



US 20120254251A1

(19) **United States**

(12) **Patent Application Publication**  
**Barbosa et al.**

(10) **Pub. No.: US 2012/0254251 A1**

(43) **Pub. Date: Oct. 4, 2012**

(54) **SYSTEMS AND METHODS FOR EFFICIENT  
TOP-K APPROXIMATE SUBTREE  
MATCHING**

(30) **Foreign Application Priority Data**

Mar. 3, 2011 (CA) ..... 2733311

(75) Inventors: **Denilson Barbosa**, Edmonton  
(CA); **Nikolaus Augsten**,  
Bozen-Bolzano (IT); **Michael  
Böhlen**, Zurich (CH); **Themis  
Palpanas**, Trento (IT)

**Publication Classification**

(51) **Int. Cl.**  
**G06F 17/30** (2006.01)

(52) **U.S. Cl.** ..... **707/797; 707/E17.012**

(73) Assignee: **THE GOVERNORS OF THE  
UNIVERSITY OF ALBERTA**,  
Edmonton (CA)

(57) **ABSTRACT**

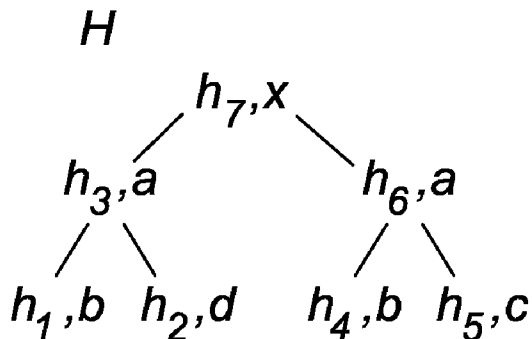
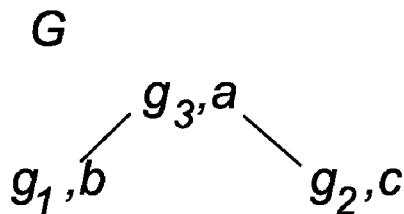
Systems and method for searching for approximate matches in a database of documents represented by a tree structure. A fast solution to the Top-k Approximate Subtree Matching Problem involves determining candidate subtrees which will be considered as possible matches to a query also represented by a tree structure. Once these candidate subtrees are found, a tree edit distance between each candidate subtree and the query tree is calculated. The results are then sorted to find those with the lowest tree edit distance.

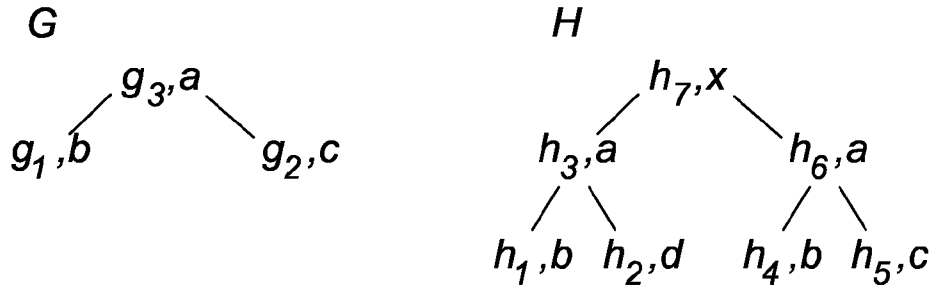
(21) Appl. No.: **13/411,494**

(22) Filed: **Mar. 2, 2012**

**Related U.S. Application Data**

(60) Provisional application No. 61/448,996, filed on Mar. 3, 2011.





**FIG. 1**

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

**FIG. 2**

---

**Algorithm 1:** TASM-dynamic( $Q, T, k, \mathbb{R}$ )

---

**Input:** query  $Q$ , document  $T$ , number of matches  $k$ ,  
 (possibly empty) ranking  $\mathbb{R}$ ,  $|\mathbb{R}| \leq k$   
**Output:** top- $k$  ranking of the subtrees of  $T$  w.r.t.  $Q$   
 merged with the input ranking  $\mathbb{R}$

```

1 begin
2   td : empty  $|Q| \times |T|$  matrix;
3   pd : empty  $(|Q| + 1) \times (|T| + 1)$  matrix;
4   foreach relevant subtree  $Q_m$  of  $Q$  (ascending  $m$ ) do
5     foreach relevant subtree  $T_n$  of  $T$  (ascending  $n$ ) do
6       pd[0, 0]  $\leftarrow$  0;
7       foreach  $t_j \in V(T_n)$  (ascending) do
8         pd[0,  $t_j$ ]  $\leftarrow$  pd[0,  $t_{j-1}$ ] +  $\gamma(\epsilon, t_j)$ ;
9         foreach  $q_i \in V(Q_m)$  (ascending) do
10          pd[ $q_i$ , 0]  $\leftarrow$  pd[ $q_{i-1}$ , 0] +  $\gamma(q_i, \epsilon)$ ;
11          if  $Q_i = \text{pix}(Q_m, q_i) \wedge T_j = \text{pix}(T_n, t_j)$ 
12          then
13            pd[ $q_i, t_j$ ]  $\leftarrow$  min(
14              pd[ $q_{i-1}, t_j$ ] +  $\gamma(q_i, \epsilon)$ ,
15              pd[ $q_i, t_{j-1}$ ] +  $\gamma(\epsilon, t_j)$ ,
16              pd[ $q_{i-1}, t_{j-1}$ ] +  $\gamma(q_i, t_j)$ );
17            td[ $Q_i, T_j$ ]  $\leftarrow$  pd[ $q_i, t_j$ ];
18          else
19            pd[ $q_i, t_j$ ]  $\leftarrow$  min(
20              pd[ $q_{i-1}, t_j$ ] +  $\gamma(q_i, \epsilon)$ ,
21              pd[ $q_i, t_{j-1}$ ] +  $\gamma(\epsilon, t_j)$ ,
22              pd[ $q_{i-1}, t_{j-1}$ ] + td[ $Q_i, T_j$ ]);
23          end
24        end
25      end
26      if  $Q_m = Q \wedge T_n = T$  then
27         $\mathbb{R} \leftarrow \text{push-heap}(\mathbb{R}, (\text{td}[Q_m, T_j], T_j))$ ;
28        if  $|\mathbb{R}| > k$  then  $\mathbb{R} \leftarrow \text{pop-heap}(\mathbb{R})$ ;
29      end
30    end
31  end
32  return  $\mathbb{R}$ ;
33 end
    
```

---

**FIG. 2A**

---

**Algorithm 2:** prb-pruning(pq,  $\tau$ )

---

**Input:** postorder queue pq of a document  $T$ , threshold  $\tau$

**Output:** candidate set  $cand(T, \tau)$

```

1 begin
2   pfx, lbl: ring buffers of size  $b = \tau + 1$ ;
3    $C \leftarrow \emptyset$ ;
4   (pfx, lbl, s, e, c, pq)  $\leftarrow$  prb-next(pfx, lbl, 1, 1, 0, pq,  $\tau$ );
5   while  $s \neq e$  do
6      $C \leftarrow C \cup \{\text{prb-subtree}(\text{pfx}, \text{lbl}, s, \text{pfx}[s])\}$ ;
7      $s \leftarrow (\text{pfx}[s] + 1) \text{ \% } b$ ;
8     (pfx, lbl, s, e, c, pq)  $\leftarrow$ 
9     prb-next(pfx, lbl, s, e, c, pq,  $\tau$ );
10  end
11 return  $C$ ;
12 end

```

---

**FIG. 2B**

---

**Algorithm 3:** prb-next(*ptx*, *lbl*, *s*, *e*, *c*, *pq*,  $\tau$ )

---

**Input:** ring buffers *ptx* 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*,  $\in \text{cand}(T, \tau)$

```

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       ( $\lambda$ , size)  $\leftarrow$  dequeue(pq);
6        $lbl[e] \leftarrow \lambda$ ;
7        $ptx[e] \leftarrow (++c) - \text{size} + 1$ ;
8       if  $\text{size} \leq \tau$  then
9          $ptx[ptx[e] \% b] \leftarrow c$ ;
10        end
11         $e \leftarrow (e + 1) \% b$ ;
12      end
13      if  $s = (e + 1) \% b$  or  $pq = \emptyset$  then
14        if  $ptx[s] \geq c + 1 - (e - s + b) \% b$  then
15          return (ptx, lbl, s, e, c, pq);
16        else
17           $s \leftarrow (s + 1) \% b$ ;
18        end
19      end
20    end
21    return (ptx, lbl, s, e, c, pq);
22 end

```

---

**FIG. 2C**

---

**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  $s \neq \epsilon$  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] % b, r % b);
10      if  $|R| = k$  then  $\tau' = \min(\tau, \max(R) + |Q|)$ ;
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
    
```

---

**FIG. 2D**

---

**Algorithm 5: TASM-dynamic<sup>+</sup>( $Q, T, k, \mathbb{R}$ )**


---

**Input:** query  $Q$ , document  $T$ , number of matches  $k$ ,  
 (possibly empty) ranking  $\mathbb{R}$ ,  $|\mathbb{R}| \leq k$

**Output:** top- $k$  ranking of the subtrees of  $T$  w.r.t.  $Q$   
 merged with the input ranking  $\mathbb{R}$

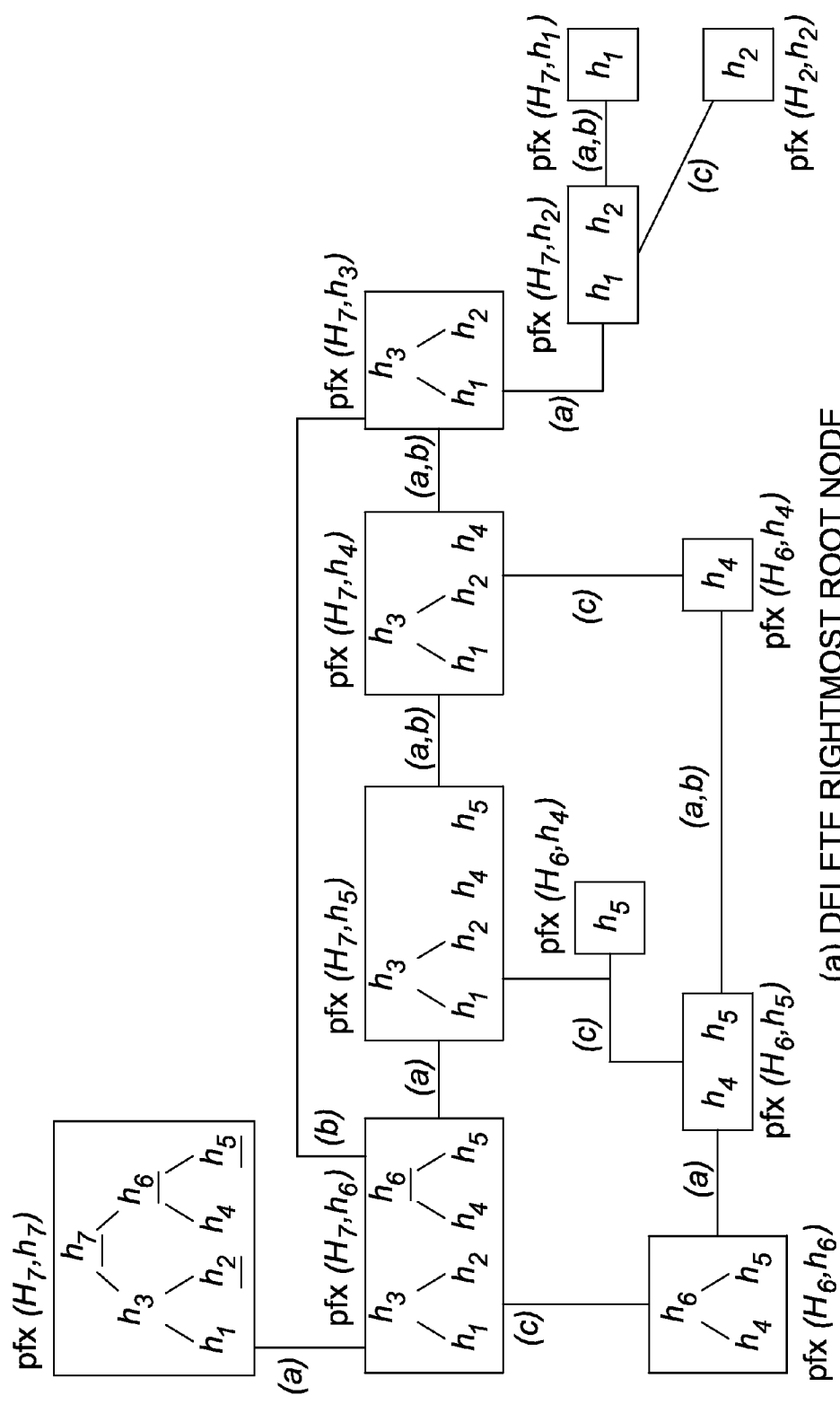
```

