

The tree edit distance algorithm computes the distance between all pairs of subtree prefixes of two trees. Some subtrees can be expressed as a prefix of a larger subtree, for example $H_3 = \text{pfx}(H_7,h_3)$ in Figure 4. All prefixes of the smaller subtree (e.g., H_3) are also prefixes of the larger subtree (e.g., H_7) and should not be considered twice in the tree edit distance computation. The *relevant subtrees* are those subtrees that cannot be expressed as prefixes of other subtrees. All prefixes of relevant subtrees must be computed.

Definition 8 (Relevant Subtree): Let T be an ordered labeled tree and let $t_i \in V(T)$. Subtree T_i is *relevant* iff it is not a prefix of any other subtree: T_i is relevant $\Leftrightarrow t_i \in V(T) \wedge \forall t_k, t_l (t_k \in V(T), t_k \neq t_i, t_l \in V(T_k) \Rightarrow T_i \neq \text{pfx}(T_k,t_l))$.

Example 2: Consider the example trees in Figure 2. The relevant subtrees of G are G_2 and G_3, the relevant subtrees of H are H_2, H_5, H_6, and H_7.

The decomposition rules for the tree edit distance are given in Figure 3; they decompose the prefixes of two (sub)trees Q_m and T_n ($q_i \leq q_m$, $t_j \leq t_n$). Rule (e) decomposes two general prefixes, (d) decomposes two prefixes that are proper trees (rather than forests), (b) and (c) decompose one prefix when the other prefix is empty, and (a) terminates the recursion.

$$
\begin{aligned}
&(a)\ \delta(\emptyset,\emptyset) = 0\\
&(b)\ \delta(\text{pfx}(Q_m,q_i),\emptyset) = \delta(\text{pfx}(Q_m,q_{i-1}),\emptyset) + \gamma(q_i,\epsilon)\\
&(c)\ \delta(\emptyset,\text{pfx}(T_n,t_j)) = \delta(\emptyset,\text{pfx}(T_n,t_{j-1})) + \gamma(\epsilon,t_j)\\
&(d)\ \delta(\text{pfx}(Q_m,q_m),\text{pfx}(T_n,t_n)) = \min(\\
&\qquad \delta(\text{pfx}(Q_m,q_{m-1}),\text{pfx}(T_n,t_n)) + \gamma(q_m,\epsilon),\\
&\qquad \delta(\text{pfx}(Q_m,q_m),\text{pfx}(T_n,t_{n-1})) + \gamma(\epsilon,t_n),\\
&\qquad \delta(\text{pfx}(Q_m,q_{m-1}),\text{pfx}(T_n,t_{n-1})) + \gamma(q_m,t_n))\\
&(e)\ \delta(\text{pfx}(Q_m,q_i),\text{pfx}(T_n,t_j)) = \min(\\
&\qquad \delta(\text{pfx}(Q_m,q_{i-1}),\text{pfx}(T_n,t_j)) + \gamma(q_i,\epsilon),\\
&\qquad \delta(\text{pfx}(Q_m,q_i),\text{pfx}(T_n,t_{j-1})) + \gamma(\epsilon,t_j),\\
&\qquad \delta(\text{pfx}(Q_m,q_{i-|Q_i|}),\text{pfx}(T_n,t_{j-|T_j|}) + \delta(Q_i,T_j))
\end{aligned}
$$

Fig. 3. Decomposition Rules for the Tree Edit Distance.

4.6 TASM-dynamic

The dynamic programming algorithm for the tree edit distance fills the tree distance matrix td, and the last row of td stores the distances between the query and all subtrees of the document. This yields a simple

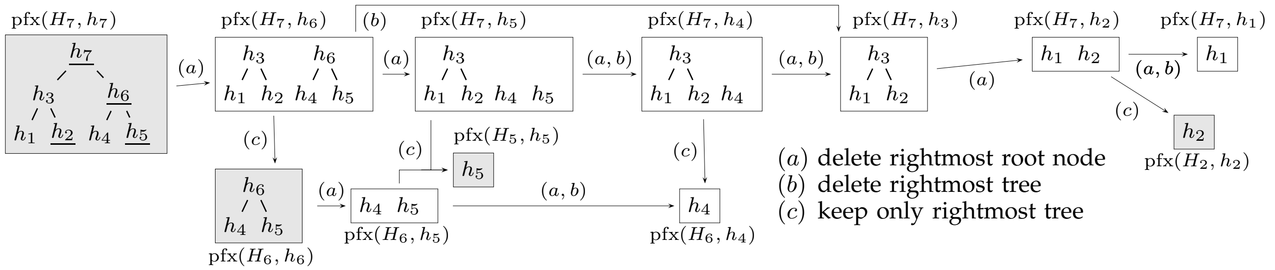


Fig. 4. Decomposing Example Document H into Prefixes.

Prefix Distance Matrix pd (only relevant parts are shown):

G_2, H_2

$t_j \rightarrow$		h_2
$q_i \downarrow$	0	1
g_2	1	1

G_2, H_5

$t_j \rightarrow$		h_5
$q_i \downarrow$	0	1
g_2	1	0

G_2, H_6

$t_j \rightarrow$		h_4	h_5	h_6
$q_i \downarrow$	0	1	2	3
g_2	1	1	1	2

G_2, H_7

$t_j \rightarrow$		h_1	h_2	h_3	h_4	h_5	h_6	h_7
$q_i \downarrow$	0	1	2	3	4	5	6	7
g_2	1	1	2	3	4	4	5	6

G_3, H_2

$t_j \rightarrow$		h_2
$q_i \downarrow$	0	1
g_1	1	1
g_2	2	2
g_3	3	3

G_3, H_5

$t_j \rightarrow$		h_5
$q_i \downarrow$	0	1
g_1	1	1
g_2	2	1
g_3	3	2

G_3, H_6

$t_j \rightarrow$		h_4	h_5	h_6
$q_i \downarrow$	0	1	2	3
g_1	1	0	1	2
g_2	2	1	0	1
g_3	3	2	1	0

G_3, H_7

$t_j \rightarrow$		h_1	h_2	h_3	h_4	h_5	h_6	h_7
$q_i \downarrow$	0	1	2	3	4	5	6	7
g_1	1	0	1	2	3	4	5	6
g_2	2	1	1	2	3	3	4	5
g_3	3	2	2	1	2	3	3	4

Tree Distance Matrix td:

	H_1	H_2	H_3	H_4	H_5	H_6	H_7
G_1	0	1	2	0	1	2	6
G_2	1	1	3	1	0	2	6
G_3	2	3	1	2	2	0	4

$R = (H_6, H_3)$

Fig. 5. Tree Edit Distance Example.

solution to TASM: compute the tree edit distance between the query and the document, sort the last row of matrix td, and add the k closest subtrees to the ranking. We refer to this algorithm as TASM-dynamic (Algorithm 1).

TASM-dynamic is a dynamic programming implementation of the decomposition rules in Figure 3. A matrix td stores the distances between all pairs of subtrees of Q and T. For each pair of relevant subtrees, Q_m and T_n, a temporary matrix pd is filled with the distances between all prefixes of Q_m and T_n. The distances between all prefixes that are proper subtrees (rather than forests) are saved in td. Note that the prefix $\text{pfx}(Q_m, q_i)$ is a proper subtree iff $\text{pfx}(Q_m, q_i) = Q_i$.

The ranking, R, is implemented as a max-heap that stores $(key, value)$ pairs: $\max(\mathsf{R})$ returns the maximum key of the heap in constant time; $\text{push-heap}(\mathsf{R}, (k, v))$ inserts a new element (k, v) in logarithmic time; and $\text{pop-heap}(\mathsf{R})$ deletes the element with the maximum key in logarithmic time. Merging two heaps R and R′ yields a new heap of size $x = \max(|\mathsf{R}|, |\mathsf{R}'|)$, which contains the x elements of R and R′ with the smallest keys. Instead of sorting the distances at the end, Algorithm 1 updates the ranking whenever a new distance between the query and a subtree of the document is available. The input ranking will be used later and is here assumed to be empty.

Example 3: We compute TASM-dynamic ($k = 2$) for query G and document H in Figure 2 (the cost for all nodes is 1, the input ranking is empty). Figure 5 shows the prefix and the tree distance matrixes that are filled by TASM-dynamic. Consider, for example, the prefix distance matrix between G_3 and H_6. The matrix is filled column by column, from left to right.

Algorithm 1: TASM-dynamic(Q, T, k, R)

```
Input: query $Q$, document $T$, number of matches $k$,
       (possibly empty) ranking R, $|\mathsf{R}| \leq k$
Output: top-$k$ ranking of the subtrees of $T$ w.r.t. $Q$
        merged with the input ranking R
begin
    td : empty $|Q| \times |T|$ matrix;
    pd : empty $(|Q|+1) \times (|T|+1)$ matrix;
    foreach relevant subtree $Q_m$ of $Q$ (ascending $m$) do
        foreach relevant subtree $T_n$ of $T$ (ascending $n$) do
            pd$[\emptyset, \emptyset] \leftarrow 0$;
            foreach $t_j \in V(T_n)$ (ascending) do
                pd$[\emptyset, t_j] \leftarrow$ pd$[\emptyset, t_{j-1}] + \gamma(\epsilon, t_j)$;
                foreach $q_i \in V(Q_m)$ (ascending) do
                    pd$[q_i, \emptyset] \leftarrow$ pd$[q_{i-1}, \emptyset] + \gamma(q_i, \epsilon)$;
                    if $Q_i = \text{pfx}(Q_m, q_i) \wedge T_j = \text{pfx}(T_n, t_j)$
                    then
                        pd$[q_i, t_j] \leftarrow \min($
                            pd$[q_{i-1}, t_j] + \gamma(q_i, \epsilon)$,
                            pd$[q_i, t_{j-1}] + \gamma(\epsilon, t_j)$,
                            pd$[q_{i-1}, t_{j-1}] + \gamma(q_i, t_j))$;
                        td$[Q_i, T_j] \leftarrow$ pd$[q_i, t_j]$;
                    else
                        pd$[q_i, t_j] \leftarrow \min($
                            pd$[q_{i-1}, t_j] + \gamma(q_i, \epsilon)$,
                            pd$[q_i, t_{j-1}] + \gamma(\epsilon, t_j)$,
                            pd$[q_{i-|Q_i|}, t_{j-|Q_j|}]$+td$[Q_i, T_j])$
                    end
                end
                if $Q_m = Q \wedge T_j = \text{pfx}(T_n, t_j)$ then
                    R $\leftarrow \text{push-heap}(\mathsf{R}, ($td$[Q_m, T_j], T_j))$;
                    if $|\mathsf{R}| > k$ then R $\leftarrow \text{pop-heap}(\mathsf{R})$;
                end
            end
        end
    end
    return R;
end
```

