Unordered tree matching and ordered tree matching: the evaluation of tree pattern queries

Yangjun Chen* and Leping Zou

Department of Applied Computer Science,
University of Winnipeg,
Winnipeg, Manitoba, R3B 2E9, Canada
E-mail: ychen2@uwinnipeg.ca
E-mail: zlpghost@gmail.com
*Corresponding author

Abstract: In this paper, we study the twig pattern matching in XML document databases. Two algorithms A1 and A2 are discussed according to two different definitions of tree embedding. By the first definition, only the ancestor-descendant relationship is considered. By the second one, we take not only the ancestor-descendant relationship, but also the order of siblings into account. Both A1 and A2 are based on a subtree reconstruction technique, by which a tree structure is reconstructed according to a given set of data streams. More importantly, by revealing an interesting property of tree encoding, we show that the subtree reconstruction can be easily extended to a strategy (i.e., A1) for checking subtree matching according to the first definition with any kind of path join or join-like operations being completely avoided. A2 needs more time and space since it deals with a more difficult problem, but without join operations involved, either. The computational complexities of both algorithms are analysed, showing that they have a better performance than any existing strategy for this problem.

Keywords: XML document; tree pattern queries; tree matching.

Reference to this paper should be made as follows: Chen, Y. and Zou, L. (2011) 'Unordered tree matching and ordered tree matching: the evaluation of tree pattern queries', *Int. J. Information Technology, Communications and Convergence*, Vol. 1, No. 3, pp.254–279.

Biographical notes: Yangjun Chen received his BS in Information System Engineering from the Technical Institute of Changsha, China, in 1982, and his Diploma and PhD in Computer Science from the University of Kaiserslautern, Germany, in 1990 and 1995, respectively. From 1995 to 1997, he worked as a Post-Doctor at the Technical University of Chemnitz-Zwickau, Germany. After that, he worked as a Senior Engineer at the German National Research Center of Information Technology (GMD) for more than two years. Since 2000, he has been a Professor in the Department of Applied Computer Science at the University of Winnipeg, Canada. His research interests include deductive databases, federated databases, document databases, constraint satisfaction problem, graph theory and combinatorics. He has more than 150 publications in these areas.

Leping Zou received his BS from the South-West JiaoTong University of China, in 2003. He is a graduate student in the Department of Applied Computer Science, University of Winnipeg, Canada.

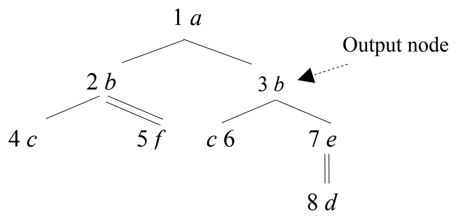
This paper is a revised and expanded version of a paper entitled 'Unordered tree matching and strict unordered tree matching: the evaluation of tree pattern queries' presented at 2010 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, (CyberC 2010), in Huangshan, China, 10–12 October 2010.

1 Introduction

In XML (World Wide Web Consortium, 2007a, 2007b), data is represented as a tree; associated with each node of the tree is an element name tag from a finite alphabet ∑. The children of a node are ordered from left to right, and represent the content (i.e., list of sub elements) of that element.

To abstract from existing query languages for XML [e.g., XPath (Florescu and Kossman, 1999), XQuery (World Wide Web Consortiumm 2007b), XML-QL (Dutch et al., 1999), and Quilt (Chamberlin et al., 2007; 2000)], we express queries as twig patterns, where nodes are labeled with symbols from $\sum \cup \{*\}$ (* is a wildcard, matching any node name) and string values, and edges are *parent-child* or *ancestor-descendant* relationships. As an example, consider the query tree shown in Figure 1(a).

Figure 1 A query tree



(a)

a[*b*[*c* and .//*f*]]/*b*[*c* and *e*//*d*

(b)

This query asks for any node of name *b* (node 3) that is a child of some node of name *a* (node 1). In addition, the node of name *b* (node 3) is the parent of some nodes of name *c* and *e* (node 6 and 7, respectively), and the node of name *e* itself is an ancestor of some node of name *d* (node 8). The node of name *b* (node 2) should also be the ancestor of a node of name *f* (node 5). The query corresponds to the XPath expression shown in Figure 1(b). In this figure, there are two kinds of edges: child edges (/-edges for short) for parent-child relationships, and descendant edges (//-edges for short) for ancestor-descendant relationships. A /-edge from node *v* to node *u* is denoted by $v \rightarrow u$ in the text, and represented by a single arc; *u* is called a */-child* of *v*. A //-edge is denoted $v \Rightarrow u$ in the text, and represented by a double arc; *u* is called a *//-child* of *v*.

In any DAG (*directed acyclic graph*), a node *u* is said to be a descendant of a node *v* if there exists a path (sequence of edges) from *v* to *u*. In the case of a twig pattern, this path could consist of any sequence of /-edges and/or //-edges. We also use *label*(*v*) to represent the symbol ($\in \sum \cup \{*\}$) or the string associated with *v*. Based on these concepts, the tree embedding can be defined as follows.

Definition 1. An embedding of a twig pattern Q into an XML document T is a mapping f: $Q \rightarrow T$, from the nodes of Q to the nodes of T, which satisfies the following conditions:

1 preserve *node label*: for each $u \in Q$, $label(u) = label(f(u))$.

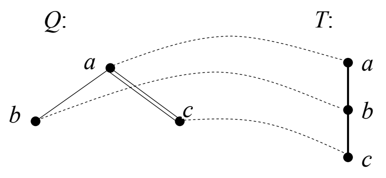
2 preserve *parent-child*/*ancestor-descendant* relationships: If $u \rightarrow v$ in Q, then $f(v)$ is a child of $f(u)$ in T; if $u \Rightarrow v$ in Q, then $f(v)$ is a descendant of $f(u)$ in T. □

If there exists a mapping from Q into T, we say, Q can be imbedded into T, or say, T contains Q.

Almost all the existing strategies for evaluating twig join patterns are designed according to this definition (Bruno et al., 2002; Chen et al., 2005; Chen et al., 2006b; Dutch et al., 1999; Gottlob et al., 2005; Lu et al., 2005; Li and Moon, 2001; Wang et al., 2003; Wang and Meng, 2005; Kaushik et al., 2002; Al-Khalifa et al., 2002; Chen, 2006a; Chen, 2006b; Chen, 2007; Chen, 2008; Chen and Che, 2006a; Chen and Che, 2006b).

This definition allows, however, a path to match a tree as illustrated in Figure 2.

Figure 2 A tree matching a path



It is because by Definition 1 the sibling order is not taken into account. Therefore, we may consider another definition as an option.

Definition 2. An embedding of a twig pattern q into an XML document T is a mapping f: $Q \rightarrow T$, from the nodes of Q to the nodes of T, which satisfies the following conditions:

1 same as (1) in Definition 1

2 same as (2) in Definition 1

3 preserve *sibling order*: For any two nodes $v_1 \in Q$ and $v_2 \in Q$, if v_1 is to the left of v_2, then $f(v_1)$ is to the left of $f(v_2)$ in T. □

This kind of tree mappings may occur in practice; especially, when the user uses axes such as 'preceding-sibling' or 'following' to indicate the left-to-right order of nodes as shown in the following example: //*A*[*B*]/following::*C*. In this expression, the axis 'following' specifies a condition that element *A* should appear to the left of element *C*.

In this paper, we present two algorithms A1 and A2 according to the above two different definitions, respectively. For both A1 and A2, the *path join* (Aghili et al., 2006; Bruno et al., 2002) or the join-like operations (such as the *result enumeration* used in Chen et al., 2006b) are completely unnecessary.

Concretely, our methods have the following advantages:

- Both A1 and A2 are able to handle twig patterns containing /-edges, //-edges, *, and branches.

- A1 runs in O(|D|·|Q|) time and O(|D|·|Q|) space, where D is a largest data stream associated with a node of Q.
- A1 generates neither matching paths nor hierarchical stacks (Chen et al., 2006b). Therefore, the costly path joins (Aghili et al., 2006; Bruno et al., 2002), as well as the result enumeration (Chen et al., 2006b), are avoided.
- A2 tackles a more difficult problem which has never been addressed before (to the best of our knowledge). This algorithm works in O(|T'|·|Q|) time and O(|T'|·$leaf_Q$) space, where T' is a subtree of T containing only those nodes that satisfy the name test of a node (or say, match a node) in Q, and $leaf_Q$ is the number of the leaf nodes of Q.
- Both the algorithms are able to output all those parts of a document, which contain one or more parts of a query tree.

The remainder of the paper is organised as follows. In Section 2, we review the related work. In Section 3, we restate the tree encoding (Zhang et al., 2001), which facilitates the recognition of different relationships among the nodes of a tree. In Section 4 and 5, we discuss our algorithms for twig pattern matching according to the above two different definitions, respectively. Finally, the paper concludes in Section 6.

2 Related work

With the growing importance of XML in data exchange, the tree pattern queries over XML documents have been extensively studied recently. Most existing techniques rely on indexing or on the tree encoding to capture the structural relationships among document elements.