1 begin
  ...
7  foreach  $t_j \in V(T_n)$  (ascending) do
    if  $|\mathbb{R}| = k \wedge |\text{pfx}(T_n, t_j)| > \max(\mathbb{R}) + |Q|$  then
      | goto line 28; // exit inner loop
    end
8     $\text{pd}[\emptyset, t_j] \leftarrow \text{pd}[\emptyset, t_{j-1}] + \gamma(e, t_j)$ 
    ...
10   end
  ...
11   return  $\mathbb{R}$ 
12 end

```

---

**FIG. 2E**



- (a) DELETE RIGHTMOST ROOT NODE
- (b) DELETE RIGHTMOST TREE
- (c) KEEP ONLY RIGHTMOST TREE

**FIG. 3**



PREFIX DISTANCE MATRIX pd (ONLY RELEVANT PARTS ARE SHOWN):

$G_2, H_2$		$G_2, H_5$		$G_2, H_4$		$G_2, H_7$	
$t_f \rightarrow h_2$	$q_i \searrow$	$t_f \rightarrow h_5$	$q_i \searrow$	$t_f \rightarrow h_4, h_5, h_0$	$q_i \searrow$	$t_f \rightarrow h_1, h_2, h_3, h_4, h_5, h_6, h_7$	$q_i \searrow$
0 1	0 1	0 1 2 3	0 1 2 3	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7		
1 1	1 0	1 1 1 2	1 1 2 3 4 5 6	1 1 2 3 4 4 5 6	1 1 2 3 4 4 5 6		
$g_2$	$g_2$	$g_2$	$g_2$	$g_2$	$g_2$		

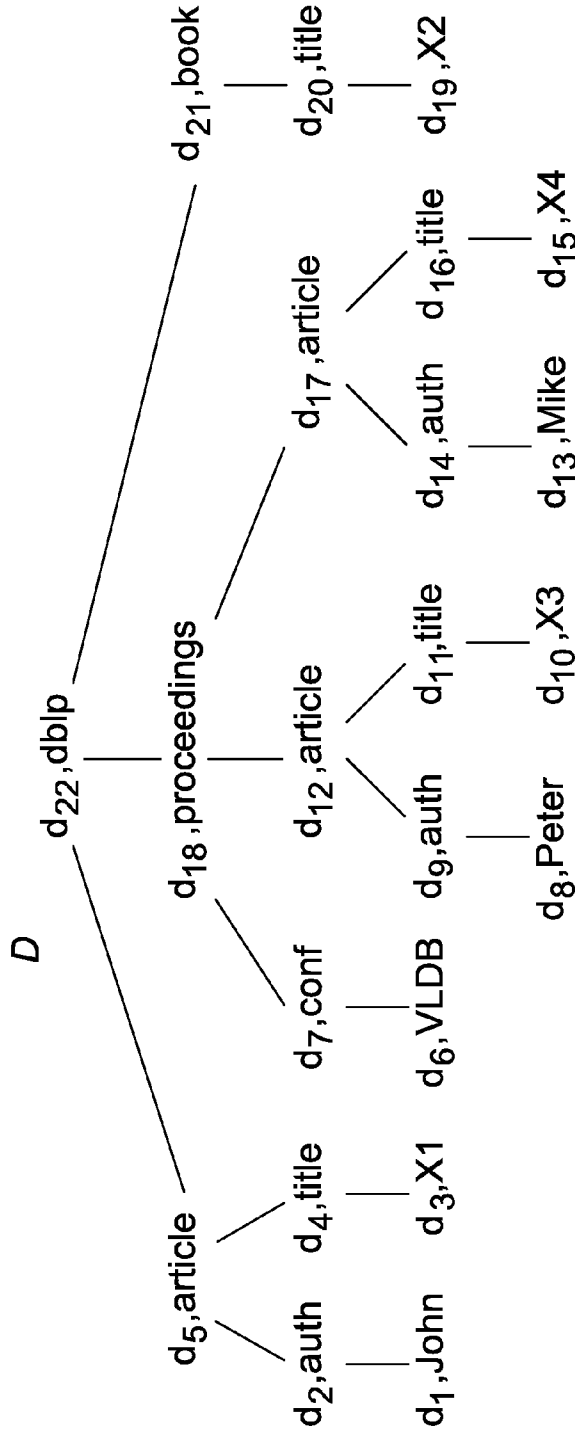
$G_a, H_2$		$G_a, H_5$		$G_a, H_6$		$G_a, H_7$	
$t_f \rightarrow h_2$	$q_i \searrow$	$t_f \rightarrow h_5$	$q_i \searrow$	$t_f \rightarrow h_4, h_5, h_0$	$q_i \searrow$	$t_f \rightarrow h_1, h_2, h_3, h_4, h_5, h_6, h_7$	$q_i \searrow$
0 1	0 1	0 1 2 3	0 1 2 3	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7		
1 1	1 1	1 0 1 2	1 0 1 2	1 0 1 2 3 4 5 6	1 0 1 2 3 4 5 6		
2 2	2 1	2 1 0 1	2 1 0 1	2 1 1 2 3 4 5	2 1 1 2 3 3 4 5		
3 3	3 2	3 2 1 0	3 2 1 0	3 2 1 2 3 3 4	3 2 2 1 2 3 3 4		
$g_1$	$g_1$	$g_1$	$g_1$	$g_1$	$g_1$		
$g_2$	$g_2$	$g_2$	$g_2$	$g_2$	$g_2$		
$g_3$	$g_3$	$g_3$	$g_3$	$g_3$	$g_3$		

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. 4

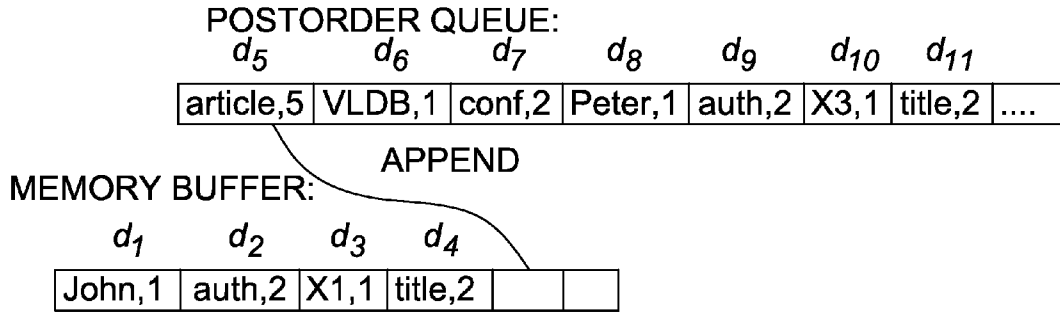


(a) EXAMPLE DOCUMENT D

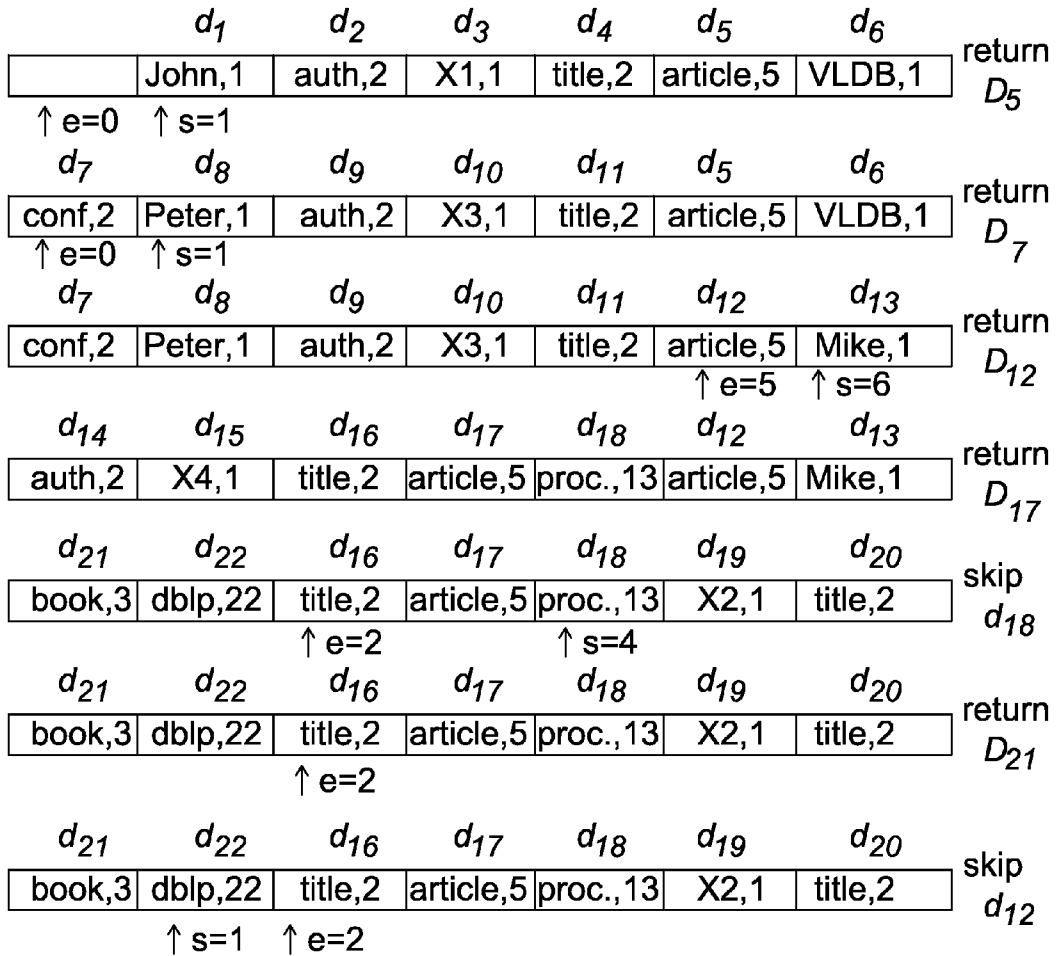
$post(D) = ((John, 1), (auth, 2), (X1, 1), (title, 2), (article, 5), (VLDB, 1), (conf, 2), (Peter, 1), (auth, 2), (X3, 1), (title, 2), (article, 5), (Mike, 1), (auth, 2), (X4, 1), (title, 2), (article, 5), (proceedings, 13), (X2, 1), (title, 2), (book, 3), (dblp, 22))$

(b) POSTORDER QUEUE OF D

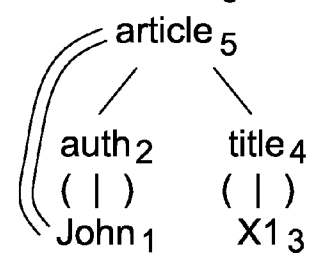
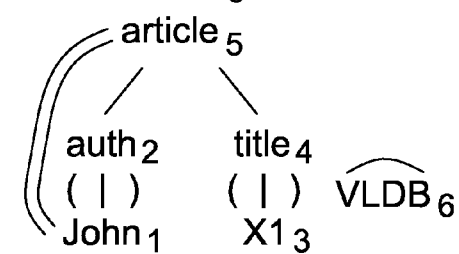
**FIG. 5**



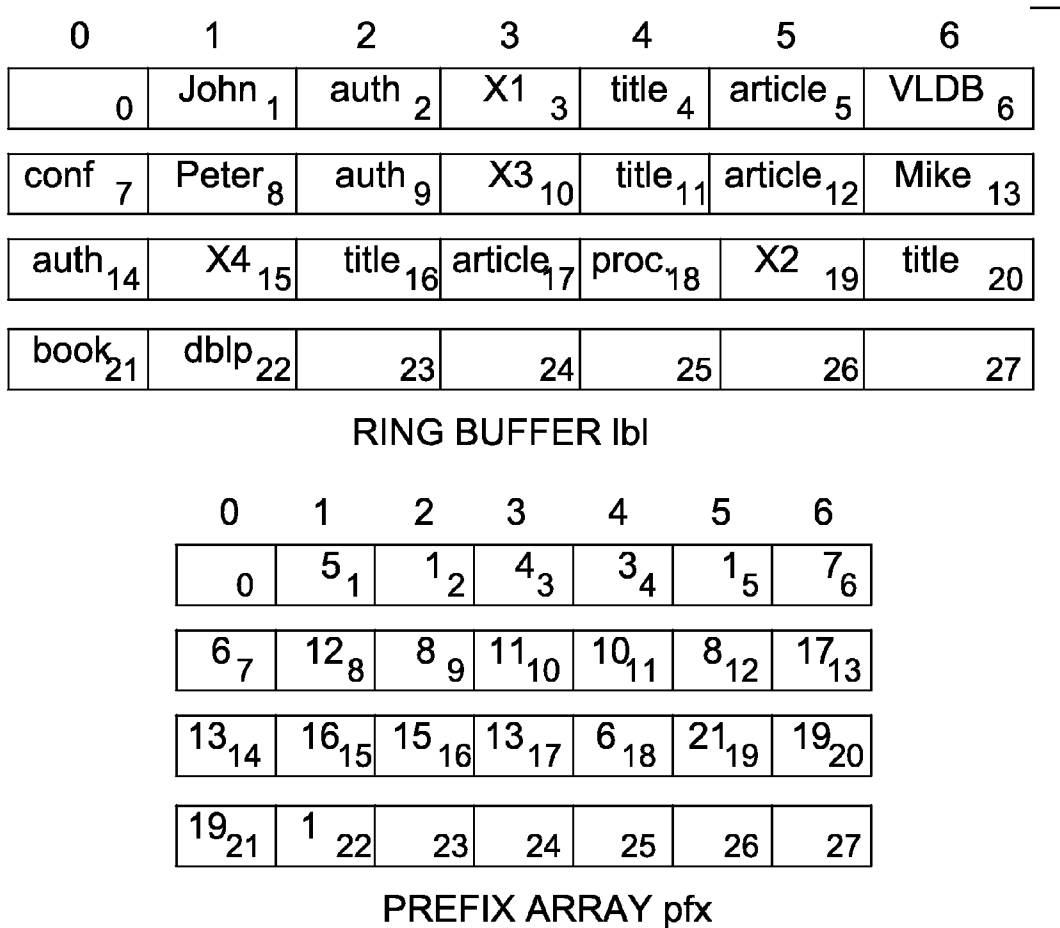
**FIG. 6**



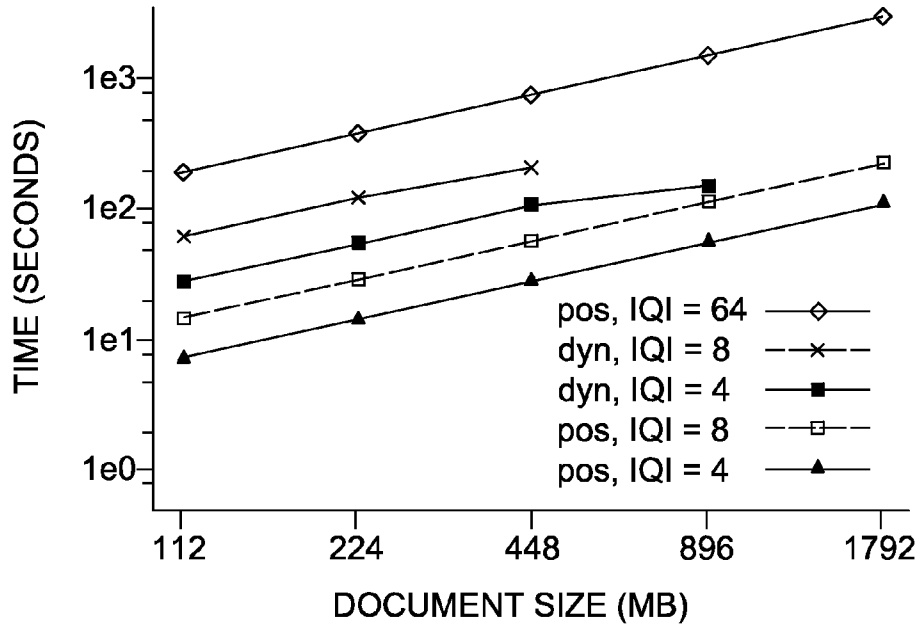
**FIG. 7**

$\text{pfx}(D, d_4) :$  $\text{auth}_2$ $\text{title}_4$ $( \quad )$ $( \quad )$ $\text{John}_1$ $\text{X1}_3$	$\text{pfx}(D, d_5) :$ 	$\text{pfx}(D, d_6) :$ 
Prefix Array: (2,1,4,3)	Prefix Array: (5,1,4,3,1)	Prefix Array: (5,1,4,3,1,6)

**FIG. 8**

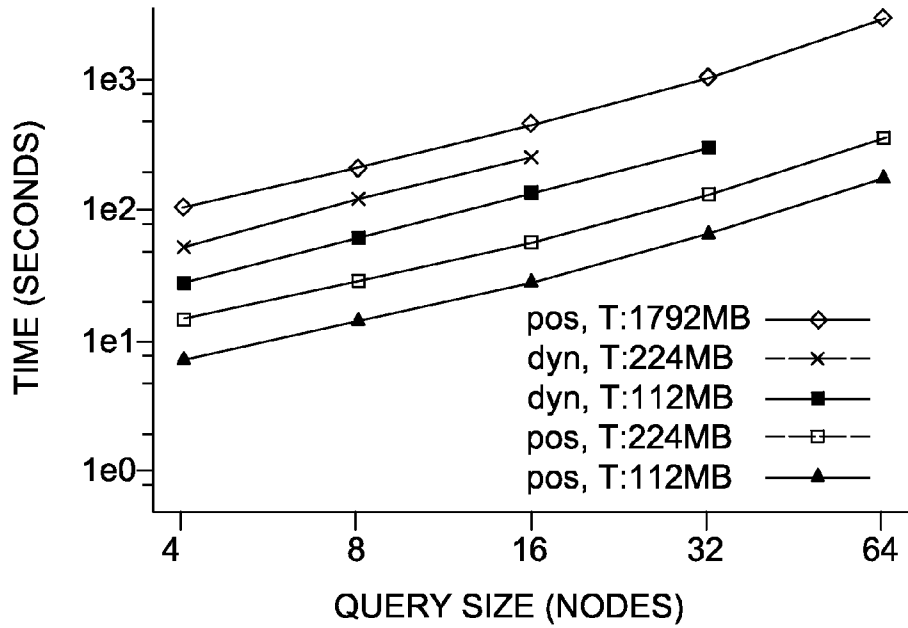


**FIG. 9**



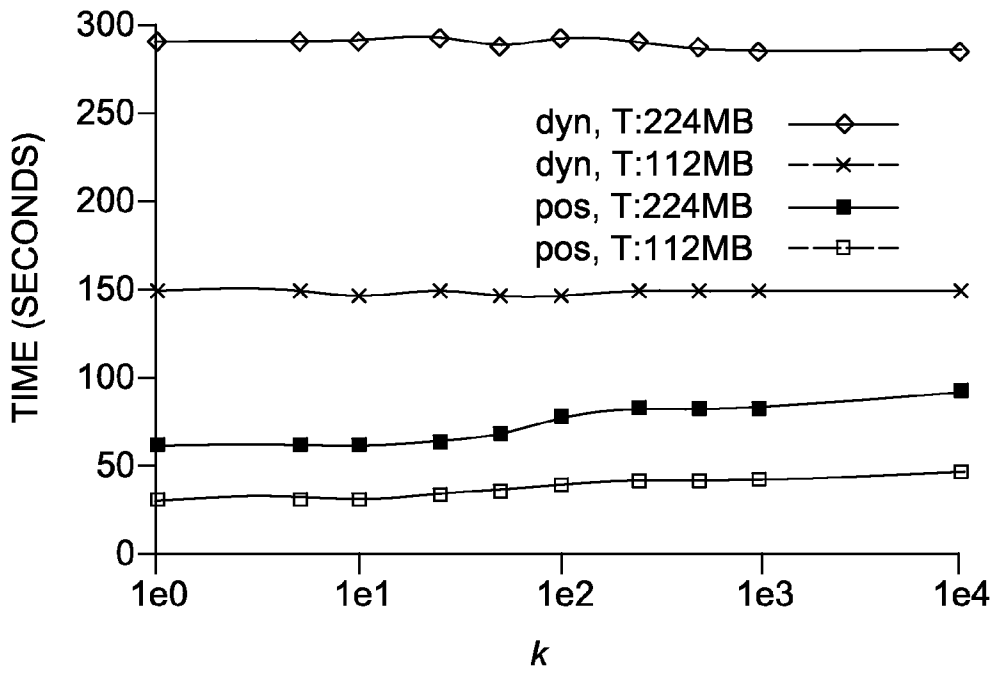
(a) VARYING DOCUMENT SIZE; k = 5

### FIG. 10A



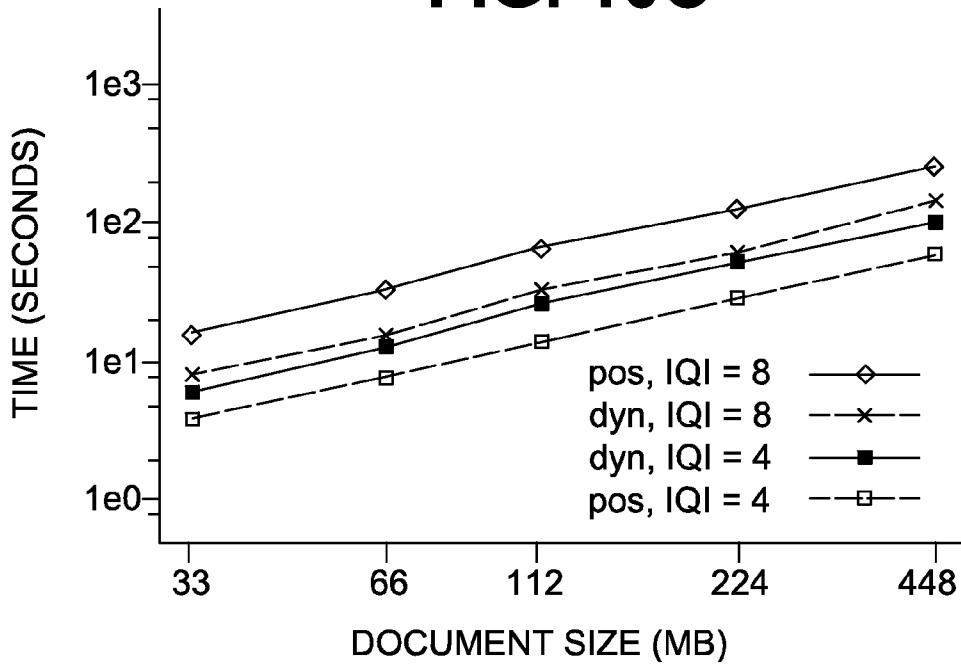
(a) VARYING QUERY SIZE; k = 5

### FIG. 10B

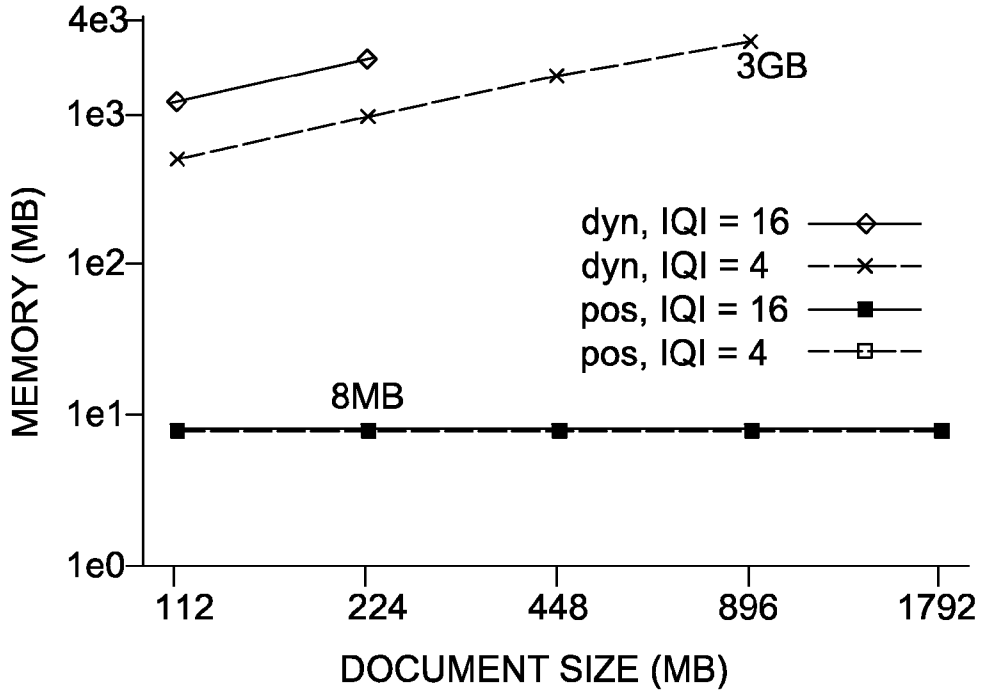


(c) VARYING  $k$ ;  $|Q| = 16$

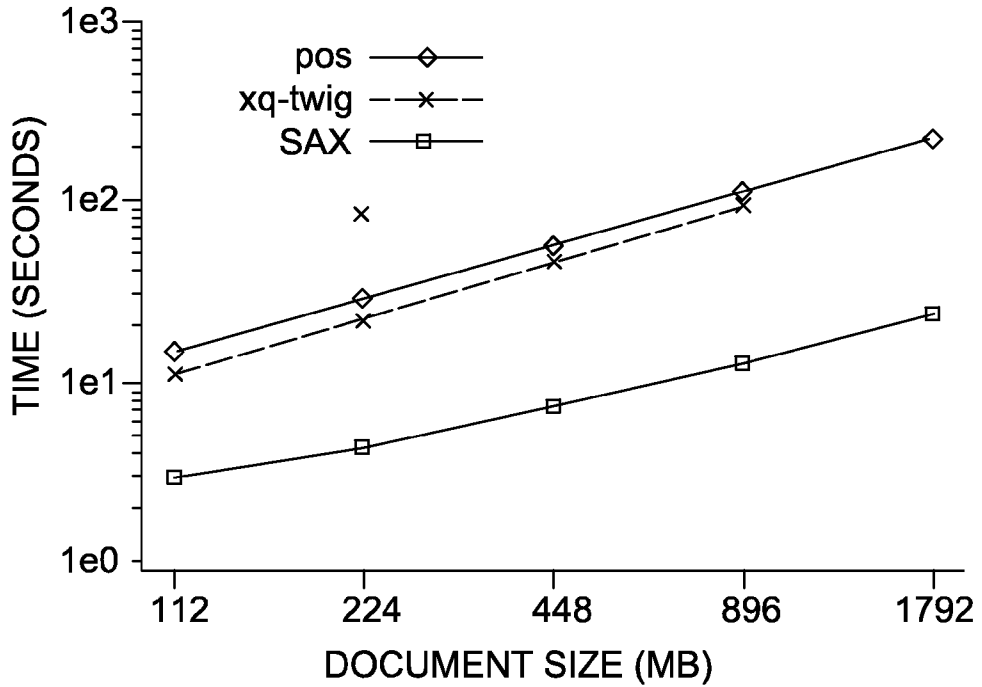
**FIG. 10C**



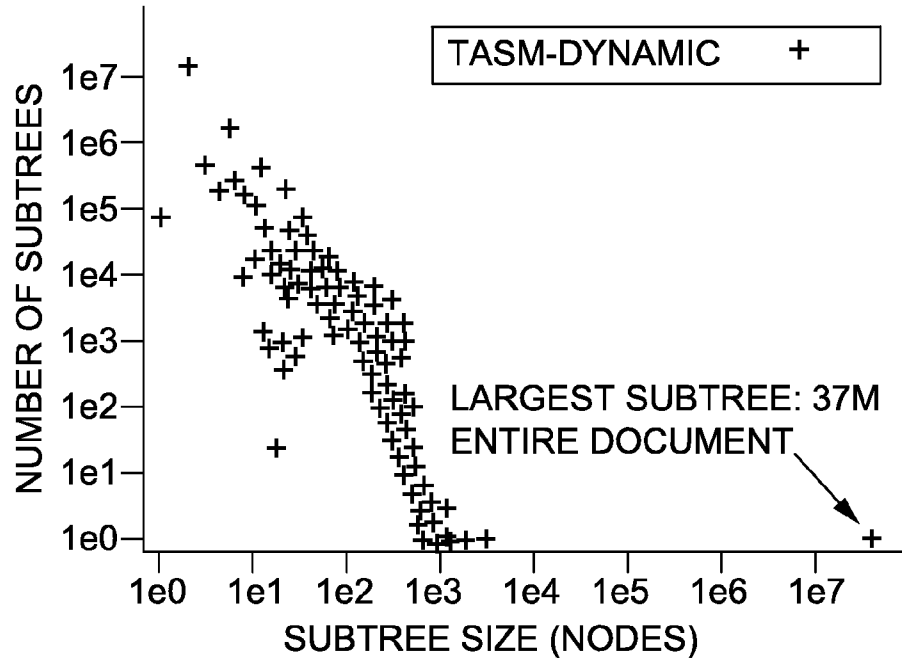
**FIG. 11**



**FIG. 12**

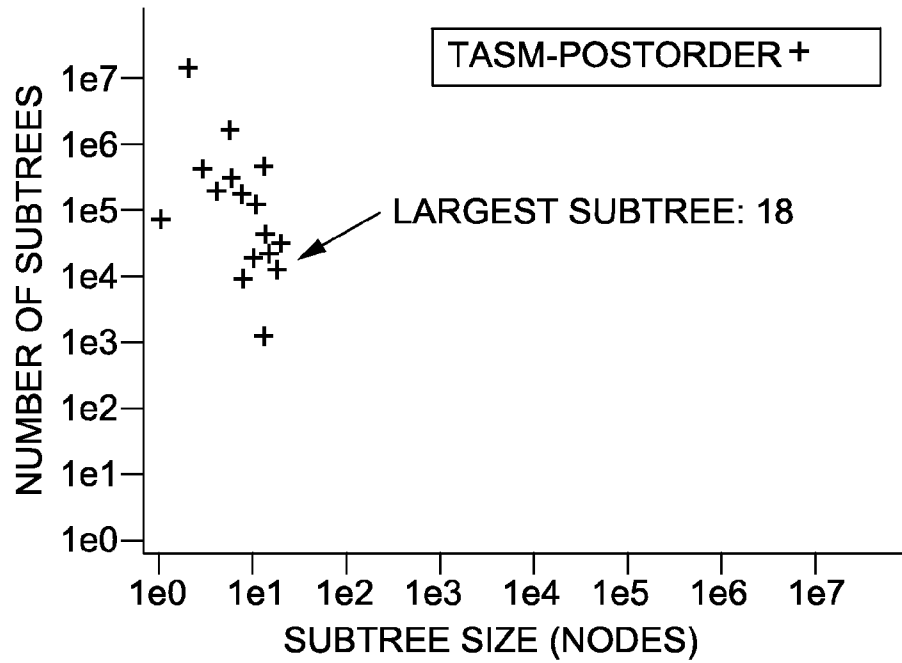


**FIG. 13**



(a) PSD: TASM-DYNAMIC

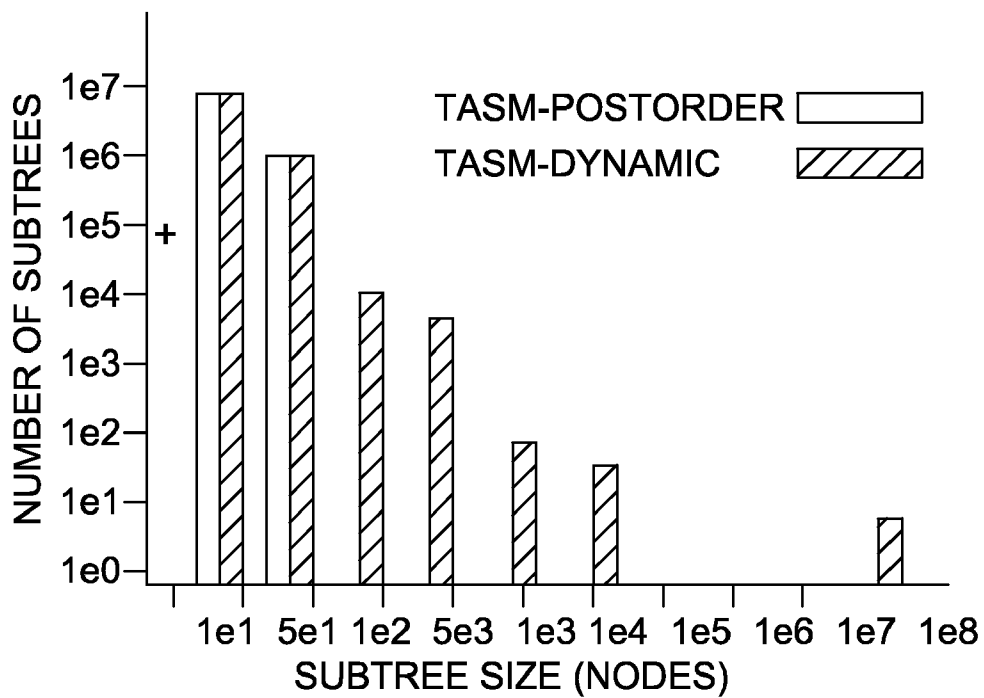
### FIG. 14A



(b) PSD: TASM-POSTORDER

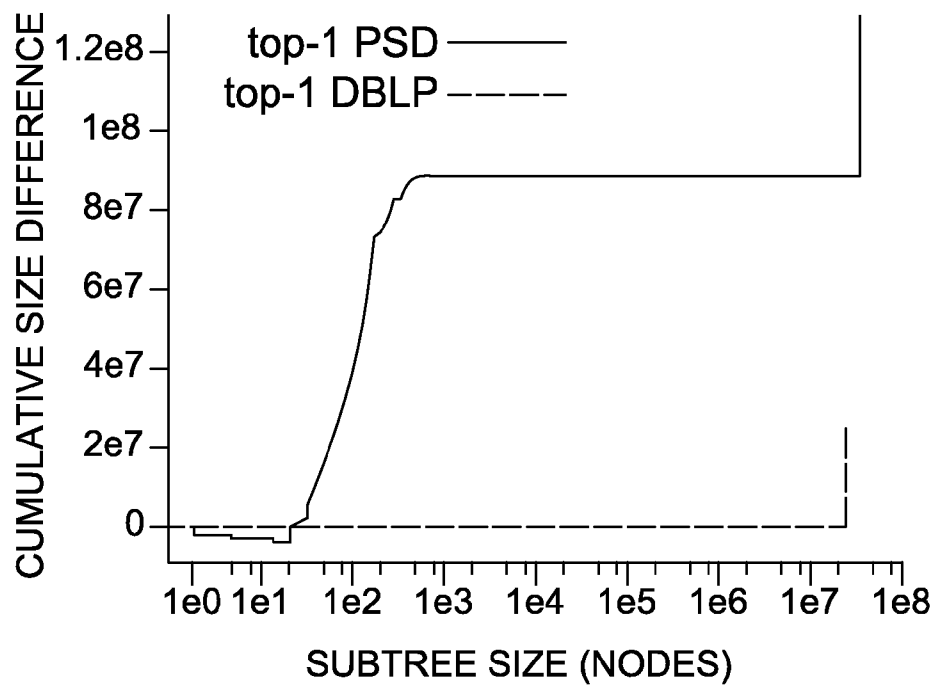
### FIG. 14B



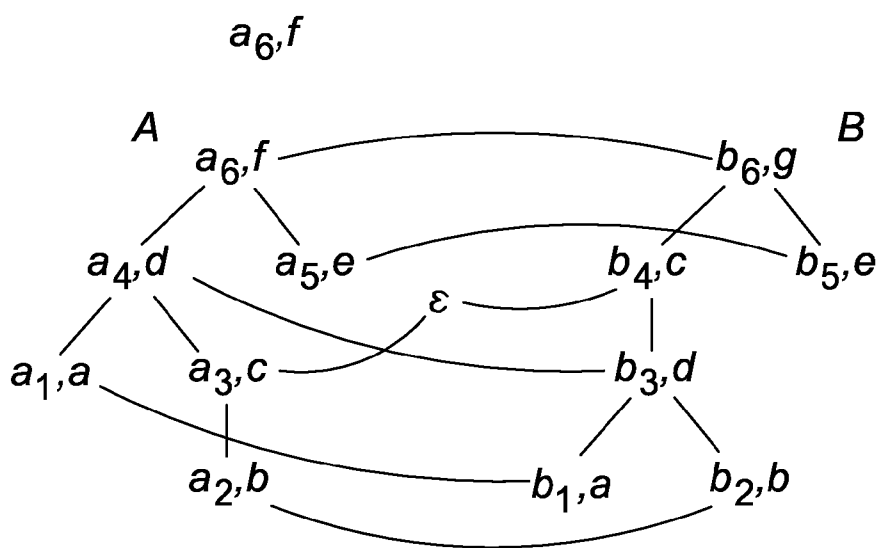


(c) DBLP: TASM-POSTORDER/TASM-DYNAMIC

**FIG. 14C**



**FIG. 15**



**FIG. 16**

**SYSTEMS AND METHODS FOR EFFICIENT  
TOP-K APPROXIMATE SUBTREE  
MATCHING**

TECHNICAL FIELD

[0001] The present invention relates to computer-based searching of databases. More specifically, the present invention relates to a tree-based searching method for finding a set of closest approximations in a database to a query.

BACKGROUND OF THE INVENTION

[0002] 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, cleaning such integrated data, as well as for answering similarity queries. For these applications, the issue is the so-called 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, what is sought is a ranking (T<sub>i</sub>1, T<sub>i</sub>2, . . . , T<sub>i</sub>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.

[0003] 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<sup>2</sup>n<sup>2</sup>) time and O(mn) space. An O(n) improvement in time leverages the dynamic programming formulation of tree edit distance algorithms: compute the distance between Q and T, and rank all subtrees of by visiting the resulting memorization table. Still, for large documents with millions of nodes, the O(mn) space complexity is prohibitive.

[0004] Answering top K queries is an active research field. Specific to XML, many authors have studied the ranking of answers to twig queries, 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 (respectively, approximate answers) to a twig query are subtrees of the document that satisfy (respectively, 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.

[0005] The goal of XML keyword search 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. study TA-style 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.

[0006] XFinder 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, no runtime analysis is given and the experiments reported use documents of up to 5 MB.

[0007] For ordered trees like XML the problem of computing the similarity between the query and the subtrees of the document can be solved with elegant dynamic programming formulations. Zhang and Shasha present an O(n<sup>2</sup> log<sup>2</sup>n) time and O(n<sup>2</sup>) space algorithm for trees with n nodes and height O(logn). Their worst case complexity is O(n<sup>4</sup>). Demaine et al. use a different tree decomposition strategy to improved the time complexity to O(n<sup>3</sup>) in the worst case. This is not a concern in practice since XML documents tend to be shallow and wide.

[0008] Guha et al. 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<sup>2</sup>) time as a pruning strategy to avoid comparing all pairs of trees from the repositories. Yang et al. and Augsten et al. provide lower bounds for the tree edit distance that can be computed in O(nlogn) time.

[0009] Approximate substructure matching has also been studied in the context of graphs. TALE 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. 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.

[0010] Based on the above, there is therefore a need for systems and methods that can provide a solution to the TASM issue or which can, at the very least, mitigate the problems with the prior art as noted above.

SUMMARY OF INVENTION

[0011] The present invention provides systems and method for searching for approximate matches in a database of documents represented by a tree structure. A fast solution to the Top-k Approximate Subtree Matching Problem involves determining candidate subtrees which will be considered as possible matches to a query also represented by a tree structure. Once these candidate subtrees are found, a tree edit distance between each candidate subtree and the query tree is calculated. The results are then sorted to find those with the lowest tree edit distance.