The element $\mathsf{pd}[g_2][h_5]$ stores the distance between the prefixes $\mathrm{pfx}(G_3, g_2)$ and $\mathrm{pfx}(H_6, g_5)$. The upper left element is 0 (Rule (a) in Figure 3); the first column stores the distances between the prefixes of G_3 and the empty prefix and is computed with Rule (b); similarly, the elements in the first row are computed with Rule (c); the shaded cells are distances between proper subtrees and are computed with formula (d); the remaining cells use formula (e). The shaded values of pd are copied to the tree distance matrix td. The two smallest distances in the last row are 0 (column 6) and 1 (column 3), thus the top-2 ranking is $R = (H_6, H_3)$.

TASM-dynamic constitutes the state-of-the-art for solving TASM. TASM-dynamic is a fairly efficient approach since it adds a minimal overhead to the already very efficient tree edit distance algorithm. The dynamic programming tree edit distance algorithm uses the result for subtrees to compute larger trees, thus no subtree distance is computed twice. Also, TASM-dynamic improves on the naive solution to TASM (Section 1) by a factor of $O(n)$ in terms of time. However, for each pair of relevant subtrees, Q_m and T_n, a matrix of size $O(|Q_m| \times |T_n|)$ must be computed in this algorithm. As a result, TASM-dynamic requires both the query and the document to be memory resident, leading to a space overhead that is prohibitive even for moderately large documents.

5 Prefix Ring Buffer

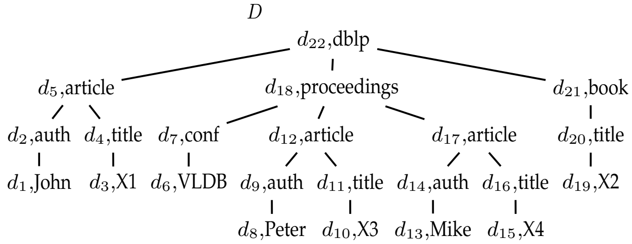
As will be discussed in Section 6, there is an effective bound on the size of the largest subtrees of a document that can be in the top-k best matches w.r.t. to a query. The key challenge in achieving an efficient solution to TASM is being able to prune large subtrees efficiently and perform the expensive tree edit distance computation on small subtrees only (for which computing the distance to the query is unavoidable). In this section we develop an essential piece of our solution to TASM, which is the *prefix ring buffer* together with a memory-efficient algorithm for pruning large subtrees. We also prove the correctness of our strategy.

The pruning algorithm uses a prefix ring buffer to produce the set of all subtrees that are within a given size threshold τ, but are not contained in a different subtree also within the threshold. This set of subtrees is called the *candidate set*.

Definition 9 (Candidate Set): Given a tree T and an integer threshold $\tau > 0$. The *candidate set of T for threshold τ* is defined as $cand(T, \tau) = \{T_i \mid t_i \in V(T), |T_i| \leq \tau, \forall t_a \in \mathrm{anc}(t_i) : |T_a| > \tau\}$. Each element of the candidate set is a *candidate subtree*.

Example 4: The candidate set of the example document D in Figure 6a for threshold $\tau = 6$ is $cand(D, 6) = \{D_5, D_7, D_{12}, D_{17}, D_{21}\}$.

We stress that the candidate set is *not* the set of all subtrees smaller than threshold τ, but a subset. If a subtree is contained in a different subtree that is also smaller than τ, then it is not in the candidate set. In the dynamic programming approach the distances for all subtrees of a candidate subtree T_i are computed as a side-effect of computing the distance for the candidate subtree T_i. Thus, subtrees of a candidate subtree need no separate computation.



(a) Example Document D

$$post(D) = ((\text{John}, 1), (\text{auth}, 2), (\text{X1}, 1), (\text{title}, 2), (\text{article}, 5), (\text{VLDB}, 1), (\text{conf}, 2), (\text{Peter}, 1), (\text{auth}, 2), (\text{X3}, 1), (\text{title}, 2), (\text{article}, 5), (\text{Mike}, 1), (\text{auth}, 2), (\text{X4}, 1), (\text{title}, 2), (\text{article}, 5), (\text{proceedings}, 13), (\text{X2}, 1), (\text{title}, 2), (\text{book}, 3), (\text{dblp}, 22))$$

(b) Postorder Queue of D

Fig. 6. Example Document and Corresponding Postorder Queue.

5.1 Memory Buffer

We now discuss how to compute the candidate set given a size threshold τ for a document represented as a postorder queue. Nodes that are dequeued from the postorder queue are appended to a memory buffer (see Figure 7) where the candidate subtrees are materialized. Once a candidate subtree is found, it is removed from the buffer, and its tree edit distance to the query is computed.

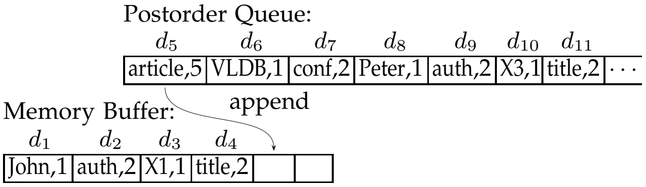


Fig. 7. Incoming Nodes are Appended to the Memory Buffer.

The nodes in the memory buffer form a prefix of the document (see Definition 7) consisting of one or more subtrees. All nodes of a subtree are stored at consecutive positions in the buffer: the leftmost leaf of the subtree is stored in the leftmost position, the root in the rightmost position. Each node that is appended to the buffer increases the prefix. New non-leaf nodes are ancestors of nodes that are already in the buffer. They either grow a subtree in the buffer or connect multiple subtrees already in the buffer into a new, larger, subtree.

Example 5: The buffer in Figure 7 stores the prefix $\text{pfx}(D, d_4)$ which consists of the subtrees D_2 and D_4. When node d_5 is appended, the buffer stores $\text{pfx}(D, d_5)$ which consists of a single subtree, D_5. The subtree D_5 is stored at positions 1 to 5 in the buffer: position 1 stores the leftmost leaf (d_1), position 5 the root (d_5).

The challenge is to keep the memory buffer as small as possible, i.e., to remove nodes from the buffer when they are no longer required. We distinguish the nodes in the postorder queue as candidate and non-candidate nodes: *candidate nodes* belong to candidate subtrees and must be buffered; *non-candidate nodes* are root nodes of subtrees that are too large for the candidate set. Non-candidate nodes are easily detected since the subtree size is stored with each node in the postorder queue. Candidate nodes must be buffered until all nodes of the candidate subtree are in the buffer. It is not obvious whether a subtree in the buffer is a candidate subtree, even if it is smaller than the threshold, because other nodes appended later may increase the subtree without exceeding τ.

5.2 Simple Pruning

A simple pruning approach is to append all incoming nodes to the buffer until a non-candidate node t_c is found. At this point, all subtrees rooted among t_c's children that are smaller than τ are candidate subtrees. They are returned and removed from the buffer. This approach must wait for the parent of a subtree root before the subtree can be returned. In the worst case, this requires to look $O(n)$ nodes ahead and thus a buffer of size $O(n)$ is required. Unfortunately, the worst case is a frequent scenario in data-centric XML with shallow and wide trees. For example, $\tau = 50$ is a reasonable threshold when matching articles in DBLP. However, over 99% of the 1.2M subtrees of the root node of DBLP are smaller than τ; with the simple pruning approach, all of them will be buffered until the root node is processed.

Example 6: Consider the example document in Figure 6. We use the simple approach to prune subtrees with threshold $\tau = 6$. The incoming nodes are appended to the buffer until a non-candidate arrives. The first non-candidate is d_{18} (represented by $(\text{proceedings}, 13)$), and all nodes appended up to this point (d_1 to d_{17}) are still in the buffer. The subtrees rooted in d_{18}'s children (d_7, d_{12}, and d_{17}) are in the candidate set. They are returned and removed from the buffer. The subtrees rooted in d_5 and d_{21} are returned and removed from the buffer when the root node arrives.

5.3 Ring Buffer Pruning

The simple pruning is not feasible for large documents. We now discuss the *ring buffer pruning* which buffers candidate trees only as long as necessary and uses a look-ahead of only $O(\tau)$ nodes. This is significant since the space complexity no longer depends on the document size.

The size of the *ring buffer* is $b = \tau + 1$. Two pointers are used: the start pointer s points to the first position in the ring buffer, the end pointer e to the position after the last element. The ring buffer is empty iff $s = e$, and the ring buffer is full iff $s = (e + 1) \,\%\, b$ ($\%$ is the modulo operator). The number of elements in the ring buffer is $(e - s + b) \,\%\, b \leq b - 1$. Two operations are defined on the ring buffer: (a) *remove* the leftmost node or subtree, (b) *append* node t_j. Removing the leftmost subtree T_i means incrementing s by $|T_i|$. Appending node t_j means storing node t_j at position e and incrementing e.

Example 7: The ring buffer $(\epsilon, d_1, d_2, d_3, d_4, d_5, d_6)$, $s = 1$, $e = 0$, is full. Removing the leftmost subtree, D_5, with 5 nodes, gives $s = 6$ and $e = 0$. Appending node d_7 results in $(d_7, d_1, d_2, d_3, d_4, d_5, d_6)$, $s = 6$, $e = 1$.