XISS (Li and Moon, 2001) is a typical method based on indexing, by which single elements/attributes are indexed as the basic unit of query and a complex path expression is decomposed into a set of basic path expressions. Then, atom expressions (single elements or attributes) are recognised by directly accessing the index structure. All other kinds of expressions need join operations to stitch individual components together to get the final results.

Paths are also used as the basic indexing unit as done by DataGuide (Goldman and Widom, 1997) and fabric (Cooper et al., 2001). By DataGuide, a concise summary of path structures for a semi-structured database is provided, but restricted to raw paths. No complex path expressions or regular expression queries can be handled. Fabric works better in the sense that the so-called *refined paths* are supported. Such queries may contain branches, wild-cards and ancestor-descendant operators (//). However, any query not in the set of refined paths has to resort to join operations. Another two strategies based on the path indexing are APEX (Chung et al., 2002) and F^+B (Kaushik et al., 2002). APEX is an adaptive path index and uses data mining technique to summarise paths that frequently appear in the query workload. It has to be updated as the query workload changes. In stead of maintaining all paths starting from the root, it keeps every path segment of length 2. Obviously, to get the final results, the join operations have to be conducted. F^+B (Kaushik et al., 2002) shares the flavour of fabric (Cooper et al., 2001). It is based on the so-called forward and backward index [F&G index (Abiteboul et al., 1999)], which covers all the branching paths. It works well for pre-defined query types. In normal cases, however, such a set of F&B indexes tends to be large and therefore the performance suffers. The method discussed in Wang et al. (2003) can be considered as a quite different method, by which a document is stored as a sequence: (a_1, p_1), …, (a_i, p_i), …, (a_n, p_n), where each a_i is an element or a word in the document, and p_i a path from the root to it. Using this method, the join operations are replaced by searching a *trie* structure (wrongly called suffix tree in Wang et al., 2003). The drawback of this method is that a relatively large index structure has to be created. Another problem of this method is that a document tree that does not contain a query pattern may be designated as one of the answers due to the *ambiguity* caused by identical sibling nodes. This problem is removed by the so-called *forward prefix* checking discussed in Wang and Meng (2005). Doing so, however, the theoretical time complexity is dramatically increased.

All the above methods need to decompose a twig pattern into a set of binary relationships between pairs of nodes, such as parent-child and ancestor-descendant relations, or into a set of paths. The sizes of intermediate relations tend to be very large, even when the input and final result sizes are much more manageable. As an important improvement, *TwigStack* was proposed by Bruno et al. (2002), which compress the intermediate results by the stack encoding, which represents in linear space a potentially exponential number of answers. However, *TwigStack* achieves optimality only for the queries that contain only //-edges. In the case that a query contains both /-edges and //-edges, some useless path matching have to be performed. In the worst case, *TwigStack* needs $O(|D|^{|Q|})$ time for doing the merge joins as shown by Chen et al. (2006b, p.287). This method is further improved by several researchers. In Chen et al. (2005), *iTwigJoin* was discussed, which exploits different data partition strategies. In Lu et al. (2005), *TJFast* accesses only leaf nodes by using extended Dewey IDs. By both methods, however, the path joins can not be avoided. The method *Twig*^{2}*Stack* proposed by Chen et al. (2006b) works in a quite different way. It represents the twig results using the so-called *hierarchical stack encoding* to avoid any possible useless path matchings. In Chen et al. (2006b), it is claimed that *Twig*^{2}*Stack* needs only $O(|D|\cdot|Q| + |subTwigResults|)$ time for generating paths. But a careful analysis shows that the time complexity for this task is actually bounded by $O(|D|\cdot|Q|^2 + |subTwigResults|)$. It is because each time a node is inserted into a stack associated with a node in Q, not only the position of this node in a tree within that stack has to be determined, but a link from this node to a node in some other stack has to be constructed, which requires to search all the other stacks in the worst case. The number of these stacks is $|Q|$ (see Figure 4 in Chen et al., 2006b, to know the working process.)

A large amount of work has also been done on XPath evaluation in an XML streaming environment, such as the method discussed in Chen et al. (2006a), Gou and Chirkova (2007) and Ramanan (2007). The time complexity of the method proposed in Chen et al. (2006a) is bounded by $O(T_h\cdot Q_d\cdot|Q|\cdot|T| + |Q|^2\cdot|T|)$, where T_h is the height of T and Q_d is the largest outdegree of a node in Q. Both the methods discussed in Gou and Chirkova (2007) and Ramanan (2007) require $O(|Q|\cdot|T|)$ time. But by Ramanan (2007), extra value joins are needed. For all the XML streaming strategies, the whole document tree is searched top-down, which makes it difficult to adapt them to an indexing environment, where each node q of Q is associated with a data stream that matches q and can be found by using an index structure. Normally, such a stream is much smaller than T.

Finally, we point out that the bottom-up tree matching was first proposed in Hoffmann and O'Donnell (1982). But it concerns a very strict tree matching, by which the matching of an edge to a path is not allowed. Gottlob et al. (2005) identified an XPath fragment called Core XPath, which can be evaluated in $O(|T|\cdot|Q|)$ time. Core XPath is slightly more expressive than the twig pattern queries in that it includes axes other than /-edges and //-edges. However, algorithms in Gottlob et al. (2005) cannot be modified to use index structures since they require scanning XML documents in multiple passes. In Miklau and Suciu (2004), an algorithm for *tree homomorphism* is discussed, which is able to check whether a tree contains another and returns only a boolean answer. But our algorithms show all the subtrees that match a given twig pattern query. The node selecting queries considered in Koch (2003) are in fact a kind of extended containment queries (whether a tree contains a certain node) and cannot be used for the general purpose of twig joins. In Götz et al. (2007), a special kind of tree matching, called *tree homeomorphism*, is considered which looks for a mapping that maps each edge in Q to a path in T.

3 Tree encoding

In Zhang et al. (2001), an interesting tree encoding method was discussed, which can be used to identify different relationships among the nodes of a tree. (In fact, this encoding is the same as the concept of *timestamps* used in the depth-first search.)

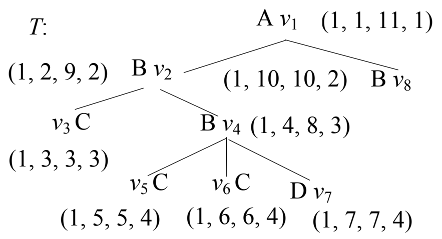
Let T be a document tree. We associate each node v in T with a quadruple (*DocId*, *LeftPos*, *RightPos*, *LevelNum*), denoted as $\alpha(v)$, where DocId is the document identifier; LeftPos and RightPos are generated by counting word numbers from the beginning of the document until the start and end of the element, respectively; and LevelNum is the nesting depth of the element in the document (see Figure 3 for illustration). By using such a data structure, the structural relationship between the nodes in an XML database can be simply determined (Zhang et al., 2001):

1 *ancestor-descendant*: a node v_1 associated with (d_1, l_1, r_1, ln_1) is an ancestor of another node v_2 with (d_2, l_2, r_2, ln_2) iff $d_1 = d_2$, $l_1 < l_2$, and $r_1 > r_2$.

2 *parent-child*: a node v_1 associated with (d_1, l_1, r_1, ln_1) is the parent of another node v_2 with (d_2, l_2, r_2, ln_2) iff $d_1 = d_2$, $l_1 < l_2$, $r_1 > r_2$, and $ln_2 = ln_1 + 1$.

3 *from left to right*: a node v_1 associated with (d_1, l_1, r_1, ln_1) is to the left of another node v_2 with (d_2, l_2, r_2, ln_2) iff $d_1 = d_2$, $r_1 < l_2$.

Figure 3 Illustration for tree encoding



In Figure 3, v_2 is an ancestor of v_6 and we have v_2.LeftPos = 2 < v_6.LeftPos = 6 and v_2.RightPos = 9 > v_6.RightPos = 6. In the same way, we can verify all the other relationships of the nodes in the tree. In addition, for each leaf node v, we set v.LeftPos = v.RightPos for simplicity, which still work without downgrading the ability of this mechanism.

In the rest of the paper, if for two quadruples $a_1 = (d_1, l_1, r_1, ln_1)$ and $a_2 = (d_2, l_2, r_2, ln_2)$, we have $d_1 = d_2$, $l_1 < l_2$, and $r_1 > r_2$, we say that a_2 is subsumed by α_1. For convenience, a quadruple is considered to be subsumed by itself. If no confusion is caused, we will use v and $a(v)$ interchangeably.

We can also assign LeftPos and RightPos values to the query nodes in Q for the same purpose.

Finally we use $T[v]$ to represent a subtree rooted at v in T.