[0012] In a first aspect, the present invention provides a method for sorting nodes in a document tree to determine a number of closest approximations to a query represented by a query tree, the method comprising:

- [0013] a) determining a limit size of subtrees of said document tree to be considered;
- [0014] b) determining candidate subtrees of said document tree, each candidate subtree of said document tree having a size equal to or less than said limit size and each candidate subtree is not a subtree of another subtree having a size less than or equal to said limit size;

- [0015] c) for each candidate subtree, determining a tree edit distance between said candidate subtree and said query tree;
  - [0016] d) sorting candidate subtrees in accordance with their respective tree edit distances with said query tree, in order to determine which candidate subtrees have least tree edit distances with said query tree;
- wherein said tree edit distance is a cost to convert contents of one subtree into contents of a second subtree.
- [0017] In a second aspect, the present invention provides computer-readable media having encoded thereon computer readable and computer executable instructions which, when executed, executes a method for sorting nodes in a document tree to determine a number of closest approximations to a query represented by a query tree, the method comprising:
- [0018] a) determining a limit size of subtrees of said document tree to be considered;
  - [0019] b) determining candidate subtrees of said document tree, each candidate subtree of said document tree having a size equal to or less than said limit size and each candidate subtree is not a subtree of another subtree having a size less than or equal to said limit size;
  - [0020] c) for each candidate subtree, determining a tree edit distance between said candidate subtree and said query tree;
  - [0021] d) sorting candidate subtrees in accordance with their respective tree edit distances with said query tree, in order to determine which candidate subtrees have least tree edit distances with said query tree;
- wherein said tree edit distance is a cost to convert contents of one subtree into contents of a second subtree.
- [0022] In yet another aspect, the present invention provides a method for determining which subtrees in a document tree most closely approximate a given query tree, the method comprising:
- [0023] a) determining a limit size of subtrees of said document tree to be considered;
  - [0024] b) determining candidate subtrees of said document tree, each candidate subtree of said document tree being, at most, equal in size to said limit size,
  - [0025] c) for each candidate subtree, determining a cost to convert contents of said candidate subtree into contents of said query tree;
  - [0026] d) sorting candidate subtrees in accordance with costs for converting said candidate subtrees into said query tree,
  - [0027] e) determining which candidate subtrees have lowest costs for converting said candidate subtrees into said query tree, candidate subtrees having lowest costs for being converted into said query tree being subtrees which most closely approximate said query tree.

BRIEF DESCRIPTION OF THE DRAWINGS

- [0028] The embodiments of the present invention will now be described by reference to the following figures, in which identical reference numerals in different figures indicate identical elements and in which:
- [0029] FIG. 1 illustrates an example query tree G and a document tree H;
- [0030] FIG. 2 lists decomposition rules for calculating tree edit distance;
- [0031] FIGS. 2A-2E show the different algorithms used in the invention;

- [0032] FIG. 3 illustrates an example of decomposing the document tree H in FIG. 1 into prefixes;
- [0033] FIG. 4 illustrates a calculation of tree edit distances using the rules in FIG. 2 and the query tree G and document tree H;
- [0034] FIGS. 5a and 5b illustrates an example document tree D and its corresponding postorder queue;
- [0035] FIG. 6 shows how incoming nodes are appended to the memory buffer;
- [0036] FIG. 7 illustrates a ring buffer as it is pruned of subtrees;
- [0037] FIG. 8 shows the prefix arrays of three prefixes derived from the document tree D in FIG. 5a;
- [0038] FIG. 9 illustrates an implementation of the prefix ring buffer;
- [0039] FIGS. 10a, 10b, and 10c illustrate execution times for varying sizes of documents, queries, and k;
- [0040] FIG. 11 illustrates a graph comparing the execution times for TASM-dynamic+ and TASM-dynamic for k=5;
- [0041] FIG. 12 is a graph illustrating memory usage as a function of document size for k=5;
- [0042] FIG. 13 is a graph showing relative performance of TASM-postorder as a function of document size for l=8 and k=5
- [0043] FIGS. 14a, 14b, and 14c are plots showing a comparison of the number of subtrees that various methods have to calculate to find the top-1 ranking of subtrees for a specifically sized query;
- [0044] FIG. 15 illustrates cumulative subtree size difference for computing top-1 queries; and
- [0045] FIG. 16 is a diagram illustrating an example edit mapping between two trees A and B.

DETAILED DESCRIPTION OF THE INVENTION

- [0046] As will be seen below, there is developed an efficient method 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. Also provided for below are:
  - [0047] A proof of an upper bound  $\tau$  on the size of the subtrees that must be considered for solving TASM. This threshold is independent of document size and structure.
  - [0048] An introduction of a prefix ring buffer to prune subtrees larger than  $\tau$  in  $O(\tau)$  space, during a single postorder scan of the document.
  - [0049] Also provided is TASM-postorder, an efficient and scalable method for solving TASM. The space complexity is independent of the document size and the time complexity is linear in the document size.
- [0050] To begin, the problem to be solved must first be defined.
- [0051] Definition 1 (Top-k Approximate Subtree Matching Problem).
- [0052] Let Q (query) and T (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,  $d(\dots)$  be a distance function between ordered labeled trees, and  $k \leq n$  be an integer. A sequence of subtrees,  $R=(T_{i_1}, T_{i_2}, \dots, T_{i_k})$ , is a top-k ranking of the subtrees of the document T with respect to the query Q iff
  - [0053] 1. the ranking contains the k subtrees that are closest to the query:
 
$$\forall T_j \notin R: d(Q, T_{i_k}) \leq d(Q, T_j), \text{ and}$$

**[0054]** 2. the subtrees in the ranking are sorted by their distance to the query:

$$\forall 1 \leq k: d(Q, T_i) \leq d(Q, T_{i+1}).$$