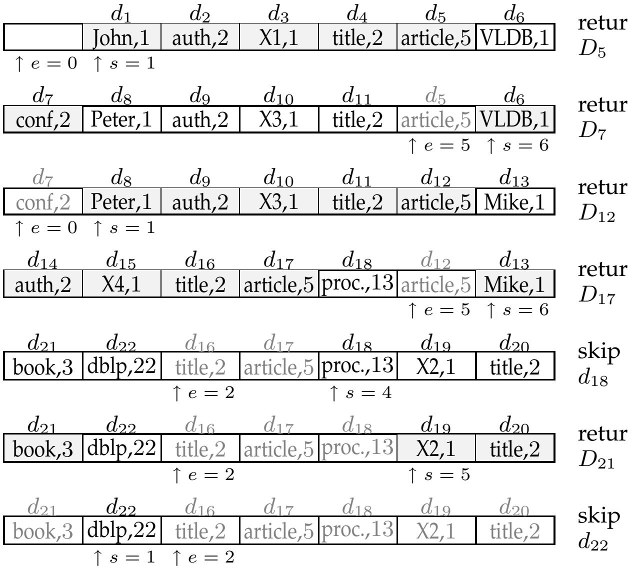
As the buffer is updated, it is possible that at a given point in time consecutive nodes in the buffer form a subtree that does not exist in the document. For example, nodes $(d_{13}, d_{14}, \ldots, d_{18})$ form a subtree with root node d_{18} that is different from D_{18}. We say a subtree in the buffer is *valid* if it exists in the document. In Section 5.5 we introduce the prefix array to find the leftmost valid subtree in constant time.

The ring buffer pruning of a postorder queue of a document T and an empty ring buffer of size $\tau + 1$ is as follows:

1) Dequeue nodes from the postorder queue and append them to a ring buffer until the ring buffer is full or the postorder queue is empty.
2) If the leftmost node of the ring buffer is a non-leaf, then remove it from the buffer, otherwise add the leftmost valid subtree to the candidate set and remove it from the buffer.
3) Go to 1) if the postorder queue is not empty; go to 2) if the postorder queue is empty but the ring buffer is not; otherwise terminate.

A non-leaf t_i appears at the leftmost buffer position if all its descendents are removed but t_i is not, for example, after removing the subtrees D_7, D_{12}, and D_{17}, the non-leaf d_{18} of document D is the leftmost node in the buffer.

Example 8: We illustrate the ring buffer pruning on the example tree in Figure 6. The ring buffer is initialized with $s = e = 1$. In Step 1 nodes d_1 to d_6 are appended to the ring buffer ($s = 1$, $e = 0$, see Figure 8). The ring buffer is full and we move to Step 2. The leftmost valid subtree, D_5, is returned and removed from the buffer ($s = 6$, $e = 0$). The postorder queue is not empty and we return to Step 1, where the ring buffer is filled for the next execution of Step 2. Figure 8 shows the ring buffer each time before Step 2 is executed. The shaded cells represent the subtree that is returned in Step 2. Note that in the fourth iteration D_{17} is returned, not the subtree rooted in d_{18}, since the subtree rooted in d_{18} is not valid. Nodes d_{18} and d_{22} are non-candidates and they are not returned. After removing d_{22} the buffer is empty and the algorithm terminates.



d_1 d_2 d_3 d_4 d_5 d_6
John,1 auth,2 X1,1 title,2 article,5 VLDB,1
↑ $e=0$ ↑ $s=1$
return D_5
d_7 d_8 d_9 d_{10} d_{11} d_5 d_6
conf,2 Peter,1 auth,2 X3,1 title,2 article,5 VLDB,1
↑ $e=5$ ↑ $s=6$
return D_7
d_7 d_8 d_9 d_{10} d_{11} d_{12} d_{13}
conf,2 Peter,1 auth,2 X3,1 title,2 article,5 Mike,1
↑ $e=0$ ↑ $s=1$
return D_{12}
d_{14} d_{15} d_{16} d_{17} d_{18} d_{12} d_{13}
auth,2 X4,1 title,2 article,5 proc.,13 article,5 Mike,1
↑ $e=5$ ↑ $s=6$
return D_{17}
d_{21} d_{22} d_{16} d_{17} d_{18} d_{19} d_{20}
book,3 dblp,22 title,2 article,5 proc.,13 X2,1 title,2
↑ $e=2$ ↑ $s=4$
skip d_{18}
d_{21} d_{22} d_{16} d_{17} d_{18} d_{19} d_{20}
book,3 dblp,22 title,2 article,5 proc.,13 X2,1 title,2
↑ $e=2$ ↑ $s=5$
return D_{21}
d_{21} d_{22} d_{16} d_{17} d_{18} d_{19} d_{20}
book,3 dblp,22 title,2 article,5 proc.,13 X2,1 title,2
↑ $s=1$ ↑ $e=2$
skip d_{22}

Fig. 8. Ring Buffer Pruning Example

5.4 Correctness

The ring buffer pruning classifies subtree T_i as candidate or non-candidate based on the nodes already buffered. Lemma 1 proves that this can be done by checking only the $\tau - |T_i|$ nodes that are appended after t_i and are ancestors of t_i: if all of these nodes are non-candidates, then T_i is a candidate tree. The intuition is that a parent of t_i that is appended later is an ancestor of both the nodes of t_i and the $\tau - |T_i|$ nodes that follow t_i; thus the new subtree must be larger than τ.

Example 9: Consider example document D of Figure 6a, $\tau = 6$. F_i is the set of $\tau - |D_i|$ nodes that are appended after d_i. The subtree D_2 is not in the candidate set since $F_2 = \{d_3, d_4, d_5, d_6\}$ contains d_5, which is an ancestor of d_2 and a candidate node. D_{21} is a candidate subtree: $|D_{21}| \leq \tau$, $F_{21} = \{d_{22}\}$, d_{22} is an ancestor of d_{21} and $|D_{22}| > \tau$. ($|F_{21}| < \tau - |D_{21}|$ since F_{21} contains the root node d_{22} which is the last node that is appended.)

Lemma 1: Let T be a tree, $cand(T, \tau)$ the candidate set of T for threshold τ, t_i the i-th node of T in postorder, and $F_i = \{t_j \mid t_j \in V(T), i < j \leq i - |T_i| + \tau\}$ the set of at most $\tau - |T_i|$ nodes following t_i in postorder. For all $1 \leq i \leq |T|$

$$\begin{aligned} &T_i \in cand(T, \tau) \Leftrightarrow \\ &\quad |T_i| \leq \tau \wedge \forall t_x (t_x \in F_i \cap \mathrm{anc}(t_i) \Rightarrow |T_x| > \tau) \end{aligned} \tag{1}$$

Proof: If $|T_i| > \tau$, then the left side of (1) is false since T_i is not a candidate tree, and the right side is false due to condition $|T_i| \leq \tau$, thus (1) holds. If $|T_i| \leq \tau$ we show

$$\begin{aligned} &(t_x \in F_i \cap \mathrm{anc}(t_i) \Rightarrow |T_x| > \tau) \Leftrightarrow \\ &\qquad\qquad (t_x \in \mathrm{anc}(t_i) \Rightarrow |T_x| > \tau), \end{aligned} \tag{2}$$

which makes (1) equivalent to the definition of the candidate set (cf. Definition 9). Case $i + \tau - |T_i| \geq |T|$: F_i contains all nodes after t_i in postorder, thus $F_i \cap \mathrm{anc}(t_i) = \mathrm{anc}(t_i)$ and (2) holds. Case $i + \tau - |T_i| < |T|$: (2) holds for all $t_x \in F_i \cap \mathrm{anc}(t_i)$. If $t_x \in \mathrm{anc}(t_i) \setminus F_i$, then $t_x \notin F_i \cap \mathrm{anc}(t_i)$ and the left side of (2) is true. Since any $t_x \in \mathrm{anc}(t_i) \setminus F_i$ is an ancestor of all nodes of both T_i and F_i, $|T_x| > |T_i| + |F_i| = \tau$, and (2) holds. □

As illustrated in Figure 8 the ring buffer pruning removes either candidate subtrees or non-candidate nodes from the buffer. After each remove operation the leftmost node in the buffer is checked. If the leftmost node is a leaf, then it starts a candidate subtree, otherwise it is non-candidate node.

Lemma 2: Let T be an ordered labeled tree, $cand(T, \tau)$ be the candidate set of T for threshold τ, t_s be the next node of T in postorder after a non-candidate node or after the root node of a candidate subtree, or $t_s = t_1$, and $lml(t_i)$ be the leftmost leaf descendant of the root t_i of subtree T_i.

$$\begin{aligned} &\mathrm{t}_s \text{ is a leaf} \Rightarrow \\ &\qquad \exists T_i : T_i \in cand(T, \tau), t_s = lml(t_i) \\ &\mathrm{t}_s \text{ is a non-leaf } \Rightarrow \\ &\qquad t_s \in \{t_x \mid t_x \in V(T), |T_x| > \tau\} \end{aligned} \tag{3}$$

Proof: Let NC be the non-candidate nodes of T.

(a) $t_s = t_1$: t_1 is a leaf, thus $t_1 \notin NC$ and there is a $t_i \in cand(T, \tau)$ such that $t_1 \in V(T_i)$. There is no node $t_k < t_1$, thus $t_1 = lml(t_i)$.

(b) t_s follows the root node of a candidate subtree T_j: t_s is either the parent t_k of the root node of T_j or a leaf descendant t_l of t_k. $t_k \in NC$ by Definition 9. Since t_l is a leaf, $t_l \notin NC$ and there must be a $T_i \in cand(T, \tau)$ such that $t_l \in V(T_i)$. We prove $t_l = lml(t_i)$ by contradiction: Assume T_i has a leaf t_x to the left of t_l. As $V(T_j) \cap V(T_i) = \emptyset$, t_x is to the left of t_j, and $t_a \in V(T_i)$, the least common ancestor of t_l and t_x, is an ancestor of t_k. This is not possible since $|T_k| > \tau \Rightarrow |T_a| > \tau \Rightarrow |T_i| > \tau$.