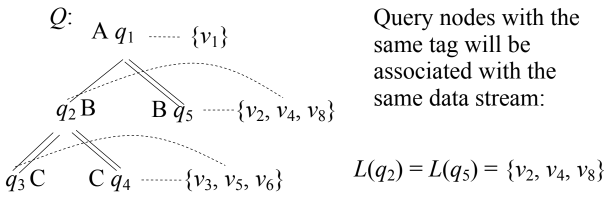
4 Algorithm A1

In this section, we discuss our first algorithm according to Definition 1. The main idea of this algorithm is the so-called *subtree reconstruction*, by which a tree structure is established according to a given set of quadruples (called a data stream in Bruno et al., 2002). Therefore, we will first discuss an algorithm for this task in Section 4.1. Then, in Section 4.2, we give our algorithm to check twig patterns that contains /-edges, //-edges, and branches. Next, in 4.3, we handle the general case with * being considered.

4.1 Tree reconstruction

As with *TwigStack*, each node q in a twig pattern (or say, a query tree) Q is associated with a data stream $L(q)$, which contains the positional representations (quadruples) of the database nodes that match q (more exactly, satisfy the node name test at q). All the quadruples in a data stream are sorted by their (DocID, LeftPos) values. For example, in Figure 4, we show a query tree containing five nodes and four edges and each node is associated with a list of matching nodes of the document tree shown in Figure 3, sorted according to their (DocID, LeftPos) values. For simplicity, we use the node names in a list, instead of the node's quadruples.

Figure 4 Illustration for $L(q_i)$'s


Q:
A q_1
$\{v_1\}$
q_2 B
B q_5
$\{v_2, v_4, v_8\}$
q_3 C
C q_4
$\{v_3, v_5, v_6\}$
Query nodes with the same tag will be associated with the same data stream:
$L(q_2) = L(q_5) = \{v_2, v_4, v_8\}$

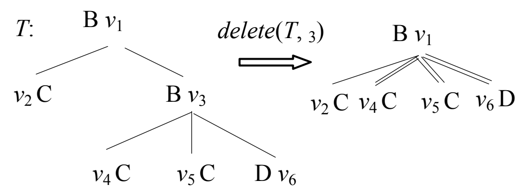
In addition, the data streams associated with different nodes in Q may be the same. We will use $\boldsymbol{q}$ to represent the set of such query nodes and denote by $L(\boldsymbol{q})$ the data stream shared by them. Without loss of generality, assume that the query nodes in $\boldsymbol{q}$ are sorted by their LeftPos values.

In the following, we will also use $L(Q) = \{L(\boldsymbol{q}_1), ..., L(\boldsymbol{q}_l)\}$ to represent all the data streams with respect to Q, where each $\boldsymbol{q}_i$ ($i = 1, ..., l$) is a set of sorted query nodes that share the same data stream.

First, we discuss how to reconstruct a tree structure from the data streams, based on the concept of *matching subtrees*, and defined below.

Let T be a tree and v be a node in T with parent node u. Denote by *delete*(T, v) the tree obtained from T by removing node v. The children of v become //-children of u (see Figure 5.)

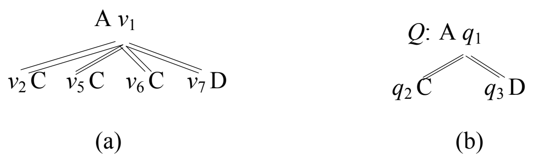
Figure 5 The effect of removing v_3 from T



Definition 3 (*matching subtrees*). A matching subtree T' of T with respect to a query tree Q is a tree obtained by a series of deleting operations to remove any node in T, which does not match any non-wildcard node in Q. □

For example, the tree shown in Figure 6(a) is a matching subtree of the document tree shown in Figure 3 with respect to the query tree shown in Figure 6(b).

Figure 6 A matching tree and a query tree



Given $L(Q)$, what we want is to construct a matching subtree from them to facilitate the checking of twig patterns.

The algorithm given below handles the case when the streams contain nodes from a single XML document. When the streams contain nodes from multiple documents, the algorithm is easily extended to test equality of DocId before manipulating the nodes in the streams.

We will execute an iterative process to access the nodes in $L(\boldsymbol{q}_1) \cup \ldots \cup L(\boldsymbol{q}_l)$ one by one:

1 Identify a data stream $L(\boldsymbol{q})$ with the *last node* being of the *maximal LeftPos* value. Choose the last node v of $L(\boldsymbol{q})$. Remove v from $L(\boldsymbol{q})$.

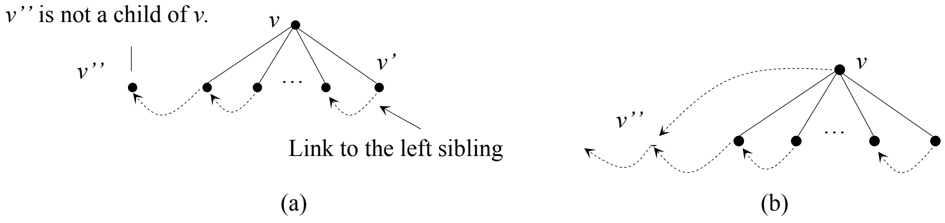
2 Let v' be the node chosen just before v. We do the following.

- If v' is not a child (descendant) of v, create a link from v to v', called a *right-sibling* link and denoted as *right-sibling*(v) = v'.

- If *v'* is a child (descendant) of *v*, we will first create a link from *v'* to *v*, called a *parent* link and denoted as *parent*(*v'*) = *v*. Then, we will go along the right-sibling chain starting from *v'* until we meet a node *v''* which is not a child (descendant) of *v*. For each encountered node *u* except *v''*, set *parent*(*u*) ← *v*. Set *right-sibling*(*v*) ← *v''*. Remove *right-sibling*(*u*) for each child *u* of *v*.

Figure 7 is a pictorial illustration of this process.

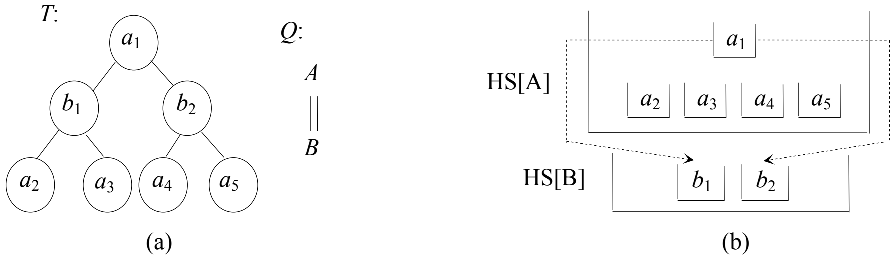
Figure 7 Illustration for construction of matching subtrees



In Figure 7(a), we show the navigation along a right-sibling chain starting from *v'* when we find that *v'* is a child (descendant) of *v*. This process stops whenever we meet *v''*, a node that is not a child (descendant) of *v*. Figure 7(b) shows that the right-sibling link of *v* is set to *v''*, which is previously pointed to by the right-sibling link of *v*'s right-most child. In addition, all the right-sibling links of the child nodes of *v* are discarded since they will no longer be used.

The following is the formal description of the algorithm, which needs only $O(D\cdot|Q|)$ time. It improves the method mentioned in Chen et al. (2006b) for generating trees within a hierarchical stack associated with a node in *Q*, which requires $O(D\cdot|Q|^2)$ time (see the brief description of the method given in Example 1 in Chen et al., 2006b). For the purpose of comparison, we briefly analyse that method below.

Figure 8 Illustration for hierarchical stacks



In Figure 8(b), we show the hierarchical stacks associated with the two nodes *A* and *B* of *Q* with respect to *T'* shown in Figure 8(a). In Chen et al. (2006b), the nodes in a data stream associated with each node of *Q* are sorted by their (DocID, RightPos) values. So a_1 is visited last. When it is inserted into HS[*A*] (hierarchical stack of *A*), all those stacks in HS[*A*], which are not a descendant of some other stack, will be checked to establish ancestor-descendant links. In addition, to generate links to some other stacks in HS[*B*], similar checks will also be performed. This needs $O(|Q|)$ time in the worst case.

We elaborate this process since it can be extended to an efficient algorithm for twig pattern matching without involving the path join (Bruno et al., 2002) or the result enumeration (Chen et al., 2006b) in an elegant way.