**[0055]** 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.

**[0056]** TASM relates to determining how similar one tree is to another. 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.

**[0057]** 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  $\Sigma$ . The empty node  $\epsilon$  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  $\epsilon$ . 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.

**[0058]** 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.

**[0059]** The number of  $t_p$ 's children is its fanout  $f_p$ . The node with no parent is the root node,  $\text{treeroot}(T)$ , 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  $\text{anc}(t_d)$  we denote the set of all ancestors of a node  $t_d$ . Node  $t_d$  is a descendant of  $t_i$  iff  $t_i \in \text{anc}(t_d)$ . 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  $t_j$ .

**[0060]**  $T_i$  is the subtree rooted in node  $t_i$  of T iff  $V(T_i) = \{t_x | t_x = t_i \text{ or } t_x \text{ is a descendant of } t_i \text{ in } T\}$  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  $\text{lml}(t_i)$  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  $E' = \{(t_i, t_j) | (t_i, t_j) \in E(T), t_i \in V', t_j \in V'\}$

**[0061]** 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.

**[0062]** Definition 2 (Postorder Queue)

**[0063]** Given a tree T with  $n = |T|$  nodes, the postorder queue,  $\text{post}(T)$ , of T is a sequence of pairs  $((1_1, s_1), (1_2, s_2), \dots, (1_n, s_n))$ , where  $1_i = \lambda(t_i)$ ,  $s_i = |T_i|$ , with  $t_i$  being the i-th node of T in postorder. The dequeue operation on a postorder queue  $p = (p_1, p_2, \dots, p_n)$  is defined as

$\text{dequeue}(p) = ((p_2, p_3, \dots, p_n), p_1)$

**[0064]** An edit operation transforms a tree Q into a tree T. We use the standard edit operations on trees: 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.

**[0065]** Definition 3 (Edit Mapping and Node Alignment).

**[0066]** 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

**[0067]** 1. every node is mapped:

**[0068]** (a)  $\forall q_i (q_i \in V(Q) \Leftrightarrow \exists t_j ((q_i, t_j) \in M))$

**[0069]** (b)  $\forall t_i (t_i \in V(T) \Leftrightarrow \exists q_j ((q_j, t_i) \in M))$

**[0070]** (c)  $(\epsilon, \epsilon) \in M$

**[0071]** 2. all pairs of non-empty nodes  $(q_i, t_j), (q_k, t_l) \in M$  satisfy the following conditions:

**[0072]** (a)  $q_i = q_k \Leftrightarrow t_j = t_l$  (one-to-one condition)

**[0073]** (b)  $q_i$  is an ancestor of  $q_k \Leftrightarrow t_j$  is an ancestor of  $t_l$  (ancestor condition)

**[0074]** (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.

**[0075]** 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.

**[0076]** 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.

**[0077]** Definition 4 (Cost of Node Alignment)

**[0078]** Let Q and T be ordered labeled trees, let  $\text{cst}(x) \geq 1$  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:

$$\gamma(q_i, t_j) =$$

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

**[0079]** Definition 5 (Cost of Edit Mapping)

**[0080]** 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:

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

**[0081]** The tree edit distance between two trees Q and T is the cost of the least costly edit mapping.

**[0082]** Definition 6 (Tree Edit Distance)

**[0083]** 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:

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

**[0084]** In the unit cost model all nodes have cost 1, and the unit cost tree edit distance is the minimum number of edit

operations that transforms 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 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

**[0085]** FIG. 16 illustrates an edit mapping  $M=((a_1, b_1), (a_2, b_2), (a_3, Q), (a_4, b_3), (Q, b_4), (a_5, b_5), (a_6, b_6))$  between trees A and B. If the cost of all nodes of A and B is 1,  $\gamma(a_6, b_6)=\gamma(a_3, Q)=\gamma(Q, b_4)=1$ ; 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).

**[0086]** The fastest algorithms for the tree edit distance use dynamic programming. This section discusses the classic algorithm by Zhang and Shasha 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 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. FIG. 3 illustrates the decomposition of the example document H in FIG. 1.

**[0087]** 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.

**[0088]** Definition 7 (Prefix)

**[0089]** 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  $V'=\{t_1, t_2, \dots, t_i\}$  and edges  $E'=\{(t_k, t_l) | (t_k, t_l) \in E(T), t_k \in V', t_l \in V'\}$

**[0090]** A tree with n nodes has n prefixes. The first line in FIG. 3 shows all prefixes of example document H.

**[0091]** 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 FIG. 3. 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.

**[0092]** Definition 8 (Relevant Subtree)

**[0093]** 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 1

**[0094]** Consider the example trees in FIG. 1. The relevant subtrees of G are  $G_2$  and  $G_3$ , the relevant subtrees of H are  $H_2, H_5, H_6$ , and  $H_7$ .

**[0095]** The decomposition rules for the tree edit distance are given in FIG. 2; 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.

**[0096]** The dynamic programming method 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 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 method as TASM-dynamic. (See FIG. 2A)

**[0097]** TASM-dynamic is a dynamic programming implementation of the decomposition rules in FIG. 2. 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  $\text{pfx}(Q_m, q_i)=Q_i$ .

**[0098]** The ranking, Heap, is implemented as a max-heap that stores (key, value) pairs:  $\text{max}(\text{Heap})$  returns the maximum key of the heap in constant time;  $\text{push-heap}(\text{Heap}, (k, v))$  inserts a new element (k, v) in logarithmic time; and  $\text{pop-heap}(\text{Heap})$  deletes the element with the maximum key in logarithmic time. Merging two heaps Heap and Heap' yields a new heap of size  $x=\text{max}(|\text{Heap}|, |\text{Heap}'|)$ , which contains the x elements of Heap and Heap' with the smallest keys. Instead of sorting the distances at the end, The method illustrated above 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 2

**[0099]** TASM-dynamic is computed for (k=2) for query G and document H in FIG. 1 (the cost for all nodes is 1, the input ranking is empty). FIG. 4 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. The element  $\text{pd}[g_2][h_5]$  stores the distance between the prefixes  $\text{pfx}(G_3, g_2)$  and  $\text{pfx}(H_6, h_5)$ . The upper left element is 0 (Rule (a) in FIG. 2); 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 (column 3), thus the top-2 ranking is  $R=(H_6, H_3)$ .