(c) t_s follows a non-candidate node, $t_x \in NC$: t_s is either the parent t_k of t_x or a leaf node t_l. $t_k \in NC$ by Definition 9, and there is a $T_i \in cand(T, \tau)$ such that $t_l = lml(t_i)$ (same rationale as above). □

Theorem 1 (Correctness of Ring Buffer Pruning): Given a document T and a threshold τ, the ring buffer pruning adds a subtree T_i of T to the candidate set iff $T_i \in cand(T, \tau)$.

Proof: We show that (1) each node of T is processed, i.e., either skipped or output as part of a subtree, and (2) the pruning in Step 2 is correct, i.e., non-candidate nodes are skipped and candidate subtrees are returned.

(1) All nodes of T are appended to the ring buffer: Steps 1 and 2 are repeated until the postorder queue is empty. In each cycle nodes are dequeued from the postorder queue and appended to the ring buffer. All nodes of the ring buffer are processed: The nodes are systematically removed from the ring buffer from left to right in Step 2, and Step 2 is repeated until both the postorder queue and the ring buffer are empty.

(2) Let t_s be the smallest node of the ring buffer. If t_s is the leftmost leaf of a candidate subtree, then the leftmost valid subtree, T_i, is a candidate subtree: Since the buffer is either full or contains the root node of T when Step 2 is executed, all nodes $F_i = \{t_j | t_j \in V(T), i < j \leq i - |T_i| + \tau\}$ are in the buffer. If a node $t_k \in F_i$ is an ancestor of t_i, then $|T_k| > \tau$: If t_s is the smallest leaf of T_k, then T_k is the leftmost valid subtree which contradicts the assumption; if the smallest leaf of T_k is smaller than t_s, then T_k is not a candidate subtree since it contains t_s which is the leftmost leaf of a candidate subtree; since t_k is an ancestor of t_s, the smallest leaf of T_k can not be larger than t_s. With Lemma 1 it follows that T_i is a candidate subtree. As T_i is a candidate subtree, with Lemma 2 the pruning in Step 2 is correct. □

5.5 Prefix Array

The ring buffer pruning removes the leftmost valid subtree from the ring buffer. A subtree is stored as a sequence of nodes that starts with the leftmost leaf and ends with the root node. A node is a $(label, size)$ pair, and in the worst case we need to scan the entire buffer to find the root node of the leftmost valid subtree. To avoid the repeated scanning of the buffer we enhance the ring buffer with a *prefix array* which encodes tree prefixes (see Definition 7). This allows us to find the leftmost valid subtree in constant time.

Definition 10 (Prefix Array): Let $\mathrm{pfx}(T, t_p)$ be a prefix of T, and $t_i \in V(T)$, $1 \leq i \leq p$, be the i-th node of T in postorder. The *prefix array for* $\mathrm{pfx}(T, t_p)$ is an integer array $(a_1, a_2, \ldots, a_p)$ where a_i is the smallest descendant of t_i if t_i is a non-leaf node, otherwise the largest ancestor of t_i in $\mathrm{pfx}(T, t_p)$ for which t_i is the smallest descendant:

$$a_i = \begin{cases} \max\{x | x \in \mathrm{pfx}(T, t_p), lml(x) = t_i\} & \text{if } t_i \text{ is a leaf} \\ lml(t_i) & \text{otherwise} \end{cases}$$

A new node t_{p+1} is appended to the prefix array $(a_1, a_2, \ldots, a_p)$ by appending the integer $a_{p+1} = lml(t_{p+1})$ and updating the ancestor pointer of its smallest descendant, $a_{(a_{p+1})} = a_{p+1}$. A node t_i is a leaf iff $a_i \geq i$. The largest valid subtree in the prefix with a given leftmost leaf t_i is $(a_i, a_{i+1}, \ldots, a_{(a_i)})$ and can be found in constant time.

Example 10: Figure 9 shows the prefix arrays of different prefixes of the example tree D and illustrates the structure of the prefix arrays with arrows. The prefix array for $\mathrm{pfx}(D, d_4)$ is $(2, 1, 4, 3)$. We append d_5 and get $(5, 1, 4, 3, 1)$ (the smallest descendant of d_5 is d_1, thus $a_5 = 1$ is appended and a_1 is updated to 5). Appending d_6 gives $(5, 1, 4, 3, 1, 6)$. The largest valid subtree in the prefix $\mathrm{pfx}(D, d_6)$ with the leftmost leaf d_1 is $(5, 1, 4, 3, 1)$ ($i = 1$, $a_i = 5$).

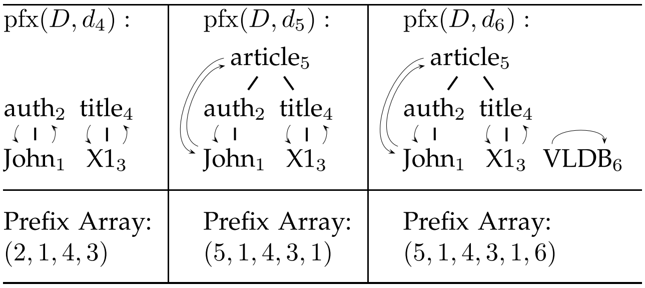


Fig. 9. The Prefix Arrays of Three Prefixes.

The pruning removes nodes from the left of the prefix ring buffer such that the prefix ring buffer stores only part of the prefix. The pointer from a leaf to the largest valid subtree in the prefix always points to the right and is not affected. This pointer changes only when new nodes are appended.

Theorem 2: The prefix ring buffer pruning for a document with n nodes and with threshold τ runs in $O(n)$ time and $O(\tau)$ space.

Proof: Runtime: Each of the n nodes is processed exactly once in Step 1 and in Step 2, then the algorithm terminates. Dequeuing a node from the postorder queue and appending it to the prefix ring buffer in Step 1 is done in constant time. Removing a node (either as non-candidate or as part of a subtree) in Step 2 is done in constant time. Space: The size of the prefix ring buffer is $O(\tau)$. No other data structure is used. □

5.6 Algorithm

Algorithm 2 (prb-pruning) implements the ring buffer pruning and computes the candidate set $cand(T, \tau)$ given the size threshold τ and the postorder queue, pq, of document T. The prefix ring buffer is realized with two ring buffers of size $b = \tau + 1$: lbl stores the node labels and pfx encodes the structure as a prefix array. The ring buffers are used synchronously and share the same start and end pointers (s,e). Counter c counts the nodes that have been appended to the prefix ring buffer.

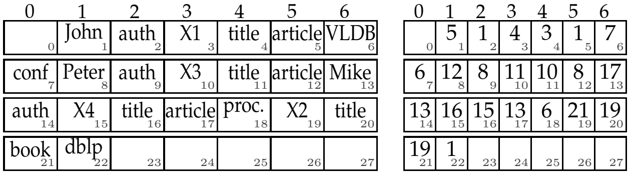
After each call of prb-next (Algorithm 3) a candidate subtree is ready at the start position of the prefix ring buffer. It is added to the candidate set and removed from the buffer (Lines 6 and 7). prb-subtree(pfx, lbl, a, b) returns the subtree formed by nodes a to b in the prefix ring buffer. Algorithm 3 is called until the ring buffers are empty.

```
Algorithm 2: prb-pruning(pq, $\tau$)
  Input: postorder queue pq of a document $T$, threshold $\tau$
  Output: candidate set $cand(T, \tau)$
  begin
      pfx, lbl: ring buffers of size $b = \tau + 1$;
      $C \leftarrow \emptyset$;
      (pfx, lbl, $s, e, c$, pq) $\leftarrow$ prb-next(pfx, lbl, $1, 1, 0$, pq, $\tau$);
      while $s \neq e$ do
          $C \leftarrow C \cup \{$prb-subtree(pfx, lbl, $s$, pfx[$s$])$\}$;
          $s \leftarrow$ (pfx[$s$] $+ 1$) % $b$;
          (pfx, lbl, $s, e, c$, pq) $\leftarrow$
              prb-next(pfx, lbl, $s, e, c$, pq, $\tau$);
      end
      return $C$;
  end
```

Algorithm 3 loops until both the postorder queue and the prefix ring buffer are empty. If there are still nodes in the postorder queue (Line 3), they are dequeued and appended to the prefix ring buffer, and the ancestor pointer in the prefix array is updated (Line 8). If the prefix ring buffer is full or the postorder queue is empty (Line 11), then nodes are removed from the prefix ring buffer. If the leftmost node is a leaf (Line 12, $c + 1 - (e - s + b)$ % b is the postorder identifier of the leftmost node), a candidate subtree is returned, otherwise a non-candidate is skipped.

```
Algorithm 3: prb-next(pfx, lbl, $s, e, c$, pq, $\tau$)
   Input: ring buffers pfx and lbl with start/end pointers
          $s$ and $e$, counter $c$ of nodes appended so far,
          (partially consumed) postorder queue pq of a
          document $T$, threshold $\tau$
   Output: next subtree $T_i \in cand(T, \tau)$
 1 begin
 2     $b \leftarrow \tau + 1$ // ring buffer size
 3     while pq $\neq \emptyset$ or $s \neq e$ do
 4         if pq $\neq \emptyset$ then
 5             (pq, ($\lambda$, $size$)) $\leftarrow$ dequeue(pq);
 6             lbl[$e$] $\leftarrow \lambda$;
 7             pfx[$e$] $\leftarrow$ (++$c$) $-$ $size$;
 8             if $size \leq \tau$ then pfx[pfx[$e$] % $b$] $\leftarrow c$;
 9             $e \leftarrow (e + 1)$ % $b$;
10         end
11         if $s = (e + 1)$ % $b$ or pq $= \emptyset$ then
12             if pfx[$s$] $\geq c + 1 - (e - s + b)$ % $b$ then
13                 return (pfx, lbl, $s, e, c$, pq);
14             else
15                 $s \leftarrow (s + 1)$ % $b$;
16             end
17         end
18     end
19     return (pfx, lbl, $s, e, c$, pq);
20 end
```

Example 11: Figure 10 illustrates the prefix ring buffer for the example document D in Figure 6. The relative positions in the ring buffer are shown at the top. The small numbers are the postorder identifiers of the nodes. The ring buffers are filled from left to right; overwritten values are shown in the next row.