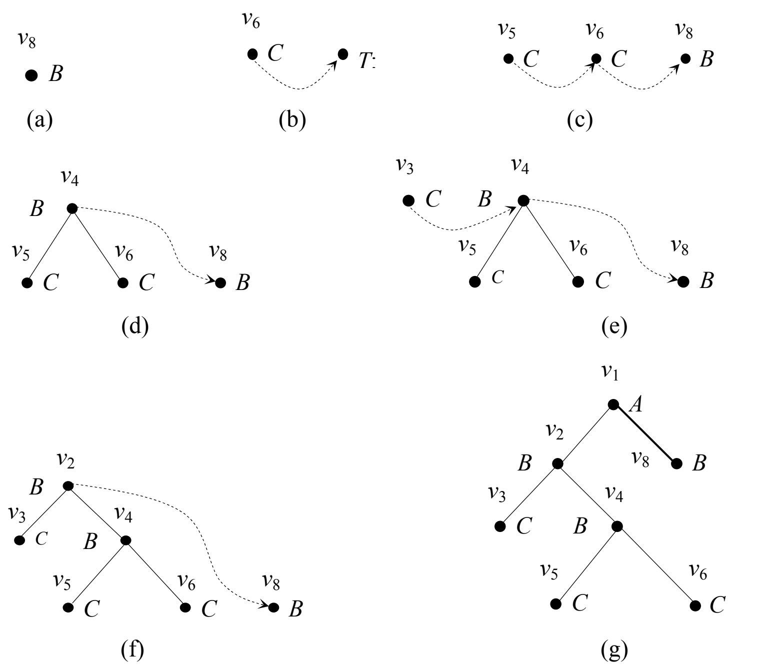
```
Algorithm matching-tree-construction(L(Q))
input: all data streams L(Q).
output: a matching subtree T'.
begin
1      repeat until each L(q) in L(Q) becomes empty
2      {identify q such that the last node v of L(q) is of the maximal LeftPos value; remove v from
       L(q);
3      generate node v;
4      if v is not the first node created then
5          {let v' be the node generated just before v;
6           if v' is not a child (descendant) of v then (*Using the tree encoding, this checking
            needs only a constant time.*)
7            {right-sibling(v) ← v';}          (*generate a right-sibling link.*)
8            else
9            {v'' ← v'; w ← v';                (*v'' and w are two temporary units.*)
10            while v'' is a child (descendant) of v do
11            {parent(v'') ← v; (*generate a parent link. Also, indicate whether v'' is a /-child or
              a //-child.*)
12             w ← v''; v'' ← right-sibling(v'');
13             remove right-sibling(w);
14            }
15            right-sibling(v) ← v'';
16     }
17  }
end
```

In the above algorithm, for each v chosen from a $L(\boldsymbol{q})$, a node is created. At the same time, if v is not the first node, a right-sibling link of v is established to the node v', which is created just before v, if v' is not a child (descendant) of v (see line 7). Otherwise, we go into a **while**-loop to travel along the right-sibling chain starting from v' until we meet a node v'' which is not a child (descendant) of v. During the process, a parent link is generated for each node encountered except v'' (see lines 9–14). Finally, the right-sibling link of v is set to be v'' (see line 15).

Example 1. Consider the query tree and the associated data streams shown in Figure 4 once again. Applying the above algorithm to the data streams, we generate a series of data structures as shown in Figure 9.

In step 1, v_8 is checked since it has the maximal LeftPos; and a node for it is created [see Figure 9(a)]. In Step 2, we meet v_6. It is not a descendant of v_8. So a right-link from v_6 to v_8 is created [see Figure 9(b)]. In Step 3, we will generate a right-link from v_5 to v_6 [see Figure 9(c)]. In a next step, we encounter v_4. It is the parent of v_5. Thus, a parent link from v_5 to v_4 will be constructed. We will also navigate the right-link chain starting from v_5, finding another child node v_6 of v_4 and stopping at node v_8, which is not a descendant of v_4. The resulting data structure is shown in Figure 9(d). The right-link from v_6 of v_8 is replaced with the right-link from v_4 of v_8. The remaining computation steps are illustrated in Figure 9(e), Figure 9(f), and Figure 9(g), respectively. □

Figure 9 Sample trace



Proposition 1. Let T be a document tree. Let Q be a twig pattern. Let $L(Q) = \{L(\boldsymbol{q}_1), \ldots, L(\boldsymbol{q}_l)\}$ be all the data streams with respect to Q and T, where each $\boldsymbol{q}_i$ $(1 \leq i \leq l)$ is a subset of sorted query nodes of Q, which share the same data stream. Algorithm *matching-tree-construction*($L(Q)$) generates the matching subtree T' of T with respect to Q correctly.

Proof. Denote $L = |L(\boldsymbol{q}_1)| + \ldots + |L(\boldsymbol{q}_l)|$. We prove the proposition by induction on L.

Basis. When $L = 1$, the proposition trivially holds.

Induction hypothesis. Assume that when $L = k$, the proposition holds.

Induction step. We consider the case when $L = k + 1$. Assume that all the quadruples in $L(\boldsymbol{q}_1) \cup \ldots \cup L(\boldsymbol{q}_l)$ are $\{v_1, \ldots, v_k, v_{k+1}\}$ with LeftPos(v_1) > LeftPos(v_2) > … > LeftPos(v_k) > LeftPos(v_{k+1}). The algorithm will first generate a tree structure T_k for $\{v_1, \ldots, v_k\}$. In terms of the induction hypothesis, T_k is correctly created. It can be a tree or a forest. If it is a forest, all the roots of the subtrees in T_k are connected through the right-sibling links. When we meet v_{k+1}, we consider two cases:

1 v_{k+1} is an ancestor of v_k,

2 v_{k+1} is to the left of v_k.

In Case 1, the algorithm will generate an edge (v_{k+1}, v_k), and then travels along the right-sibling chain starting from v_k until we meet a node v which is not a descendant of v_{k+1}. For each node v' encountered, except v, an edge (v_{k+1}, v') will be generated. Therefore, T_{k+1} is correctly constructed. In Case 2, the algorithm will generate a right-sibling link from v_{k+1} to v_k. It is obviously correct since in this case v_{k+1} cannot be an ancestor of any other node. This completes the proof. □

The time complexity of this process is easy to analyse. First, we notice that each quadruple in all the data streams is accessed only once. Secondly, for each node in *T'*, all its child nodes will be visited along a right-sibling chain for a second time. So we get the total time

$$O(|D|\cdot|Q| + = O(|D|\cdot|Q|) + O(|D|) = O(|D|\cdot|Q|),$$

where d_i represents the outdegree of node v_i in *T'*.

During the process, for each encountered quadruple, a node v will be generated. Associated with this node have we at most two links (a right-sibling link and a parent link). These two links, as well as v's quadruple, will be kept until v's parent is met. So at any time point, the used extra space is bounded by $min\{|D|, leaf_{T'}\}$ since for any two nodes on the same path only one is associated with links. Here, $leaf_{T'}$ is the number of the leaf nodes of *T'*.

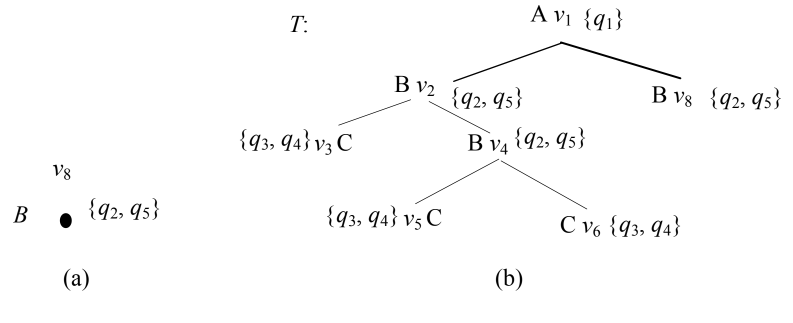
*4.2 Twig pattern matching without **

In fact, the algorithm discussed in Section 4.1 hints an efficient way for twig pattern matching.

We first observe that during the reconstruction of a matching subtree *T'*, we can also associate each node v in *T'* with a *query node stream* $QS(v)$. That is, each time we choose a v with the largest LeftPos value from a data stream $L(\boldsymbol{q})$, we will insert all the query nodes in $\boldsymbol{q}$ into $QS(v)$. For example, in the first step shown in Figure 9, the query node stream for v_8 can be determined as shown in Figure 10(a).

In this way, we can create a matching subtree as illustrated in Figure 10(b), in which each node in *T'* is associated with a sorted query node stream. If we check, before a q is inserted into the corresponding $QS(v)$, whether $Q[q]$ (the subtree rooted at q) can be imbedded into $T'[v]$, we get in fact an algorithm for twig pattern matching. The challenge is how to conduct such a checking efficiently.

Figure 10 Illustration for generating *QS*'s



For this purpose, we associate each q in Q with a variable, denoted $\chi(q)$. During the process, $\chi(q)$ will be dynamically assigned a series of values $a_0, a_1, \ldots, a_m$ for some m in sequence, where $a_0 = \phi$ and a_i's ($i = 1, \ldots, m$) are different nodes of T'. Initially, $\chi(q)$ is set to $a_0 = \phi$. $\chi(q)$ will be changed from a_{i-1} to $a_i = v$ ($i = 1, \ldots, m$) when the following conditions are satisfied.

1 v is the node currently encountered

2 q appears in $QS(u)$ for some child node u of v

3 q is a //-child, or q is a /-child, and u is a /-child with $label(u) = label(q)$.

Then, each time before we insert q into $QS(v)$, we will do the following checking:

1 Let $q_1, \ldots, q_k$ be the child nodes of q.

2 If for each q_i ($i = 1, \ldots, k$), $\chi(q_i)$ is equal to v and $label(v) = label(q)$, insert q into $QS(v)$.

Since the matching subtree is constructed in a bottom-up way, the above checking guarantees that for any $q \in QS(v)$, $T'[v]$ contains $Q[q]$.

Let $v_1, \ldots, v_j$ be the children of v in T'. All the $QS(v_i)$'s ($i = 1, \ldots, j$) should also be added into $QS(v)$. This process can be elaborated as follows.