**[0100]** The TASM-dynamic method is one method for solving TASM. It is a fairly efficient approach since it adds a minimal overhead to the already very efficient tree edit distance method. The dynamic programming tree edit distance method 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 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  $O(|Q_m| \text{Times} |T_n|)$  must be computed. 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.

**[0101]** As will be discussed in below, 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). One piece of a solution to TASM is the prefix ring buffer together with a memory-efficient method for pruning large subtrees.

**[0102]** Definition 9 (Candidate Set):

**[0103]** Given a tree  $T$  and an integer threshold  $\tau > 0$ . The candidate set of  $T$  for threshold  $\tau$  is defined as  $\text{cand}(T, \tau) = \{T_i | t_i \in V(T), |T_i| \leq \tau, \forall t_a \in \text{anc}(t_i): |T_a| > \tau\}$ . Each element of the candidate set is a candidate subtree.

#### Example 3

**[0104]** The candidate set of the example document  $D$  in FIG. 5a for threshold  $\tau=6$  is  $\text{cand}(D, 6) = \{D_5, D_7, D_{12}, D_{17}, D_{21}\}$ .

**[0105]** It should be noted that the candidate set is not the set of all subtrees smaller than threshold  $\tau$ , but a subset. If a subtree is contained in a different subtree that is also smaller than  $\tau$ , 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.

**[0106]** Explained below is how to compute the candidate set given a size threshold  $\tau$  for a document represented as a postorder queue. Nodes that are dequeued from the postorder queue are appended to a memory buffer (see FIG. 6) 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.

**[0107]** 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 4

**[0108]** The buffer in FIG. 6 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 ( $d_5$ ).

**[0109]** 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. The nodes in the postorder queue are distinguished 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  $\tau$ .

**[0110]** 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  $\tau$  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  $\tau$ ; with the simple pruning approach, all of them will be buffered until the root node is processed.

#### Example 5

**[0111]** Consider the example document in FIG. 5. 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 (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.

**[0112]** The simple pruning is not feasible for large documents. Discussed below is ring buffer pruning which buffers candidate trees only as long as necessary and uses a look-ahead of only  $O(t)$  nodes. This is significant since the space complexity no longer depends on the document size.

**[0113]** 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 6

**[0114]** 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  $(d_7, d_1, d_2, d_3, d_4, d_5, d_6)$ ,  $s=6$ ,  $e=1$ .

**[0115]** 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  $(d_{13}, d_{14}, \dots, d_{18})$  form a subtree with root node  $d_{18}$  that is different from  $D_{18}$ . A subtree in the buffer is valid if it exists in the document. Further below is introduced the prefix array to find the leftmost valid subtree in constant time.

**[0116]** The ring buffer pruning process of a postorder queue of a document  $T$  and an empty ring buffer of size  $\tau+1$  is as follows:

- [0117]** 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.

**[0118]** 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.

**[0119]** 3. Go to 1) if the postorder queue is not empty; go to) if the postorder queue is empty but the ring buffer is not; otherwise terminate.

**[0120]** A non-leaf  $t_i$  appears at the leftmost buffer position if all its descendants 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 7

**[0121]** Ring buffer pruning is illustrated on the example tree in FIG. 5. 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 FIG. 7). 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 the process returns to Step 1 where the ring buffer is filled for the next execution of Step 2. FIG. 7 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 process terminates.

**[0122]** The following relates to a proof for the correctness of ring buffer pruning. 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  $\tau$ .

#### Example 8

**[0123]** Consider Example document  $D$  of FIG. 5,  $\tau=6$ .  $F_i$  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.)

**[0124]** Lemma 1 Let  $T$  be a tree,  $\text{cand}(T, \tau)$  the candidate set of  $T$  for threshold  $\tau$ ,  $t_i$  the  $i$ -th node of  $T$  in postorder, and  $F_i=\{t_j | t_j \in V(T), i < j \leq i-|T_i|+\tau\}$  the set of at most  $\tau-|T_i|$  nodes following  $t_i$  in postorder. For all  $1 \leq i \leq |T|$

$$T_i \in \text{cand}(T, \tau) \Leftrightarrow |T_i| \leq \tau \wedge \forall t_x (t_x \in F_i \cap \text{anc}(t_i) \Rightarrow |T_x| > \tau) \quad (1)$$

**[0125]** Proof 1

**[0126]** 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  $|T_i| \leq \tau$  it can be shown that

$$(t_x \in F_i \cap \text{anc}(t_i) \Rightarrow |T_x| > \tau) \Leftrightarrow (t_x \in \text{anc}(t_i) \Rightarrow |T_x| > \tau). \quad (2)$$

**[0127]** 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  $F_i \cap \text{anc}(t_i) = \text{anc}(t_i)$  and (2) holds. Case  $i+\tau-|T_i| < |T|$ : (2) holds for all  $t_x \in F_i \cap \text{anc}(t_i)$ . If  $t_x \in \text{anc}(t_i) \setminus F_i$ , then  $t_x \notin F_i \cap \text{anc}(t_i)$  and the left side of (2) is true.

Since any  $t_x \in \text{anc}(t_i) \setminus F_i$  is an ancestor of all nodes of both  $T_i$  and  $F_i$ ,  $|T_x| > |T_i| + |F_i| = \tau$ , and (2) holds.

**[0128]** As illustrated in FIG. 7 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.

**[0129]** Lemma 2

**[0130]** Let  $T$  be an ordered labeled tree,  $\text{cand}(T, \tau)$  be the candidate set of  $T$  for threshold  $\tau$ ,  $t_s$  be the next node of  $T$  in postorder after a non-candidate node or after the root node of a candidate subtree, or  $t_s = t_r$  and  $\text{lml}(t)$  be the leftmost leaf descendant of the root  $t_i$  of subtree  $T_i$ .

$$t_s \text{ is a leaf} \Rightarrow \exists T_i, T_j \in \text{cand}(T, \tau), t_s = \text{lml}(t_i)$$

$$t_s \text{ is a non-leaf} \Rightarrow t_s \in \{t_x | t_x \in V(T), |T_x| > \tau\} \quad (3)$$