0	1	2	3	4	5	6
$_{0}$	John $_{1}$	auth $_{2}$	X1 $_{3}$	title $_{4}$	article $_{5}$	VLDB $_{6}$
conf $_{7}$	Peter $_{8}$	auth $_{9}$	X3 $_{10}$	title $_{11}$	article $_{12}$	Mike $_{13}$
auth $_{14}$	X4 $_{15}$	title $_{16}$	article $_{17}$	proc. $_{18}$	X2 $_{19}$	title $_{20}$
book $_{21}$	dblp $_{22}$	$_{23}$	$_{24}$	$_{25}$	$_{26}$	$_{27}$

Ring Buffer lbl

0	1	2	3	4	5	6
$_{0}$	5 $_{1}$	1 $_{2}$	4 $_{3}$	3 $_{4}$	1 $_{5}$	7 $_{6}$
6 $_{7}$	12 $_{8}$	8 $_{9}$	11 $_{10}$	10 $_{11}$	8 $_{12}$	17 $_{13}$
13 $_{14}$	16 $_{15}$	15 $_{16}$	13 $_{17}$	6 $_{18}$	21 $_{19}$	19 $_{20}$
19 $_{21}$	1 $_{22}$	$_{23}$	$_{24}$	$_{25}$	$_{26}$	$_{27}$

Prefix Array pfx

Fig. 10. Implementation of the Prefix Ring Buffer.

6 TASM-POSTORDER

We now present a solution for TASM whose space complexity is independent of the document size and, thus, scales well to XML documents that do not fit into memory. Unlike TASM-dynamic (Section 4.6), which requires the whole document in memory, our solution uses the prefix ring buffer and keeps only candidate subtrees in memory at any point in time. We start the section by showing an effective threshold τ for the size of the largest candidate subtree in the document. Then we present TASM-postorder and prove its correctness.

6.1 Upper Bound on Candidate Subtree Size

Recall that solving TASM consists of finding a ranking of the subtrees of the document according to their tree edit distance to a query. We distinguish intermediate and final rankings. An *intermediate* ranking, $R' = (T_{i'_1}, T_{i'_2}, \ldots, T_{i'_k})$, is the top-$k$ ranking of a subset of at least k subtrees of a document T with respect to a query Q, the *final* ranking, $R = (T_{i_1}, T_{i_2}, \ldots, T_{i_k})$, is the top-$k$ ranking of all subtrees of document T with respect to the query.

We show that any intermediate ranking provides an upper bound for the maximum subtree size that must be considered (Lemma 4). The tightness of such a bound improves with the quality of the ranking, i.e., with the distance between the query and the lowest ranked subtree. We initialize the intermediate ranking with the first k subtrees of the document in postorder. Lemma 5 provides bounds for the size of these subtrees and their distance to the query. The ranking of the first k subtrees provides the upper bound $\tau = |Q|(c_Q + 1) + kc_T$ for the maximum subtree size that must be considered (Theorem 3), where c_Q and c_T denote the maximum costs of any node in Q and the first k nodes in T, respectively (cf. Section 4.4). Note that this upper bound τ is independent of size and structure of the document

Lemma 3: Let Q and T be ordered labeled trees, then $|T| \leq \delta(Q, T) + |Q|$.

Proof: We show $|T| - |Q| \leq \delta(Q, T)$. True for $|T| \leq |Q|$ since $\delta(Q, T) \geq 0$. Case $|T| > |Q|$: At least $|T| - |Q|$ nodes must be inserted to transform Q into T. The cost of inserting a new node, t_x, into T is $\gamma(\epsilon, t_x) = cst(t_x) \geq 1$. □

Lemma 4 (Upper Bound): Let $R' = (T_{i'_1}, T_{i'_2}, \ldots, T_{i'_k})$ be any intermediate ranking of at least k subtrees of a document T with respect to a query Q, and let R be the final top-k ranking of all subtrees of T, then $\forall T_{i_j}(T_{i_j} \in R \Rightarrow |T_{i_j}| \leq \delta(Q, T_{i'_k}) + |Q|)$.

Proof: $|T_{i_j}| \leq \delta(Q, T_{i_j}) + |Q|$ follows from Lemma 3. We show $\forall T_{i_j}(|T_{i_j}| \in R \Rightarrow \delta(Q, T_{i_j}) \leq \delta(Q, t'_{i_k}))$ by contradiction: Assume a subtree $T_{i_j} \in R$, $\delta(Q, T_{i_j}) > \delta(Q, T_{i'_k})$. Then by Definition 1 also $T_{i'_k} \in R$; if $T_{i'_k} \in R$, then also all other $T_{i'_l} \in R'$ are in R, i.e., $R' \subseteq R$. $T_{i_j} \notin R'$ (since $\delta(Q, T_{i_j}) > \delta(Q, T_{i'_k})$) but $T_{i_j} \in R$, thus $R' \cup \{T_{i_j}\} \subseteq R$. This contradicts $|R| = k$. □

Lemma 5 (First Ranking): Let Q and T be ordered labeled trees, $k \leq |T|$, c_Q and c_T be the maximum costs of a node in Q and the first k nodes in T, respectively, t_i be the i-th node of T in postorder, then for all $T_i, 1 \leq i \leq k$, the following holds: $|T_i| \leq k \wedge \delta(Q, T_i) \leq |Q|c_Q + kc_T$.

Proof: Let q_i be the i-th node of Q in postorder, and $lml(t_i)$ the leftmost leaf of T_i. The nodes of a subtree have consecutive postorder numbers. The smallest node is the leftmost leaf, the largest node is the root. Since the leftmost leaf of T_i, $1 \leq i \leq k$, is larger or equal 1 and the root is at most k, the subtree size is bound by k. The distance between the query and the document is maximum if the edit mapping is empty, i.e., all nodes of Q are deleted and all nodes of T_i are inserted: $\delta(Q, T_i) \leq \sum_{q_i \in V(Q)} \gamma(q_i, \epsilon) + \sum_{t_i \in V(T_i)} \gamma(\epsilon, t_i) \leq |Q|c_Q + kc_T$ since $\gamma(q_i, \epsilon) \leq c_Q$, $\gamma(\epsilon, t_i) \leq c_T$, and $|T_i| \leq k$. □

The three lemmas above are the elements for our main result in this section:

Theorem 3 (Maximum Subtree Size): Let query Q and document T be ordered labeled trees, c_Q and c_T be the maximum costs of a node in Q and the first k nodes in T, respectively, $R = (T_{i_1}, T_{i_2}, \ldots, T_{i_k})$ be the final top-k ranking of all subtrees of T with respect to Q, then the size of all subtrees in R is bound by $\tau = |Q|(c_Q + 1) + kc_T$:

$$\forall T_{i_j}(T_{i_j} \in R \Rightarrow |T_{i_j}| \leq |Q|(c_Q + 1) + kc_T) \qquad (4)$$

Proof: $|T| < k$: (4) holds since $|T_{i_j}| \leq |T| < k \leq |Q|(c_Q + 1) + kc_T$. $|T| \geq k$: According to Lemma 5 there is an intermediate ranking $R' = (T_{i'_1}, T_{i'_2}, \ldots, T_{i'_k})$ with $\delta(Q, T_{i'_k}) \leq |Q|c_Q + kc_T$, thus $\delta(Q, T_{i_j}) \leq |Q|c_Q + kc_T$ (Lemma 4) and $|T_{i_j}| \leq |Q|c_Q + kc_T + |Q|$ (Lemma 3) for all subtrees $T_{i_j} \in R$. □

6.2 Algorithm

TASM-postorder (Algorithm 4) uses the upper bound τ (see Theorem 3) to limit the size of the subtrees that must be considered, and the set of candidate subtrees, $cand(T, \tau)$, is computed using the prefix ring buffer proposed in Section 5. When a candidate subtree $T_i \in cand(T, \tau)$ is available in the prefix ring buffer (Lines 5 and 19), it is processed and removed (Line 18). If an intermediate ranking is available (i.e., $|\mathsf{R}| = k$) the upper bound τ' provided by the intermediate ranking (see Lemma 4) may be tighter than τ. Only subtrees of T_i that are smaller than τ' must be considered. The subtrees of T_i (including T_i itself) are traversed in reverse postorder, i.e., in descending order of the postorder numbers of their root nodes. If a subtree of T_i is below the size threshold τ', then TASM-dynamic is called for this subtree and the ranking R is updated. All subtrees of the processed subtree are skipped (Line 13), and the remaining subtrees of T_i are traversed in reverse postorder.