Let $QS(v_i) = \{q_{i_1}, \ldots, q_{i_j}\}$ ($i = 1, \ldots j$). Then, we have q_{i_1}.LeftPos < … < q_{i_j}.LeftPos. (Recall that all the query nodes inserted into $QS(v_i)$ come from a same $\boldsymbol{q}$, in which all the elements are sorted by their LeftPos values.) Each time we insert a q into $QS(v_i)$, we can check whether it is subsumed by the query node q' which has just been inserted before. If it the case, q will not be inserted since the embedding of $Q[q']$ in $T[v_i]$ implies the embedding of $Q[q]$ in $T[v_i]$. (Note that LeftPos(q') < LeftPos(q). q cannot be an ancestor of q'.)

Thus, $QS(v_i)$ contains only those query nodes which are not on the same path. Therefore, we must also have q_{i_1}.RightPos < … < q_{i_j}.RightPos. So the query nodes in $QS(v_i)$ are increasingly sorted by both LeftPos and RightPos values. Obviously, $|QS(v_i)| \leq leaf_Q$. We can store $QS(v_i)$ as a linked list. Let QS_1 and QS_2 be two sorted lists with $|QS_1| \leq leaf_Q$ and $|QS_2| \leq leaf_Q$. The union of QS_1 and QS_2 ($QS_1 \cup QS_2$) can be performed by scanning both QS_1 and QS_2 from left to right and inserting the query node of QS_2 into QS_1 one by one. During this process, any query node in QS_1, which is subsumed by some query node in QS_2 will be removed; and any query node in QS_2, which is subsumed by some query in QS_1, will not be inserted into QS_1. The result is stored in QS_1. From this, we can see that the resulting linked list is still sorted and its size is bounded by $leaf_Q$. We denote this process as $merge(QS_1, QS_2)$ and define $merge(QS_1, \ldots, QS_{j-1}, QS_j)$ to be $merge(merge(QS_1, \ldots QS_{j-1}), QS_j)$.

In the following, we present our first algorithm *A1-1*($L(Q)$) for queries containing only /-edges, //-edges, and branches. During the process, another algorithm *subsumption-check*(v, $\boldsymbol{q}$) may be invoked to check whether any $q \in \boldsymbol{q}$ can be inserted into $QS(v)$, where $\boldsymbol{q}$ is a subset of query nodes such that $L(\boldsymbol{q})$ contains v.

The algorithm *A1-1*($L(Q)$) is similar to Algorithm *matching-tree-construction*( ), by which a quadruple is removed in turn from the data stream and a node v for it is generated and inserted into the matching subtree.

In addition, two data structures are used:

- D_{root} – a subset of document nodes v such that Q can be embedded in $T[v]$
- D_{output} – a subset of document nodes v such that $Q[q_{output}]$ can be embedded in $T[v]$, where q_{output} is the output node of Q.

In these two data structures, all nodes are decreasingly sorted by their LeftPos values.

```
Algorithm A1-1(L(Q))
input: all data streams L(Q).
output: a matching subtree T' of T, D_root and D_output.
begin
1   repeat until each L(q) in L(Q) becomes empty {
2       identify q such that the last node v of L(q) is of the maximal LeftPos value; remove v
        from L(q); generate node v;
3       if v is the first node created then
4       {QS(v) ← subsumption-check(v, q);}
5       else
6       {let v' be the quadruple chosen just before v, for which a node is constructed;
7        if v' is not a child (descendant) of v then
8        {right-sibling(v) ← v';
9           QS(v) ← subsumption-check(v, q); }
10       else
11       {v'' ← v'; w ← v';                (*v'' and w are two temporary units.*)
12          while v'' is a child (descendant) of v do
13          {parent(v'') ← v;             (*generate a parent link. Also, indicate whether v''
                                           is a /-child or a //-child.*)
14          for each q in QS(v'') do {
15          if ((q is a //-child) or
16               (q is a /-child and v'' is a /-child and
17               label(q) = label(v'')))
18          then χ(q) ← v;}
19          w ← v''; v'' ← right-sibling(v'');
20          remove right-sibling(w);
21            }
22          right-sibling(v) ← v'';
23          }
24          q ← subsumption-check(v, q);
25          let v_1, …, v_j be the child nodes of v;
26          q' ← merge(QS(v_1), …, QS(v_j));
27          remove QS(v_1), …, QS(v_j);
28          QS(v) ← merge(q, q');
29          }
30      }
end
Function subsumption-check($v$, $\boldsymbol{q}$) (*$v$ satisfies the node name test at each $q$ in $\boldsymbol{q}$.*)
1   $QS \leftarrow \Phi$;
2   for each $q$ in $\boldsymbol{q}$ do {
3       let $q_1, \ldots, q_j$ be the child nodes of $q$.
4       if for each /-child $q_i$ $\chi(q_i) = v$ and for each //-child $q_i$ $\chi(q_i)$ is subsumed by $v$ then
5       {$QS \leftarrow QS \cup \{q\}$;
6         if $q$ is the root of $Q$ then $D_{root} \leftarrow D_{root} \cup \{v\}$;
7         if $q$ is the output node then $D_{output} \leftarrow D_{output} \cup \{v\}$;
8       }
9   }
10  return $QS$;
end
```

The output of *A1-1*( ) is D_{root} and D_{output}. Based on them, we can generate another subtree *T''* of *T* (like a matching subtree), which contains only those nodes v such that $T[v]$ contains $Q[r]$ with $label(v) = label(r)$ or contains $Q[o]$ with $label(v) = label(o)$, where r and o represent the root and the output node of Q, respectively. We call a node v an *r-node* if $T[v]$ contains $Q[r]$ with $label(v) = label(r)$, or an *o-node* if $T[v]$ contains $Q[o]$ with $label(v) = label(o)$. Search *T''*. Any node v, which is an *o*-node and also a child of some *r*-node, should be an answer if o is not a /-child of r. Otherwise, an *o*-node has to be a /-child of some *r*-node to be an answer.

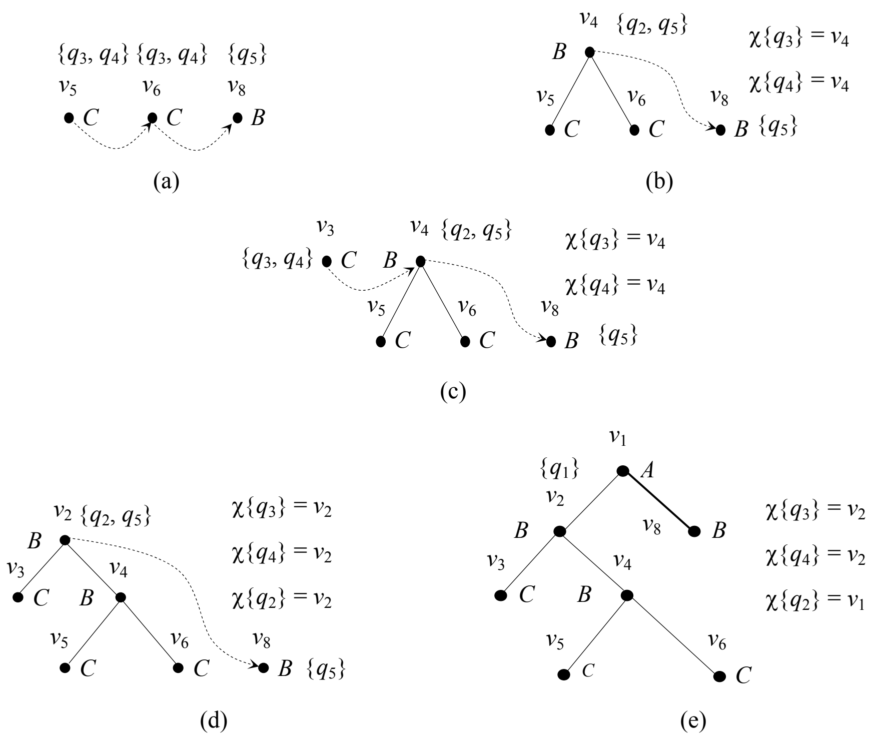
Algorithm *A1-1*( ) does almost the same work as Algorithm *matching-tree-construction*( ). The main difference is lines 14–18 and lines 24–28. In lines 14 - 18, we set χ values for some q's. Each of them appears in a $QS(v')$, where v' is a child node of v, satisfying the conditions 1–3 given above. In lines 24–28, we use the merging operation to construct $QS(v)$.

In function *subsumption-check*( ), we check whether any q in $\boldsymbol{q}$ can be inserted into QS by examining the ancestor-descendant/parent-child relationships (see line 4). For each q that can be inserted into QS, we will further check whether it is the root of Q or the output node of Q, and insert it into D_{root} or D_{output}, respectively (see lines 6–7).

Example 2. Applying Algorithm *A1-1* to the data streams shown in Figure 4, we will find that the document tree shown in Figure 3 contains the query tree shown in Figure 4. We trace the computation process as shown in Figure 11.