**[0131]** Proof 2

**[0132]** Let  $\text{NC}$  be the non-candidate nodes of  $T$ .

**[0133]** (a)  $t_s = t_i$ :  $t_i$  is a leaf, thus  $t_i \notin \text{NC}$  and there is a  $t_j \in \text{cand}(T, \tau)$  such that  $t_j \in V(T_i)$ . There is no node  $t_k < t_j$ , thus  $t_j = \text{lml}(t_i)$ .

**[0134]** (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  $t_k$ .  $t_k \in \text{NC}$  by Definition 9. Since  $t_l$  is a leaf,  $t_l \notin \text{NC}$  and there must be a  $T_i \in \text{cand}(T, \tau)$  such that  $t_l \in V(T_i)$ . The equation  $t_l = \text{lml}(t_i)$  is proven by contradiction: Assume  $T_i$  has a leaf  $t_x$  to the left of  $t_l$ . As  $V(T_i) \cap V(T_j) = \emptyset$ ,  $t_x$  is to the left of  $t_j$ , and  $t_x \in V(T_i)$ , the least common ancestor of  $t_j$  and  $t_x$  is an ancestor of  $t_k$ . This is not possible since  $|T_k| > \tau \Rightarrow |T_x| > \tau \Rightarrow |T_i| > \tau$ .

**[0135]** (c)  $t_s$  follows a non-candidate node,  $t_x \in \text{NC}$ :  $t_s$  is either the parent  $t_k$  of  $t_x$  or a leaf node  $t_j$ .  $t_k \in \text{NC}$  by Definition 9, and there is a  $T_i \in \text{cand}(T, \tau)$  such that  $t_j = \text{lml}(t_i)$  (same rationale as above).

**[0136]** Theorem 1 (Correctness of Ring Buffer Pruning)

**[0137]** Given a document  $T$  and a threshold  $\tau$ , the ring buffer pruning adds a subtree  $T_i$  of  $T$  to the candidate set iff  $T_i \in \text{cand}(T, \tau)$ .

**[0138]** Proof 3

**[0139]** It can be shown 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.

**[0140]** (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.

**[0141]** (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  $t_i$ . 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.

**[0142]** With the correctness of the ring buffer pruning proven, a prefix array may now be explained.

**[0143]** 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.

**[0144]** Definition 10 (Prefix Array)

**[0145]** Let  $\text{pfx}(T, t_p)$  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  $\text{pfx}(T, t_p)$  is an integer array  $(a_1, a_2, \dots, a_p)$  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  $\text{pfx}(T, t_p)$  for which  $t_i$  is the smallest descendant:

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

**[0146]** A new node  $t_{p+1}$  is appended to the prefix array  $(a_1, a_2, \dots, a_p)$  by appending the integer  $a_{p+1} = \text{lml}(t_{p+1})$  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  $(a_i, a_{i+1}, \dots, a_{(a_i)})$  and can be found in constant time.

#### Example 9

**[0147]** FIG. 8 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  $\text{pfx}(D, d_4)$  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  $\text{pfx}(D, d_6)$  with the leftmost leaf  $d_1$  is  $(5, 1, 4, 3, 1)$  ( $i=1, a_i=5$ ).

**[0148]** 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.

**[0149]** Theorem 2

**[0150]** The prefix ring buffer pruning for a document with  $n$  nodes and with threshold  $\tau$  runs in  $O(n)$  time and  $O(\tau)$  space.

**[0151]** Proof 4 Runtime:

**[0152]** 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.

**[0153]** Algorithm 2 (prb-pruning) implements the ring buffer pruning and computes the candidate set  $\text{cand}(T, \tau)$  given the size threshold  $\tau$  and the postorder queue,  $\text{pq}$ , of document  $T$ . The prefix ring buffer is realized with two ring

buffers of size  $b = \tau + 1$ :  $\text{rbl}$  stores the node labels and  $\text{rbs}$  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. (See FIG. 2B)

**[0154]** After each call of  $\text{prb-next}$  (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).  $\text{prb-subtree}(\text{rbs}, \text{rbl}, a, b)$  returns the subtree formed by nodes  $a$  to  $b$  in the prefix ring buffer. Algorithm 3 is called until the ring buffers are empty.

**[0155]** 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 9). If the prefix ring buffer is full or the postorder queue is empty (Line 13), then nodes are removed from the prefix ring buffer. If the leftmost node is a leaf (Line 14,  $c+1-(e-s+b)\%b$  is the postorder identifier of the leftmost node), a candidate subtree is returned, otherwise a non-candidate is skipped. (See FIG. 2C)

#### Example 10

**[0156]** FIG. 9 illustrates the prefix ring buffer for the example document  $D$  in FIG. 5. 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.

**[0157]** Now presented is 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 explained above, which requires the whole document in memory, this solution uses the prefix ring buffer and keeps only candidate subtrees in memory at any point in time. The explanation for this solution starts by showing an effective threshold  $\tau$  for the size of the largest candidate subtree in the document.

**[0158]** 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,  $R' = (T_{i_1}, T_{i_2}, \dots, T_{i_k})$ , 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,  $R = (T_{i_1}, T_{i_2}, \dots, T_{i_k})$ , is the top- $k$  ranking of all subtrees of document  $T$  with respect to the query.

**[0159]** It can be shown 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. Note that this upper bound  $\tau$  is independent of size and structure of the document.

**[0160]** Lemma 3

**[0161]** Let  $Q$  and  $T$  be ordered labeled trees, then  $|T| \leq \delta(Q, T) + |Q|$ .

**[0162]** Proof 5

**[0163]** It can be shown that  $|T|-|Q|\leq\delta(Q, T)$ . 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  $\gamma(\epsilon, t_x)=\text{cst}(t_x)\geq 1$

**[0164]** Lemma 4 (Upper Bound)

**[0165]** Let  $R'=(T_{i_1}, T_{i_2}, \dots, T_{i_k})$  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  $\forall T_{i_j}(T_{i_j}\in R\Rightarrow |T_{i_j}|\leq\delta(Q, T_{i_k})+|Q|)$ .

**[0166]** Proof 6

**[0167]**  $|T_{i_j}|\leq\delta(Q, T_{i_j})+|Q|$  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.,  $R'=R$ .  $T_{i_j}\notin R'$  (since  $\delta(Q, T_{i_j})>\delta(Q, T_{i_k})$ ) but  $T_{i_j}\in R$ , thus  $R'\cap\{T_{i_j}\}\subset R$ . This contradicts  $|R|=k$ .

**[0168]** Lemma 5 (First Ranking)

**[0169]** 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  $T_{i_j}$ ,  $1\leq i\leq k$ , the following holds:  $|T_{i_j}|\leq k\wedge\delta(Q, T_{i_j})\leq|Q|c_Q+kc_T$ .

**[0170]** Proof 7

**[0171]** Let  $q_i$  be the  $i$ -th node of  $Q$  in postorder, and  $\text{lml}(t_i)$  the leftmost leaf of  $T_{i_j}$ . 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  $T_{i_j}$ ,  $1\leq i\leq k$ , 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  $T_{i_j}$  are inserted:

$$\delta(Q, T_{i_j})\leq \sum_{q_i\in V(Q)} \gamma(q_i, \epsilon) + \sum_{t_i\in V(T_{i_j})} \gamma(\epsilon, t_i) \leq |Q|c_Q + kc_T$$

since  $\gamma(q_i, \epsilon)\leq c_Q$ ,  $\gamma(\epsilon, t_i)\leq c_T$ , and  $|T_{i_j}|\leq k$ .

**[0172]** The three lemmas above are the elements for the main result in this section:

**[0173]** Theorem 3 (Maximum Subtree Size)

**[0174]** 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,  $R=(T_{i_1}, T_{i_2}, \dots, T_{i_k})$  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  $\tau=|Q|(c_Q+1)+kc_T$ :

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