Algorithm 4: TASM- postorder(Q, pq, k)

```
   Input: query $Q$, postorder queue pq of a document $T$,
          result size $k$
   Output: top-$k$ ranking of the subtrees of $T$ w.r.t. $Q$
 1 begin
 2     R : empty max-heap; // top-k ranking
 3     $\tau \leftarrow |Q|(c_Q + 1) + kc_T$; $\tau' \leftarrow \tau$;
 4     pfx, lbl: ring buffers of size $b = \tau + 1$;
 5     (pfx, lbl, $s, e, c$, pq) $\leftarrow$ prb-next(pfx, lbl, 1, 1, 0, pq, $\tau$);
 6     while $s \neq e$ do
 7         $r \leftarrow$ pfx[$s$] // candidate subtree root
 8         while $r \geq$ pfx[pfx[$s$] % $b$] do
 9             $T_i \leftarrow$ prb-subtree(pfx, lbl, pfx[$r$ % $b$], $r$ % $b$);
10             if |R| $= k$ then $\tau' = \min(\tau, \max(\mathsf{R}) + |Q|)$;
11             if |R| $< k \vee |T_i| < \tau'$ then
12                 R $\leftarrow$ TASM- dynamic($Q, T_i, k$, R);
13                 $r \leftarrow r - |T_i|$;
14             else
15                 $r \leftarrow r - 1$;
16             end
17         end
18         $s \leftarrow$ (pfx[$s$] + 1) % $b$;
19         (pfx, lbl, $s, e, c$, pq) $\leftarrow$
               prb-next(pfx, lbl, $s, e, c$, pq, $\tau$);
20     end
21     return R;
22 end
```

Theorem 4 (Correctness): Given a query Q, a document T, and $k \leq |T|$, TASM-postorder (Algorithm 4) computes the top-k ranking R of all subtrees of T with respect to Q.

Proof: If no intermediate ranking is available, all subtrees within size $\tau = |Q|(c_Q + 1) + kc_T$ are considered. The correctness of τ follows from Theorem 3. Subtrees of size $\tau' = \min(\tau, \max(\mathsf{R}) + |Q|)$ and larger are pruned only if an intermediate ranking with k subtrees is available. Then the correctness of τ' follows from Lemma 4. □

Theorem 5 (Complexity): Let Q and T be ordered labeled trees, $m = |Q|$, $n = |T|$, $k \leq |T|$, c_Q and c_T be the maximum costs of a node in Q and the first k nodes in T, respectively. Algorithm 4 uses $O(m^2 n)$ time and $O(m^2 c_Q + mkc_T)$ space.

Proof: The space complexity of Algorithm 4 is dominated by the call of TASM-dynamic(Q, T_i, k, R)

in Line 12, which requires $O(m|T_i|)$ space. Since $|T_i| \leq \tau = m(c_Q + 1) + kc_T$, the overall space complexity is $O(m^2 c_Q + mkc_T)$. The runtime of TASM-dynamic(Q, T_i, k, R) is $O(m^2|T_i|)$. τ is the size of the maximum subtree that must be computed. There can be at most n/τ subtrees of size τ in the document and the runtime complexity is $O(\frac{n}{\tau}m^2\tau) = O(m^2 n)$. □

The space complexity is independent of the document size. c_Q and c_T are typically small constants, for example, $c_Q = c_T = 1$ for the unit cost tree edit distance, and the document is often much larger than the query. For example, a typical query for an article in DBLP has 15 nodes, while the document has 26M nodes. If we look for the top 20 articles that match the query using the unit cost edit distance, TASM-postorder only needs to consider subtrees up to a size of $\tau = 2|Q| + k = 50$ nodes, compared to 26M in TASM-dynamic. Note that for TASM-postorder a subtree with 50 nodes is the worst case, whereas TASM-dynamic always computes the distance between the query and the whole document with 26M nodes.

6.3 Pushing Pruning into TASM-dynamic

TASM-postorder calls TASM-dynamic for document subtrees that can not be pruned. TASM-dynamic computes the distances between the query and all subtrees. In this section we apply our pruning rules inside TASM-dynamic and stop the execution early, i.e., before all matrixes are filled. We leverage the fact that the ranking improves during the execution of TASM-dynamic, giving rise to a tighter upper bound for the maximum subtree size.

We refer to TASM-dynamic with pruning as TASM-dynamic$^+$ (Algorithm 5). The pruning is inserted between Lines 7 and 8 of TASM-dynamic, all other parts remain unchanged. Whenever the pruning condition holds, the unprocessed columns of the current prefix distance matrix (pd) are skipped.

Algorithm 5: TASM-dynamic$^+(Q, T, k, \mathsf{R})$

```
Input: query $Q$, document $T$, number of matches $k$,
       (possibly empty) ranking $\mathsf{R}$, $|\mathsf{R}| \leq k$
Output: top-$k$ ranking of the subtrees of $T$ w.r.t. $Q$
        merged with the input ranking $\mathsf{R}$
1  begin
       ...
7      foreach $t_j \in V(T_n)$ (ascending) do
           if $|\mathsf{R}| = k \wedge |\mathrm{pfx}(T_n, t_j)| > \max(\mathsf{R}) + |Q|$ then
               goto line 28;  // exit inner loop
           end
8          $\mathsf{pd}[\emptyset, t_j] \leftarrow \mathsf{pd}[\emptyset, t_{j-1}] + \gamma(\epsilon, t_j)$
           ...
28     end
       ...
31     return $\mathsf{R}$
32 end
```

Example 12: We compute TASM-dynamic$^+$ ($k = 2$) for the query G and the document H in Figure 2 (the cost for all nodes is 1, the input ranking is empty). The gray values in the prefix and tree distance matrixes in Figure 5 are the values that TASM-dynamic$^+$ does not need to compute due to the pruning. Before column h_5 in the prefix distance matrix between G_3 and H_7 is computed, $\mathsf{R} = ((H_6, 0), (H_3, 1))$ and the pruning condition holds ($|\mathsf{R}| = 2$, $|\mathrm{pfx}(H_7, h_5)| = 5$, $\max(\mathsf{R}) = 1$, $|G| = 3$). The columns h_5, h_6, and h_7 can be skipped and the distances $\delta(G_1, H_7)$ and $\delta(G_3, H_7)$ need not be computed.

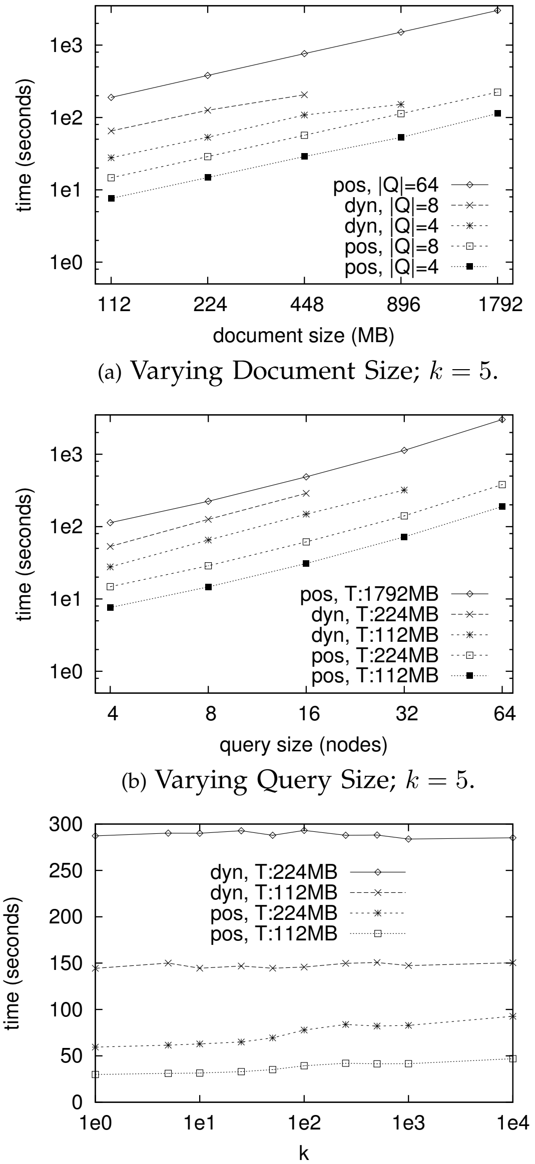
Theorem 6 (Correctness of TASM-dynamic$^+$): Given a query Q, a document T, $k \leq |T|$, and a ranking R of at most k subtrees with respect to the query Q, TASM-dynamic$^+$ (Algorithm 5) computes the top-k ranking of the subtrees in the ranking R and all subtrees of document T with respect to the query Q.

Proof: Without pruning the algorithm computes all distances between the query Q and the subtrees of document T. Whenever a new distance is available, the ranking is updated and the final ranking R is correct. If the pruning condition holds for a prefix $\mathrm{pfx}(T_n, t_j)$ of the relevant subtree T_n, then column t_j of the prefix distance matrix pd, all following columns of pd, and some values of the tree distance matrix td will not be computed. We need to show that (1) we do not miss a subtree that should be in the final ranking R, and (2) the values of td that are not computed are not needed later.

(1) Let $p_i = \mathrm{pfx}(T_n, t_i)$ be a prefix of T_n. We need to show $\forall p_i (t_i \geq t_j \Rightarrow p_i \notin R)$: If p_i is not a subtree then $p_i \notin R$ (Definition 1). If p_i is a subtree, $p_i \notin R$ follows from Lemma 4: Since the pruning condition requires $|\mathsf{R}| = k$, an intermediate ranking $(T_{i'_1}, T_{i'_2}, \ldots, T_{i'_k})$ is available and $\delta(Q, T_{i'_k}) = \max(\mathsf{R})$; thus a subtree T_i can not be in the final ranking if $|T_i| > \max(\mathsf{R}) + |Q|$. $|\mathrm{pfx}(T_n, t_j)| > \max(\mathsf{R}) + |Q|$ (pruning condition) and $p_i \geq |\mathrm{pfx}(T_n, t_j)|$ for $t_i \geq t_j$, thus $p_i \notin R$.

(2) Let pd be the prefix distance matrix between two relevant subtrees Q_m and T_n. A column t_j of pd can be computed if (a) all columns of pd to the left of t_j are filled, and (b) all prefix distance matrixes between T_n and the relevant subtrees Q_i of Q_m ($Q_i \neq Q_m$) are filled up to column t_j (follows from the decomposition rules in Figure 3). (a) holds since the columns are computed from left to right, and columns to the right of a pruned column are pruned as well; (b) holds since the prefix distance matrixes for the subtrees Q_i are computed before pd, and if the pruning condition holds for column t_j in the matrix of a subtree Q_i, then it also holds for column t_j in the matrix of Q_m (in the pruning condition, $|\mathrm{pfx}(T_n, t_j)|$ and $|Q|$ do not change and $\max(\mathsf{R})$ can not increase). □

We adapt TASM-postorder (Algorithm 4) by replacing TASM-dynamic with TASM-dynamic$^+$ in Line 12 and use this version of the algorithm in our experimental evaluation.