In the first three steps, we will generate part of the matching subtree as shown in Figure 11(a). Associated with v_8 is a query node stream: $QS(v_8) = \{q_5\}$. Although q_2 also matches v_8, it cannot survive the subsumption check (see line 4 in *subsumption-check*( )). So it does not appear in $QS(v_8)$. In addition, we have $QS(v_5) = QS(v_6) = \{q_3, q_4\}$. It is because both q_3 and q_4 are leaf nodes and can always satisfy the subsumption checking. In a next step, we will meet the parent v_4 (appearing in $L(\{q_2, q_5\})$ of v_5 and v_6. So we are able to get $\chi(q_3) = v_4$ and $\chi(q_4) = v_4$ [see Figure 11(b)]. In terms of these two values, we know that q_2 should be inserted into $QS(v_4)$. q_5 is a leaf node and also inserted into $QS(v_4)$. In addition, $QS(v_5)$ and $QS(v_6)$ should also be merged into it. In the fifth step, we meet v_3. $QS(v_3) = \{q_3, q_4\}$ [see Figure 11(c)]. In the sixth step, we meet v_2 (in $L(\{q_2, q_5\})$). It is the parent of v_3 and v_4. According to $QS(v_3) = \{q_3, q_4\}$ and $QS(v_4) = \{q_2, q_5\}$, as well as the fact that both q_5 and v_4 are /-child nodes and $label(q_5) = label(v_4) = B$, we will set $\chi(q_3) = \chi(q_4) = \chi(q_2) = \chi(q_5) = v_2$ (see Figure 11(d)). Thus, we have $QS(v_2) = \{q_2, q_5\}$. Finally, in step 7, according to $QS(v_2) = \{q_2, q_5\}$ and $QS(v_8) = \{q_5\}$, we will set $\chi(q_2) = v_1$ and $\chi(q_5) = v_1$ [see Figure 11(e)], leading to the insertion of q_1 into $QS(v_1)$. □

Figure 11 Sample trace



In Example 2, we see that if we just want to record only those parts of *T*, which contain the whole *Q* or the subtree rooted at the output node, a $QS(v)$ can be removed once *v*'s parent is encountered. However, if we maintain them, we are able to tell all the possible containment, i.e., which parts of *T* contain which parts of *Q*.

In the following, we prove the correctness of this algorithm. First, we prove a simple lemma.

Lemma 1. Let v_1, v_2, and v_3 be three nodes in a tree with v_3.LeftPos < v_2.LeftPos < v_1.LeftPos. If v_1 is a descendant of v_3, then v_2 must also be a descendant of v_3.

Proof. We consider two cases:

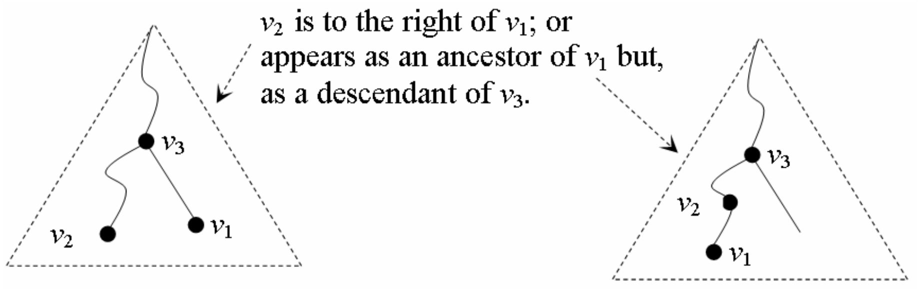
1 v_2 is to the left of v_1

2 v_2 is an ancestor of v_1.

In Case 1, we have v_1.*RightPos* > v_2.*RightPos*. So we have v_3.*RightPos* > v_1.*RightPos* > v_2.*RightPos*. This shows that v_2 is a descendant of v_3. In Case 2, v_1, v_2, and v_3 are on the same path. Since v_2.*LeftPos* > v_3.*LeftPos*, v_2 must be a descendant of v_3. □

We illustrate Lemma 1 by Figure 12, which is helpful for understanding the proof of Proposition 2 given below.

Figure 12 Illustration for Lemma 1


v_2 is to the right of v_1; or appears as an ancestor of v_1 but, as a descendant of v_3.
v_3
v_2
v_1
v_3
v_2
v_1

Proposition 2. Let *Q* be a twig pattern containing only /-edges, //-edges and branches. Let *v* be a node in the matching subtree *T'*, with respect to *Q* created by Algorithm *A1-1*. Let *q* be a node in *Q*. Then, *q* appears in *QS*(*v*) if and only if *T'*[*v*] contains *Q*[*q*].

Proof. If-part. A query node *q* is inserted into *QS*(*v*) by executing function *subsumption-check*( ), which shows that for any *q* inserted into *QS*(*v*) we must have *T'*[*v*] containing *Q*[*q*] for the following reason:

1 *label*(*v*) = *label*(*q*)

2 For each //-child *q'* of *q* there exists a child *v'* of *v* such that *T*[*v'*] contains *Q*[*q'*] (see line 15 in *A1-1*( ))

3 For each /-child *q''* of *q* there exists a /-child *v''* of *v* such that *T*[*v''*] contains *Q*[*q''*] and *label*(*v''*) = *label*(*q''*) (see lines 16–17 in *A1-1*( )).

In addition, a query node *q* in *QS*(*v*) may come from a *QS* of some child node of *v*. Obviously, we have *T'*[*v*] containing *Q*[*q*].

Only-if-part. The proof of this part is tedious. In the following, we give only a proof for the simple case that *Q* contains no /-edges, which is done by induction of the height *h* of the nodes in *T'*.

Basis. When $h = 0$, for the leaf nodes of *T'*, the proposition trivially holds.

Induction step. Assume that the proposition holds for all the nodes at height $h \leq k$. Consider the nodes *v* at height $h = k + 1$. *A*ssume that there exists a *q* in *Q* such that *T'*[*v*] contains *Q*[*q*] but *q* does not appear in *QS*(*v*). Then, there must be a child node q_i of *q* such that (1) $\chi(q_i) = \phi$, or (2) $\chi(q_i)$ is not subsumed by *v* when *q* is checked against *v*. Obviously, Case 1 is not possible since *T'*[*v*] contains *Q*[*q*] and q_i must be contained in a subtree rooted at a node *v'* which is a child (descendant) of *v*. So $\chi(q_i)$ will be changed to a value not equal to ϕ in terms of the induction hypothesis. Now we show that Case 2 is not possible, either. First, we note that during the whole process, $\chi(q_i)$ may be changed several times since it may appear in more than one *QS*'s. Assume that there exist a sequence of nodes $v_1, \ldots, v_k$ for some $k \geq 1$ with v_1.LeftPos > v_2.LeftPos >... > v_k.LeftPos such that q_i appears in $QS(v_1), \ldots, QS(v_k)$. In terms of the induction hypothesis, $v' = v_j$ for some $j \in \{1, \ldots, k\}$. Let l be the largest integer $\leq k$ such that v_l.LeftPos > v.LeftPos. Then, for each v_p ($j \leq p \leq l$), we have

v'.LeftPos ≥ v_l.LeftPos > v.LeftPos.

In terms of Lemma 1, each v_p ($j \leq p \leq l$) is subsumed by v. When we check q against v, the actual value of $\chi(q_i)$ is the node name for some v_p's parent, which is also subsumed by v (in terms of Lemma 1), contradicting (2). The above explanation shows that Case 2 is impossible. This completes the proof of the proposition. □

Lemma 1 helps to clarify the only-if part of the above proof. In fact, it reveals an important property of the tree encoding, which enables us to save both space and time. That is, it is not necessary for us to keep all the values of $\chi(q_i)$, but only one to check the ancestor-descendant/parent-child relationship. Due to this property, the path join (Bruno et al., 2002), as well as the result enumeration (Chen et al., 2006b), can be completely avoided.

The time complexity of the algorithm can be divided into three parts:

1 The first part is the time spent on accessing $L(Q)$. Since each element in a $L(Q)$ is visited only once, this part of cost is bounded by O(|D|·|Q|).

2 The second part is the time used for constructing $QS(v_j)$'s. For each node v_j in the matching subtree, we need $O\left(\sum_i c_{j_i}\right)$ time to do the task, where c_{j_i} is the outdegree of q_{j_i}, which matches v_j (see line 2 and 3 in Function *subsumption-check*( ) for explanation). So this part of cost is bounded by

$$O\left(\sum_j \sum_i c_{j_i}\right) \leq \left(O\,|D| \cdot \sum_k^{|Q|} c_k\right) = O(|D| \cdot |Q|).$$

3 The third part is the time for establishing χvalues, which is the same as the second part since for each q in a $QS(v)$ its χvalue is assigned only once.

Therefore, the total time is $O(|D|\cdot|Q|)$.

The space overhead of the algorithm is easy to analyse. Besides the data streams, each node in the matching subtree needs a parent link and a right-sibling link to facilitate the subtree reconstruction, and an QS to calculate χ values. So the extra space requirement is bounded by $O(|D|\cdot|Q| + |D| + |Q|) = O(|D|\cdot|Q|)$.