**[0175]** Proof 8

**[0176]**  $|T|<k$ : (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  $R'=(T_{i_1}, T_{i_2}, \dots, T_{i_k})$  with  $\delta(Q, T_{i_j})\leq|Q|c_Q+kc_T$ , 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  $T_{i_j}\in R$ .

**[0177]** TASM-postorder (Algorithm 4) uses the upper bound  $\tau$  (see Theorem 3) to limit the size of the subtrees that must be considered, and the set of candidate subtrees,  $\text{cand}(T, \tau)$ , is computed using the prefix ring buffer proposed above. When a candidate subtree  $T_i\in\text{cand}(T, \tau)$  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.,  $|\text{Heap}|=k$ ) the upper bound  $\tau'$  provided by the intermediate ranking (see Lemma 4) may be tighter than  $\tau$ . Only subtrees of  $T_i$  that are smaller than  $\tau'$  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  $\tau'$ , then TASM-dynamic is called for this subtree and the ranking Heap is updated. All subtrees of the processed subtree are skipped (Line 13), and the remaining subtrees of  $T_i$  are traversed in reverse postorder. (See FIG. 2D)

**[0178]** Theorem 4 (Correctness)

**[0179]** 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  $Q$ .

**[0180]** Proof 9

**[0181]** If no intermediate ranking is available, all subtrees within size  $\tau=|Q|(c_Q+1)+kc_T$  are considered. The correctness of  $\tau$  follows from Theorem 3. Subtrees of size  $\tau'=\min(\tau, \max(\text{Heap})+|Q|)$  and larger are pruned only if an intermediate ranking with  $k$  subtrees is available. Then the correctness of  $\tau'$  follows from Lemma 4.

**[0182]** Theorem 5 (Complexity)

**[0183]** Let  $Q$  and  $T$  be ordered labelled 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  $O(m^2n)$  time and  $O(m^2c_Q+mkc_T)$  space.

**[0184]** Proof 10

**[0185]** The space complexity of Algorithm 4 is dominated by the call of TASM-dynamic ( $Q, T, k, \text{Heap}$ ) 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  $O(m^2c_Q+mkc_T)$ . The runtime of  $\text{tasmDynamic}(Q, T, k, \text{Heap})$  is  $O(m^2|T_i|)$ .  $\tau$  is the size of the maximum subtree that must be computed. There can be at most  $n/\tau$  subtrees of size  $\tau$  in the document and the runtime complexity is

$$O\left(\frac{n}{\tau}m^2\tau\right) = O(m^2n).$$

**[0186]** 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.

**[0187]** TASM-postorder calls TASM-dynamic for document subtrees that cannot 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.

**[0188]** We refer to TASM-dynamic with pruning as TASM-dynamic<sup>+</sup> (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. (See FIG. 2E)

#### Example 11

**[0189]** We compute TASM-dynamic<sup>+</sup> (k=2) for the query G and the document H in FIG. 1 (the cost for all nodes is 1, the input ranking is empty). The gray values in the prefix and tree distance matrixes in FIG. 4 are the values that TASM-dynamic<sup>+</sup> 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,  $\text{Heap}=(H_6, 0), (H_3, 1)$  and the pruning condition holds ( $|\text{Heap}|=2, |\text{pfx}(H_7, h_5)|=5, \max(\text{Heap})=1, |G|=3$ ). 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.

**[0190]** Theorem 6 (Correctness of TASM-Dynamic<sup>+</sup>)

**[0191]** 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<sup>+</sup> (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.

**[0192]** Proof 11

**[0193]** 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  $\text{pfx}(T_n, t_j)$  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. It needs to be shown that (1) a subtree that should be in the final ranking R is not missed, and (2) the values of td that are not computed are not needed later.

**[0194]** (1) Let  $p_i = \text{pfx}(T_n, t_i)$  be a prefix of  $T_n$ . We need to show  $\forall p_i (t_i \geq t_j \Rightarrow p_i \in R)$ : 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  $|\text{Heap}|=k$ , an intermediate ranking  $(T_{i_1}, T_{i_2}, \dots, T_{i_k})$  is available and  $\delta(Q, T_{i_1}) = \max(\text{Heap})$ ; thus a subtree  $T_i$  can not be in the final ranking if  $|T_{i_1}| > \max(\text{Heap}) + |Q|$ .  $|\text{pfx}(T_n, t_j)| > \max(\text{Heap}) + |Q|$  (pruning condition) and  $p_i \geq |\text{pfx}(T_n, t_j)|$  for  $t_i \geq t_j$ , thus  $p_i \notin R$ .

**[0195]** (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 FIG. 2). (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,  $|\text{pfx}(T_n, t_j)|$  and  $|Q|$  do not change and  $\max(\text{Heap})$  cannot increase).

**[0196]** We adapt TASM-postorder (Algorithm 4) by replacing TASM-dynamic with TASM-dynamic<sup>+</sup> in Line 12 and use this version of the algorithm in the experimental evaluation below.

**[0197]** Provided below is an experimental evaluation of the solution. The scalability of TASM-postorder is studied 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 a dictionary was used 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.

**[0198]** The scalability of TASM-postorder is studied using synthetic data from the standard XMark benchmark, 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 112 MB and 3.4M nodes to 1792 MB 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 four trees were used. A comparison is made of TASM-postorder against the state-of-the-art solution, TASM-dynamic, implemented using the tree edit distance algorithm by Zhang and Shasha.

**[0199]** Execution Time:

**[0200]** FIG. 10a shows the execution time as a function of the document size for different query sizes  $|Q|$  and fixed  $k=5$ . Similarly, FIG. 10b 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 (4 GB). As predicted above, 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 500 MB, except for very small queries. Besides scaling to much larger problems, TASM-postorder is also around four times faster than TASM-dynamic.

**[0201]** FIG. 10c 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.

**[0202]** FIG. 11 compares the execution times of TASM-dynamic<sup>+</sup> and TASM-dynamic. TASM-dynamic<sup>+</sup> is, on average, 45% faster than TASM-dynamic since distance computations to large subtrees are pruned.

**[0203]** Main Memory Usage:

**[0204]** FIG. 12 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.) It should be noted that plots for other query sizes were omitted since they follow the same trend as the ones shown in FIG. 12: the memory requirements are independent of 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 (4

GB). 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.

**[0205]** 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 FIG. 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)
```