(a) Varying Document Size; $k = 5$.

(b) Varying Query Size; $k = 5$.

(c) Varying k; $|Q| = 16$.

Fig. 11. Execution Times for Varying Sizes of Document, Query and k.

7 Experimental Validation

In this section we experimentally evaluate our solution. We study the scalability of TASM-postorder using realistic synthetic XML datasets of varying sizes and the effectiveness of the prefix ring buffer pruning on large real world datasets. All algorithms were implemented as single-thread applications in Java 1.6 and run on a dual-core AMD64 server. A standard XML parser was used to implement the postorder queues (i.e., parse and load documents and queries). In all algorithms we use a dictionary to assign unique integer identifiers to node labels (element/attribute tags as well as text content). The integer identifiers provide compression and faster node-to-node comparisons, resulting in overall better scalability.

7.1 Scalability

We study the scalability of TASM-postorder using synthetic data from the standard XMark benchmark [25], whose documents combine complex structures and realistic text. There is a linear relation between the size of the XMark documents (in MB) and the number of nodes in the respective XML trees; the height does not vary with the size and is 13 for all documents. We used documents ranging from 112MB and 3.4M nodes to 1792MB and 55M nodes. The queries are randomly chosen subtrees from one of the XMark documents with sizes varying from 4 to 64 nodes. For each query size we have four trees. We compare TASM-postorder against the state-of-the-art solution, TASM-dynamic (Section 4.6) implemented using the tree edit distance algorithm by Zhang and Shasha [9].

Execution Time: Figure 11a shows the execution time as a function of the document size for different query sizes $|Q|$ and fixed $k = 5$. Similarly, Figure 11b shows the execution time versus query size (from 4 to 64 nodes) for different document sizes $|T|$ and fixed $k = 5$. The graphs show averages over 20 runs. The data points missing in the graphs correspond to settings in which TASM-dynamic runs out of main memory (4GB). As predicted by our analysis (Section 6), the runtime of TASM-postorder is linear in the document size. TASM-postorder scales very well with both the document and the query size, and can handle very large documents or queries. In contrast, TASM-dynamic runs out of memory for trees larger than 500MB, except for very small queries. Besides scaling to much larger problems, TASM-postorder is also around four times faster than TASM-dynamic.

Figure 11c shows the impact of parameter k on the execution time of TASM-postorder ($|Q| = 16$). As expected, TASM-dynamic is insensitive to k since it always must compute all subtrees. TASM-postorder, on the other hand, prunes large subtrees, and the size of the pruned subtrees depends on k. As the graph shows (observe the log-scale on the x-axis), TASM-postorder scales extremely well with k: an increase of 4 orders of magnitude in k results only in doubling the low runtime.

Figure 12 compares the execution times of TASM-dynamic$^+$ and TASM-dynamic. TASM-dynamic$^+$ is, on average, 45% faster than TASM-dynamic since distance computations to large subtrees are pruned.

Main Memory Usage: Figure 13 compares the main memory usage of TASM-postorder and TASM-dynamic for different document sizes. The graph shows the average memory used by the Java virtual machine over 20 runs for each query and document size. (The memory used by the virtual machine depends on several factors and is not constant across runs.) We omit the plots for other query sizes since they follow the same trend as the ones shown in Figure 13: the memory requirements are independent of

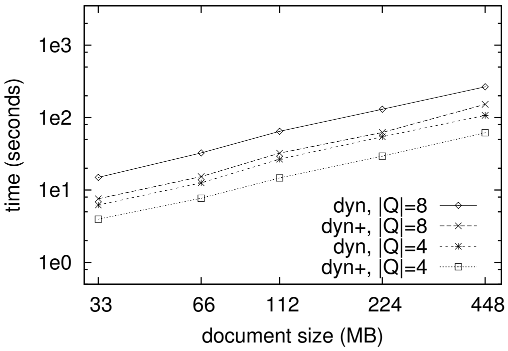


Fig. 12. TASM-dynamic$^+$ vs. TASM-dynamic; $k = 5$.

the document size for TASM-postorder and linearly dependent on the document size for TASM-dynamic. In both cases the experiment agrees with our analysis. The missing points in the plot correspond to settings for which TASM-dynamic runs out of memory (4GB). The difference in memory usage is remarkable: while for TASM-postorder only small subtrees need to be loaded to main memory, TASM-dynamic requires data structures in main memory that are much larger than the document itself.

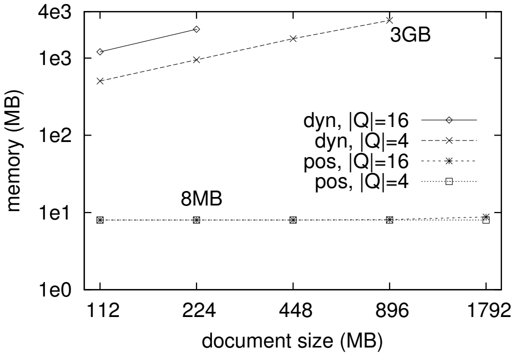


Fig. 13. Memory Usage as a Function of the Document Size; $k = 5$.

7.2 TASM-postorder vs. XQuery

In order to give a feel for the overall performance of TASM-postorder we compare its execution time against XQuery-based twig queries that find exact matches of the query tree. This can be seen as a very restricted solution to TASM and is the special case when $k = 1$ *and* an identical copy of the query exists in the document. For example, query G in Figure 2 can be expressed as follows:

```
for $v1 in //a[count(node()) eq 2]
let $v2:=$v1/b[1][not (node())],
$v3:=$v1/c[1][not (node())]
where $v2 << $v3
return node-name($v1)
```

We used Saxon [26], a state-of-the-art main-memory, Java-based XQuery processor in our tests. Figure 14 shows the results. As another reference point, the graph shows the cost of parsing each document using SAX. Compared to the XQuery program (xq-twig), TASM-postorder is on average only 26% slower. With respect to SAX, TASM-postorder is within one order of magnitude. xq-twig runs out of memory (4GB) for larger documents and queries, whereas TASM-postorder does not. In summary, the performance of TASM-postorder compared to the special case of exact pattern matching is very encouraging.

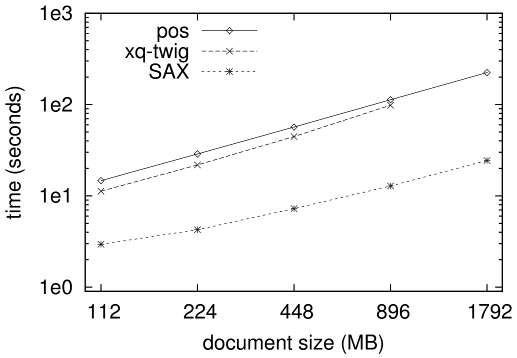


Fig. 14. Relative Performance of TASM-postorder as a Function of Document Size; $|Q| = 8$, $k = 5$.

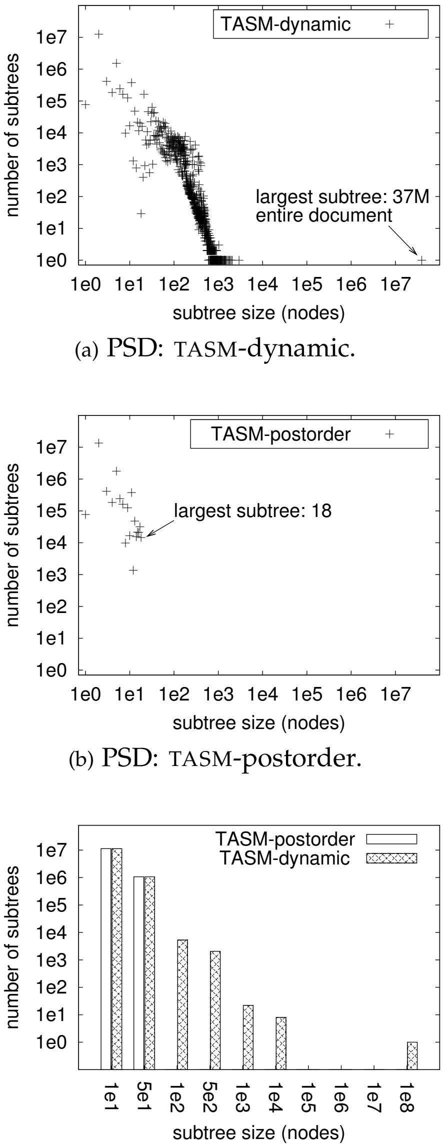
Observe that TASM and twig matching are very different query paradigms and the runtime comparison presented above only serves as a reference. The twig query is an explicit definition of the set of all possible query answers; if there is no exact match, the result set is empty. In TASM, the query is a single tree pattern; all subtrees of the document are ranked, and even if there is no exact match, TASM will return the k closest matches. TASM does not substitute twig queries, but complements them and allows users to ask queries when they do not have enough knowledge about possible answers to define a twig query.

7.3 Pruning of Search Space

In this section we evaluate the effectiveness of the prefix ring buffer pruning leveraged by TASM-postorder. Recall that the tree edit distance algorithm decomposes the input trees into relevant subtrees, and for each pair of relevant subtrees, Q_i and T_j, a matrix of size $|Q_i| \times |T_j|$ must be filled (see Section 4.6). The size and number of the relevant subtrees are the main factors for the computational complexity of the tree edit distance. TASM-dynamic incurs the maximum cost as it computes the distance between the query and every subtree in the document. In contrast, TASM-postorder prunes subtrees that are larger than a threshold.