However, if we record only those parts of T', which contain the whole Q or the subtree rooted at the output node, the runtime memory usage must be much less than O(|D|·|Q|) for the following two reasons:

1 The QS data structure for a node is removed once its parent node is created. So the space overhead is bounded by O(|D|·$leaf_Q$)

2 During the whole process, the elements in the data streams are removed one by one.

Of course, if we want to record all those parts of T', which contain one or more parts of Q, we need $O(|D|\cdot|Q|)$ space to store all the results.

*4.3 Twig pattern matching with **

In the case that *Q* contains *, any query node labeled with * will be associated with the same data stream *L* that contains all the elements of a document. For this reason, when we reconstruct the matching subtree, we only use this stream and the data streams associated with any non-wildcard query node needn't be considered. But any node *v* in *L* should be associated with a query node stream, denoted *s*(*v*), which contains all the query nodes *q* that match *v* (i.e., *v* satisfies the node name test at *q*). Note that *s*(*v*) should also contains all the *-nodes.

In terms of such an arrangement, the algorithm *A1-1*( ) given in the previous section is changed as follows.

```
Algorithm A1-2(L)
input: L – a data stream containing all the elements of a document.
output: a matching subtree T' of T, $D_{root}$ and $D_{output}$.
begin
1      Let $L = \{v_1, \ldots, v_n\}$;
2      for i = n downto 1 do {
3        generate node v for $v_i$;
4        if v is the first node created then
5        {QS(v) ← subsumption-check(v, s($v_i$)); }
6        else
7        {let v' be the quadruple chosen just before v, for which a node is constructed;
8        if v' is not a child (descendant) of v then
9        {right-sibling(v) ← v';
10         QS(v) ← subsumption-check(v, s($v_i$)); }
11       else
12       {v'' ← v'; w ← v';              (*v'' and w are two temporary units.*)
13         while v'' is a child (descendant) of v do
14         {parent(v'') ← v;   (*generate a parent link.*)
15           for each q in QS(v'') do {
16             if ((q is a //-child) or
17                 (q is a /-child and v'' is a /-child and
18                 label(q) = label(v'')))
19             then χ(q) ← v; }
20             w ← v''; v'' ← right-sibling(v'');
21             remove right-sibling(w);
22         }
23         right-sibling(v) ← v'';
24       }
25       let $v_1, \ldots, v_j$ be the child nodes of v;
26       q' ← merge(QS($v_1$), …, QS($v_k$));
27         remove $QS(v_1)$, …, $QS(v_k)$;
28         $\boldsymbol{q}$ ← subsumption-check($v$, $\boldsymbol{q}$);
29     $QS(v)$ ← merge($\boldsymbol{q}$, $\boldsymbol{q}$');
30  }
end
```

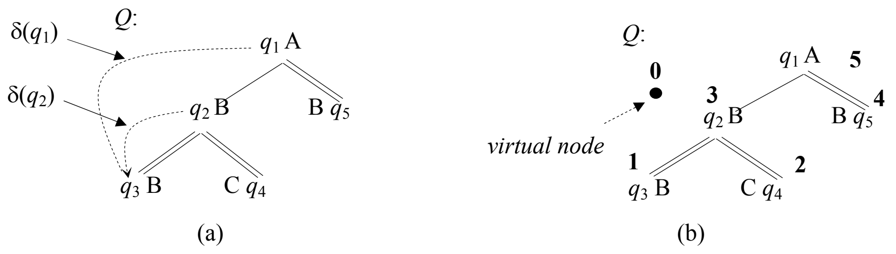
The above algorithm works in a way similar to *A1-1*( ). The only difference is that this algorithm generates the matching subtree along a single data stream that contains all the elements of a document. The computational complexities can also be analysed similar to *A1-1*( ). If the number of the wildcards is bounded by a constant, the time cost of the algorithm is bounded by $O(|D|\cdot|Q|)$. Otherwise, it needs $O(|T|\cdot|Q|)$ time. In both cases, the space overhead is bounded by $O(|D|\cdot|Q|)$.

5 Algorithm A2

In this section, we discuss our second algorithm A2 according to Definition 2. It needs more time and space. But it deals with a more difficult problem, by which both the ancestor-descendant relationship and the sibling order are considered.

For this purpose, we associate each node q of Q with a link from it to the left-most leaf node in $Q[q]$, denoted by $\delta(q)$ [see Figure 13(a)].

Figure 13 Labelled trees and postorder numbering



For a leaf node q', $\delta(q')$ is defined to be q' itself. So in Figure 13(a), we have $\delta(q_1) = \delta(q_2) = \delta(q_3) = q_3$. If we consider $q' = \delta(q)$ as a function, we can also define its reverse function, denoted by $\delta^{-1}(q')$. Its value is a set containing all those nodes q such that $\delta(q) = q'$, including q' itself. For example, for q_3 in Figure 13(a), we have $\delta^{-1}(q_3) = \{q_1, q_2, q_3\}$, $\delta^{-1}(q_4) = \{q_4\}$, and $\delta^{-1}(q_5) = \{q_5\}$.

In addition, we devise a new data structure, which is able to record both subtree embedding and ordering simultaneously and efficiently.

1 First, we number the nodes of Q in postorder. So the nodes in Q will be referenced by their postorder numbers. Additionally, we set a *virtual node* for Q, numbered 0, considered to be to the left of any node in Q. (See the boldfaced numbers in Figure 13(b) for illustration)

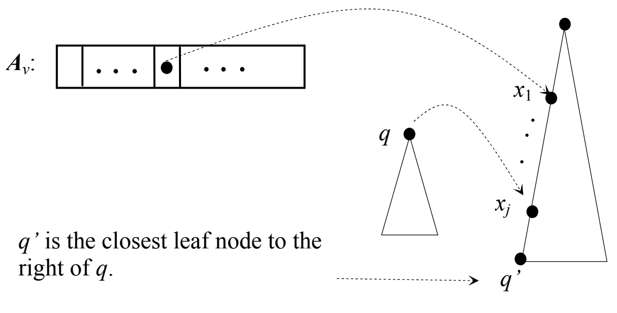
2 Each time we create a node v in T', we associate it with an array $\boldsymbol{A}_v$ of length $|Q|$, indexed from 0 to $|Q| - 1$. In A_v, each entry is a query node (represented by its postorder number) or ϕ, defined below:

$$A_v[q] = \begin{cases} \max\{x \mid x \in \delta^{-1}(q') \wedge T[v]\ \mathit{embeds}\ Q[x], & \text{if there is a leaf node } q' \text{ larger than } q \text{ such that } \delta^{-1}(q') \text{ contains at least one } x \text{ with } T[v] \text{ embedding } Q[x]; \\ \phi & \text{otherwise.} \end{cases}$$

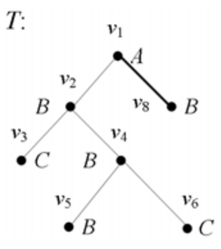
See Figure 14(a) for illustration.

In Figure 14(a), *q'* represents a closest leaf node to the right of *q* (i.e., the least leaf node larger than *q*) such that there exists at least one $x \in \delta^{-1}(q')$ with $T[v]$ embedding $Q[x]$. We may have more than one nodes $x \in \delta^{-1}(q')$ such that $T[v]$ embeds $Q[x]$. But we make $A_v[q]$ point to the largest one since the embedding of a tree in $T[v]$ implies the embedding of any of its subtrees in $T[v]$. In this way, the left-to-right order is implicitly recorded.

Figure 14 Illustration for $A_v[q]$ and a tree



(a)



(b)

Such entries can be produced in a computation as below.

- If we find $Q[x]$ can be embedded in $T[v]$, we will set $A_v[q_1]$, ..., $A_v[q_k]$ to *x*, where each q_l ($1 \le l \le k$) is a query node to the left of *x*, to record the fact that *x* is the closest node to the right of q_l such that $T[v]$ embeds $Q[x]$.
- If some time later we find another node *x'* such that $Q[x']$ can be embedded in $T[v]$, we distinguish between two cases:
  a If *x'* is to the right of *x*, we will set $A_v[p_1]$, ..., $A_v[p_s]$ to *x'*, where each p_l ($1 \le l \le s$) is to the left of *x'* but to the right of q_k.

b If x' is an ancestor of x, we will find all those entries in $\boldsymbol{A}_v$ pointing to a descendant of x' on the left-most path in $Q[x']$. Replace these entries with x'.

As an example, consider node v_4 in T shown in Figure 14(b).

After it is checked against node 1 (q_3) of Q shown in Figure 13(b), we will set A_{v_4} [0] to 1 since node 1 (q_3) of Q is the closest node to the right of node 0 (the virtual node of Q) such that $T[v_4]$ embeds $Q[q_3]$ [see Figure 15(a)]. At a later time point, we find that $T[v_4]$ also embeds $Q[q_2]$. Then, A_{v_4} [0] is changed to 3 [see Figure 15(b)]. It is because node 1 (q_3) is a descendant of node 3 (q_2) on the left-most path in $Q[q_2]$. In the subsequent computation, we will find that $T[v_4]$ can embed $Q[q_5]$. In order to record this fact, A_{v_4} will be further modified as shown in Figure 15(c) since node 4 (q_5) is the closest node to the right of node 1 (q_3), 2 (q_4), and 3 (q_2) such that $T[v_4]$ embeds $Q[q_5]$.