---

**[0206]** Saxon, a state-of-the-art main-memory, Java-based XQuery processor was used in the tests. FIG. 13 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 (4 GB) 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.

**[0207]** 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.

**[0208]** Provided below is an evaluation of 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. 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.

**[0209]** FIG. 14a shows the number of relevant subtrees (y-axis) of a specific size (x-axis) that TASM-dynamic must compute to find the top-5 ranking of the subtrees of the PSD7003 dataset for a query with  $|Q|=9$  nodes. FIG. 14b 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 (while the theoretical maximum is 23). FIG. 14c 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. It should be noted that the FIGS. 14a, 14b, and 14c, for TASM-dynamic do not depend on  $k$ , but they do for TASM-postorder. With  $k=1$  for TASM-dynamic, the amount of virtual memory space required would be as large and would take as long to compute as any other value of  $k$ , for example,  $k=10$ . Such a discrepancy in the parameters used in the determination of the figures is not significant as one would imagine since TASM-dynamic always looks at subtrees the same way.

**[0210]** 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. Define  $CSS(x, T)$  is 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:

$$CSS(x, T) = \sum_{i=1}^x f_i, \quad 1 \leq x \leq |T|$$

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 FIG. 15 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).

**[0211]** Discussed above is 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. This problem has applications in the integration and cleaning of heterogeneous XML repositories, as well as in answering similarity queries. Also discussed is 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. Proved above is 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. Also provided is an effective pruning strategy that uses a prefix ring buffer and keeps only the necessary subtrees from the document in memory. As a result, provided is an algorithm that solves TASM in a single pass over the document and whose memory requirements are independent of the document itself. The analysis is verified experimentally and showed that the solution scales extremely well w.r.t. document size, query size, and the parameter  $k$ .

- [0212] The above 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, which is prevalent among persistent XML stores. The present invention opens up the possibility of applying the established and well-understood tree edit distance in practical XML systems.
- [0213] As noted above, the present invention can be used in searching databases, documents, anything that can be represented by a tree structure. As well, queries are, preferably, representable in a tree structure as well.
- [0214] The method or algorithmic steps of the invention may be embodied in sets of executable machine code stored in a variety of formats such as object code or source code. Such code is described generically herein as programming code, or a computer program for simplification. Clearly, the executable machine code may be integrated with the code of other programs, implemented as subroutines, by external program calls or by other techniques as known in the art.
- [0215] The following references are useful for a better understanding of the present invention.
- [0216] [1] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu, "Approximate XML joins," in *SIGMOD*, 2002, pp. 287-298.
- [0217] [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.
- [0218] [3] N. Augsten, M. H. Böhlen, C. E. Dyreson, and J. Gamper, "Approximate joins for data-centric XML," in *ICDE*, 2008, pp. 814-823.
- [0219] [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.
- [0220] [5] M. Weis and F. Naumann, "Dogmatix tracks down duplicates in XML," in *SIGMOD*, 2005, pp. 431-442.
- [0221] [6] N. Agarwal, M. G. Oliveras, and Y. Chen, "Approximate structural matching over ordered XML documents," in *IDEAS*, 2007, pp. 54-62.
- [0222] [7] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRANK: Ranked keyword search over XML documents," in *SIGMOD*, 2003, pp. 16-27.
- [0223] [8] K.-C. Tai, "The tree-to-tree correction problem," *J. of the ACM*, vol. 26, no. 3, pp. 422-433, 1979.
- [0224] [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.
- [0225] [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.
- [0226] [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.
- [0227] [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.
- [0228] [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.
- [0229] [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.
- [0230] [15] Z. Liu and Y. Chen, "Identifying meaningful return information for XML keyword search," in *SIGMOD*, 2007, pp. 329-340.
- [0231] [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.
- [0232] [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.
- [0233] [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, 1em plus 0.5em minus 0.4em Springer, 2007, pp. 146-157.
- [0234] [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.
- [0235] [20] R. Yang, P. Kalnis, and A. K. H. Tung, "Similarity evaluation on tree-structured data," in *SIGMOD*, 2005, pp. 754-765.
- [0236] [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.
- [0237] [22] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. of the ACM*, vol. 23, no. 1, pp. 31-42, 1976.
- [0238] [23] Y. Tian and J. M. Patel, "TALE: A tool for approximate large graph matching," in *ICDE*, 2008, pp. 963-972.
- [0239] [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.
- [0240] [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.
- [0241] [26] M. Kay, "Ten reasons why saxon xquery is fast," *IEEE Data Eng. Bull.*, vol. 31, no. 4, pp. 65-74, 2008.
- [0242] The embodiments of the invention may be executed by a computer processor or similar device programmed in the manner of method steps, or may be executed by an electronic system which is provided with means for executing these steps. Similarly, an electronic memory means such as computer diskettes, CD-ROMs, Random Access Memory (RAM), Read Only Memory (ROM) or similar computer software storage media known in the art, may be programmed to execute such method steps. As well, electronic signals representing these method steps may also be transmitted via a communication network.
- [0243] Embodiments of the invention may be implemented in any conventional computer programming language. For example, preferred embodiments may be implemented in a procedural programming language (e.g. "C") or an object-oriented language (e.g. "C++", "java", or "C#"). Alternative embodiments of the invention may be implemented as pre-

programmed hardware elements, other related components, or as a combination of hardware and software components.

**[0244]** Embodiments can be implemented as a computer program product for use with a computer system. Such implementations may include a series of computer instructions fixed either on a tangible medium, such as a computer readable medium (e.g., a diskette, CD-ROM, ROM, or fixed disk) or transmittable to a computer system, via a modem or other interface device, such as a communications adapter connected to a network over a medium. The medium may be either a tangible medium (e.g., optical or electrical communications lines) or a medium implemented with wireless techniques (e.g., microwave, infrared or other transmission techniques). The series of computer instructions embodies all or part of the functionality previously described herein. Those skilled in the art should appreciate that such computer instructions can be written in a number of programming languages for use with many computer architectures or operating systems. Furthermore, such instructions may be stored in any memory device, such as semiconductor, magnetic, optical or other memory devices, and may be transmitted using any communications technology, such as optical, infrared, microwave, or other transmission technologies. It is expected that such a computer program product may be distributed as a removable medium with accompanying printed or electronic documentation (e.g., shrink-wrapped software), preloaded with a computer system (e.g., on system ROM or fixed disk), or distributed from a server over a network (e.g., the Internet or World Wide Web). Of course, some embodiments of the invention may be implemented as a combination of both software (e.g., a computer program product) and hardware. Still other embodiments of the invention may be implemented as entirely hardware, or entirely software (e.g., a computer program product).

**[0245]** A person understanding this invention may now conceive of alternative structures and embodiments or variations of the above, all of which are intended to fall within the scope of the invention as defined in the claims that follow.

Having thus described the invention, what is claimed as new and secured by Letters Patent is:

1. A method for sorting nodes in a document tree to determine a number of closest approximations to a query represented by a query tree, the method comprising:

- a) determining a limit size of subtrees of said document tree to be considered;
- b) determining candidate subtrees of said document tree, each candidate subtree of said document tree having a size equal to or less than said limit size and each candidate subtree is not a subtree of another subtree having a size less than or equal to said limit size;
- c) for each candidate subtree, determining a tree edit distance between said candidate subtree and said query tree;
- d) sorting candidate subtrees in accordance with their respective tree edit distances with said query tree, in order to determine which candidate subtrees have least tree edit distances with said query tree;

wherein said tree edit distance is a cost to convert contents of one subtree into contents of a second subtree.

2. A method according to claim 1 wherein said candidate subtrees are stored in a memory buffer.

3. A method according to claim 1 wherein subtrees of candidate subtrees are removed from consideration as candidate subtrees.

4. A method according to claim 2 wherein said memory buffer is a ring buffer.

5. A method according to claim 2 wherein a number of nodes which can be stored in said memory buffer is equal to or less than said limit size.

6. A method according to claim 1 wherein said nodes are processed in an order such that the root node of said document tree is processed last.

7. A method according to claim 1 wherein only candidate subtrees which exist in the document tree are processed for step c).

8. Computer-readable media having encoded thereon computer readable and computer executable instructions which, when executed, executes a method for sorting nodes in a document tree to determine a number of closest approximations to a query represented by a query tree, the method comprising:

- a) determining a limit size of subtrees of said document tree to be considered;
- b) determining candidate subtrees of said document tree, each candidate subtree of said document tree having a size equal to or less than said limit size and each candidate subtree is not a subtree of another subtree having a size less than or equal to said limit size;
- c) for each candidate subtree, determining a tree edit distance between said candidate subtree and said query tree;
- d) sorting candidate subtrees in accordance with their respective tree edit distances with said query tree, in order to determine which candidate subtrees have least tree edit distances with said query tree;

wherein said tree edit distance is a cost to convert contents of one subtree into contents of a second subtree.

9. Computer-readable media according to claim 8 wherein said candidate subtrees are stored in a memory buffer.

10. Computer-readable media according to claim 8 wherein subtrees of candidate subtrees are removed from consideration as candidate subtrees.

11. Computer-readable media according to claim 9 wherein said memory buffer is a ring buffer.

12. Computer-readable media according to claim 9 wherein a number of nodes which can be stored in said memory buffer is equal to or less than said limit size.

13. Computer-readable media according to claim 8 wherein said nodes are processed in an order such that the root node of said document tree is processed last.

14. Computer-readable media according to claim 8 wherein only candidate subtrees which exist in the document tree are processed for step c).

15. A method for determining which subtrees in a document tree most closely approximate a given query tree, the method comprising:

- a) determining a limit size of subtrees of said document tree to be considered;
- b) determining candidate subtrees of said document tree, each candidate subtree of said document tree being, at most, equal in size to said limit size;
- c) for each candidate subtree, determining a cost to convert contents of said candidate subtree into contents of said query tree;
- d) sorting candidate subtrees in accordance with costs for converting said candidate subtrees into said query tree;
- e) determining which candidate subtrees have lowest costs for converting said candidate subtrees into said query tree;

tree, candidate subtrees having lowest costs for being converted into said query tree being subtrees which most closely approximate said query tree.

**16.** A method according to claim **15** wherein subtrees which are a subtree of another subtree having a size which is, at most, equal to said limit size are excluded from being a candidate subtree.

**17.** A method according to claim **15** further comprising the step of determining which candidate subtrees exist in said document tree.

**18.** A method according to claim **17** wherein candidate subtrees which do not exist in said document tree are not processed according to step c).

**19.** A method according to claim **15** wherein candidate subtrees are stored in a ring buffer.

**20.** A method according to claim **19** wherein unsuitable candidate subtrees are pruned from said buffer.

\* \* \* \* \*