Figure 15a shows the number of relevant subtrees (y-axis) of a specific size (x-axis) that TASM-dynamic must compute to find the top-1 ranking of the subtrees of the PSD7003 dataset for a query with $|Q| = 4$ nodes. Figure 15b shows the equivalent plot for TASM-postorder. The differences are significant: while TASM-dynamic computes the distance to all relevant subtrees, including the entire PSD document tree with

37M nodes, the largest subtree that is considered by TASM-postorder has only 18 nodes. Figure 15c shows a similar comparison for DBLP using a histogram. In the histogram, 1e1 shows the number of subtrees of sizes 0-9, 5e1 shows the sizes 10-49, 1e2 the sizes 50-99, etc. TASM-postorder computes much fewer and smaller trees: the bins for the subtree sizes 50 and larger are empty.



(a) PSD: TASM-dynamic.

(b) PSD: TASM-postorder.

(c) DBLP: TASM-postorder/TASM-dynamic.

Fig. 15. Number of Subtree Computations for PSD (scatter plots) and DBLP (histogram).

The subtrees computed by TASM-postorder are not always a subset of the subtrees computed by TASM-dynamic. If TASM-postorder prunes a large subtree, it may need to compute small subtrees of the pruned subtree that TASM-dynamic does not need to consider. Note, however, that every subtree that is computed by TASM-postorder is either computed by TASM-dynamic or contained in one that is. Thus TASM-dynamic is always more expensive. We define the *cumulative subtree size* which adds the sizes of the relevant subtrees up to a specific size x that are computed by a TASM algorithm: $\text{css}(x, T) = \sum_{i=1}^{x} i f_i, 1 \leq x \leq |T|$, where f_i is the number of subtrees of size i that are computed for document T. The difference of the cumulative subtree sizes of TASM-dynamic and TASM-postorder measures the extra computational effort for TASM-dynamic. In Figure 16 we show the cumulative subtree size difference, $css_{dyn}(x, T) - css_{pos}(x, T)$, over the subtree size x for answering a top-1 query on the documents DBLP and PSD. For small subtrees the curves are negative, which means that TASM-postorder computes more small trees than TASM-dynamic. Nevertheless, TASM-dynamic ends up performing a considerably larger computation task than TASM-postorder. TASM-dynamic processes around 27M (129M) nodes more than TASM-postorder for the DBLP (PSD) document (660K resp. 89M excluding the processing of the entire document by TASM-dynamic in its final step).

8 Conclusion

This paper discussed TASM: the problem of finding the top-k matches for a query Q in a document T w.r.t. the established tree edit distance metric [9]. This problem has applications in the integration and cleaning of heterogeneous XML repositories, as well as in answering similarity queries. We discussed the state-of-the-art solution that leverages the best dynamic programming algorithms for the tree edit distance and characterized its limitation in terms of memory requirements: namely, the need to compute and memorize the distance between the query and every subtree in the document. We proved an upper bound on the size of the largest subtree of the document that needs to be evaluated. This size depends on the query and the parameter k alone. We gave an effective pruning strategy that uses a prefix ring buffer and keeps only the necessary subtrees from the document in memory. As a result, we arrived at an algorithm that solves TASM in a single pass over the document and whose memory requirements are independent of the document itself. We verified our analysis experimentally and showed that our solution scales extremely well w.r.t. document size, query size, and the parameter k.

Our solution to TASM is portable. It relies on the postorder queue data structure which can be implemented by any XML processing or storage system that allows an efficient postorder traversal of trees. This is certainly the case for XML parsed from text files, for XML streams, and for XML stores based on variants of the interval encoding [24], which is prevalent among persistent XML stores. This work opens up the possibility of applying the established and well-understood tree edit distance in practical XML systems.

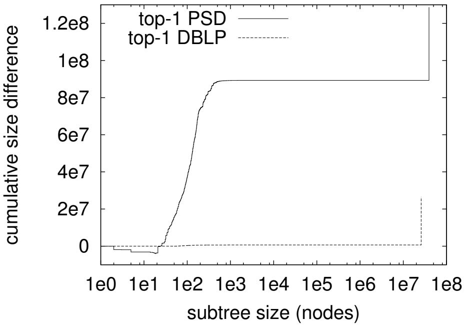


Fig. 16. Cumulative Subtree Size Difference for Computing Top-1 Queries.

Acknowledgments. This work was funded in part by the BIT Joint School for Information Technology, FP7 EU IP OKKAM (contract no. ICT-215032, http://www.okkam.org), NSERC, and AIF.

REFERENCES

[1] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu, "Approximate XML joins," in *SIGMOD*, 2002, pp. 287–298.

[2] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity flooding: A versatile graph matching algorithm and its application to schema matching," in *ICDE*, 2002, pp. 117–128.

[3] N. Augsten, M. H. Böhlen, C. E. Dyreson, and J. Gamper, "Approximate joins for data-centric XML," in *ICDE*, 2008, pp. 814–823.

[4] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching." *VLDB J.*, vol. 10, no. 4, pp. 334–350, 2001.

[5] M. Weis and F. Naumann, "Dogmatix tracks down duplicates in XML," in *SIGMOD*, 2005, pp. 431–442.

[6] N. Agarwal, M. G. Oliveras, and Y. Chen, "Approximate structural matching over ordered XML documents," in *IDEAS*, 2007, pp. 54–62.

[7] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRANK: Ranked keyword search over XML documents," in *SIGMOD*, 2003, pp. 16–27.

[8] K.-C. Tai, "The tree-to-tree correction problem," *J. of the ACM*, vol. 26, no. 3, pp. 422–433, 1979.

[9] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM J. on Computing*, vol. 18, no. 6, pp. 1245–1262, 1989.

[10] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Computing Surveys*, vol. 40, no. 4, 2008.

[11] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman, "Structure and content scoring for XML," in *VLDB*, 2005, pp. 361–372.

[12] A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava, "Adaptive processing of top-k queries in XML," in *ICDE*, 2005, pp. 162–173.

[13] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum, "TopX: Efficient and versatile top-k query processing for semistructured data," *VLDB J.*, vol. 17, no. 1, pp. 81–115, 2008.

[14] M. S. Ali, M. P. Consens, X. Gu, Y. Kanza, F. Rizzolo, and R. K. Stasiu, "Efficient, effective and flexible XML retrieval using summaries," in *INEX*, 2006, pp. 89–103.

[15] Z. Liu and Y. Chen, "Identifying meaningful return information for XML keyword search," in *SIGMOD*, 2007, pp. 329–340.

[16] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan, "On the integration of structure indexes and inverted lists," in *SIGMOD*, 2004, pp. 779–790.

[17] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *J. of Computer and System Sciences*, vol. 66, no. 4, pp. 614–656, 2003.

[18] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann, "An optimal decomposition algorithm for tree edit distance," in *ICALP*, ser. LNCS, vol. 4596. Springer, 2007, pp. 146–157.

[19] D. Barbosa, L. Mignet, and P. Veltri, "Studying the XML Web: Gathering statistics from an XML sample," *World Wide Web J.*, vol. 8, no. 4, pp. 413–438, 2005.

[20] R. Yang, P. Kalnis, and A. K. H. Tung, "Similarity evaluation on tree-structured data," in *SIGMOD*, 2005, pp. 754–765.

[21] N. Augsten, M. Böhlen, and J. Gamper, "The pq-gram distance between ordered labeled trees," *ACM Transactions on Database Systems*, vol. 35, no. 1, 2010.

[22] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. of the ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[23] Y. Tian and J. M. Patel, "TALE: A tool for approximate large graph matching," in *ICDE*, 2008, pp. 963–972.

[24] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang, "Storing and querying ordered XML using a relational database system," in *SIGMOD*, 2002, pp. 204–215.

[25] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "XMark: A benchmark for XML data management," in *VLDB*, 2002, pp. 974–985.

[26] M. Kay, "Ten reasons why saxon xquery is fast," *IEEE Data Eng. Bull.*, vol. 31, no. 4, pp. 65–74, 2008.

Nikolaus Augsten received his PhD degree in Computer Science from Aalborg University, Denmark, in 2008. He is currently an Assistant Professor in the Database and Information Systems Group, Free University of Bozen-Bolzano, Italy. His research interests include data-centric applications in database and information systems with a particular focus on approximate matching techniques for complex data structures, efficient index structures for distance computations, and similarity search in massive data collections.

Denilson Barbosa received a PhD degree in Computer Science from the University of Toronto, Canada, while working on storing, updating and incremental constraint checking of XML data. He received an IBM Faculty Award for his work on XML benchmarking, and an Alberta Ingenuity New Faculty Award for his work on extraction and integration of data from the Web. He is currently an Assistant Professor of Computing Science at the University of Alberta, Canada.

Michael H. Böhlen received his M.Sc. and Ph.D. degrees from ETH Zürich in respectively 1990 and 1994. He is a professor of computer science at the University of Zürich where he heads the database technology group. Before joining the University of Zürich he was a faculty member at Aalborg University and the Free University of Bozen-Bolzano. His research interests include various aspects of data management, and have focused on time-varying information, data warehousing, similarity search in hierarchical data, and data analysis. With his colleagues, he has published widely in the main database outlets. He served as a PC member for many international conferences (including SIGMOD, VLDB, ICDE and EDBT). Currently, he serves as an associate editor for the VLDB Journal and ACM TODS, and he is a member of the VLDB Endowment's Board of Trustees. He is a member of ACM and IEEE.

Themis Palpanas received the BS degree from the National Technical University of Athens, Greece, and the MSc and PhD degrees from the University of Toronto, Canada. He is a faculty member of computer science at the University of Trento, Italy. Before joining the University of Trento, he worked at the IBM T.J. Watson Research Center and the University of California, Riverside, and visited Microsoft Research and the IBM Almaden Research Center. His interests include data management, data analysis, and streaming algorithms. He is the author of five US patents, and is serving on the program committees of several top database and data mining conferences.