Figure 15 Changes in A_{v_4}

A_{v_4} :

1	ϕ	ϕ	ϕ	ϕ

(a)

3	ϕ	ϕ	ϕ	ϕ

(b)

1	5	5	5	ϕ

(c)

Using A_v's, the ordered tree embedding can be checked very efficiently as follows:

- Let v be a node taken from $L(q)$. Let $v_1, \ldots, v_k$ be the child nodes of v. Let $q_1, \ldots, q_l$ be the child nodes of q. We first check A_{v_1} starting from A_{v_1} [p], where $p = \delta(q) - 1$.

  We begin the searching from $\delta(q)$ - 1 because it is the closest node to the left of the first child of q. Let A_{v_1} [p] = p'. If p' is an ancestor of q_1, this shows that $T[v_1]$ embes $Q[q]$ and thus $T[v]$ embes $Q[q]$. If p' is q_1 and (q, q_1) is a //-edge, or both (q, q_1) and (v, v_1) are /-edges, we will check A_{v_2} [p'] in a next step. Otherwise, we check A_{v_2} [p] in a next step. This process continues until one of the following conditions is satisfied:

  1 all A_{v_i} 's (i = 1, …, k) are exhausted

  2 all q_j (j = 1, …, l) are covered.

Case 1 shows that $T[v]$ is not able to embed $Q[q]$ while the Case 2 indicates an embedding of $Q[q]$ in $T[v]$. If $T[v]$ embeds $Q[q]$, A_v should be changed as described by (a) and (b) above. To facilitate this process, each node v in T is associated with a variable, denoted $\tau(v)$, used to keep the most recently found query node r such that $T[v]$ embeds $Q[r]$. Initially, $\tau(v)$ is 0.

In the following algorithm *A2*( ), the input is also a set of data streams for Q: $B(Q) = \{B(\boldsymbol{q}_1), ..., B(\boldsymbol{q}_l)\}$, where each $B(\boldsymbol{q}_j)$ contains the same entries as $L(\boldsymbol{q}_j)$, but sorted by the RightPos values. From this, a matching subtree can also be constructed, but each time we identify a data stream $B(\boldsymbol{q})$ with the first node being of the minimal RightPos value. In addition, for each node v generated for an element from a $B(\boldsymbol{q})$, A_v is created and each entry is initialised to ϕ. Then, for each $q \in \boldsymbol{q}$, we will check whether $T[v]$ embeds $Q[q]$. This is done by executing lines 7 - 16, in which two index variables: i and j are used to scan the children of v and q, respectively. The searching begins from A_{v_1} [p], where $p = \delta(q) - 1$ (see line 8). In each iteration of the **while**-loop (see lines

10–15), we check v_i against q_j by examining whether one of the following two conditions is satisfied:

1 $A_{v_i}[p]$ is an ancestor of q_j

2 $A_{v_i}[p]$ is equal to q_j, and (q, q_j) is a //-edge, or both (q, q_j) and (v, v_i) are /-edges.

If (i) holds, $T[v_i]$ embeds $Q[q]$. So $T[v]$ embeds $Q[q]$. If (ii) holds, $T[v_i]$ embeds $Q[q_j]$. We will continue to check $T[v_{i+1}]$ against $Q[q_{j+1}]$ in a next step. This is done by executing $p' \leftarrow A_{v_{i+1}}[p']$ (line 13), by which we get a query node represented by p', which is the closest to the right of q_j, such that $T[v_{i+1}]$ embeds $Q[p']$.

```
Algorithm A2(B(Q)) (*B(Q) is similar to L(Q). But each data stream in it is sorted by
                        RightPos values.*)
Input: all data streams B(Q).
Output: S_v's, which show the tree embedding.
begin
1     repeat until each B(q) in B(Q) become empty
2     {identify q such that the first element v of B(q) is of the minimal RightPos value;
        remove v from B(q);
3       generate node v; $A_v \leftarrow \phi$; $S_v \leftarrow \phi$;
4       let $v_1, \ldots, v_k$ be the children of v.
5       for each $q \in$ q do {              (*nodes in q are sorted.*)
6           let $q_1, \ldots, q_l$ be the children of q;
7           if $l = 0$ then $j \leftarrow 1$             (*set j = 1 to execute lines 16–25.*)
8           else {  $p \leftarrow \delta(q) - 1$;
9                   $i \leftarrow 1$; $j \leftarrow 1$; $p' \leftarrow [p]$;
10                  while $i \le k$ and $j \le l$ do
11    {             if ($p'$ is an ancestor of $q_j$ then $j \leftarrow l + 1$ (*T[v_i] embeds Q[q].*)
12                  else if     $p'$ is $q_j$ and ((q, q_j) is a //-edge, or
                                both (q, q_j) and (v, v_i) are /-edges))
13                  then { $p' \leftarrow A_{v_{i+1}}[p']$; $i \leftarrow i + 1$; $j \leftarrow j + 1$; }
14                  else {$p' \leftarrow A_{v_{i+1}}[p]$; $i \leftarrow i + 1$;}
15              }
16    }
17    if $j = l + 1$ then
18       { $S_v \leftarrow S_v \cup \{q\}$;
19         if q is to the right of $\tau(v)$
20         then {   $r \leftarrow \tau(v)$;
21                  for $b = r$ to $q - 1$ do
22                  {if $b$ is to the left of $q$ then $A_v[b] \leftarrow q$; }
23              }
24          else replace with $q$ all those entries pointing to a descendant of $q$ on the left-most path
            in $Q[q]$ in $A_v$;
25          $\tau(v) \leftarrow q$;
26      }
27  }
28  for $i = 1$ to $k$ do $A_v \leftarrow merge(A_v, A_{v_i})$;
29  remove $A_{v_1}, \ldots, A_{v_k}$;
30  }
end
```

In the above algorithm, if for v_i against q_j both (i) and (ii) do not hold, $T[v_i]$ cannot embed $Q[q_j]$ and we will check v_{i+1} against q_j by doing $p' \leftarrow A_{v_{i+1}}[p]$ (see line 14). This process continues until one of the following conditions is met: $i > k$, or $j > l$ (see line 10).

If $i > k$, there exists a $Q[q_a]$ ($1 \leq a \leq l$) which cannot be embedded in any $T'[v_c]$ ($1 \leq c \leq k$). If $j > l$, we must have $j = l + 1$ (see line 17), showing that each $Q[q_a]$ ($1 \leq a \leq l$) is embedded in a $T'[v_c]$ ($1 \leq c \leq k$) in the left-to-right order. So $T[v]$ contains $Q[q]$. If q is to the right of $\tau(v)$, for all those postorder numbers b such that $\tau(v) \leq b \leq q - 1$ and b is to the left of q, $A_v[b]$ is set to q. (See lines 18 - 23.) If q is an ancestor of $\tau(v)$, replace with q all those entries in A_v pointing to a descendant of q on the left-most path in $Q[q]$. It is because the embedding of $Q[q]$ in $T[v]$ implies the embedding of $Q[\tau(v)]$ in $T[v]$ (see line 24). Finally, we need to merge each into A_v (line 28) since the embedding of a subtree in $T[v_i]$ implies the embedding of that subtree in $T[v]$. $merge(A_v, A_{v_i})$ is defined as below:

$$merge\left(A_v, A_{v_i}\right)[j] = \begin{cases} \max\left\{A[j], A_{v_i}[j]\right\} & \textit{if } A[j] \textit{ and } A_{v_i}[j] \textit{ are on the same path}; \\ \min\left\{A[j], A_{v_i}[j]\right\} & \textit{otherwise.} \end{cases}$$

In this definition, we handle ϕ as a negative integer (e.g., –1) and consider it as a descendant of any node. Obviously, if $A_v[j]$ and $A_{v_i}[j]$ are on the same path, $merge(A_v[j], A_{v_i}[j])$ should be set to be $max\{A_v[j], A_{v_i}[j]\}$. However, if $A_v[j]$ and $A_{v_i}[j]$ are on different paths, $merge(A_v[j], A_{v_i}[j])$ is set to be $min\{A_v[j], A_{v_i}[j]\}$. It is because in A_v each entry $A_v[j]$ is the closest node j' to the right of j such that $T[v]$ contains $Q[j']$. In line 29, we remove $A_{v_1}, \ldots, A_{v_k}$ since they will not be used any more.

The time complexity of A2( ) is obviously bounded by $O(|T'|\cdot|Q|)$. But its space overhead is in the order of $O(leaf_{T'}\cdot|Q|)$. It is because after a v is checked all the arrays associated with its children are removed. So at any time point during the execution, at most $leaf_{T'}$ nodes in T' are associated with an array (see line 29.)

The algorithm is somehow related to the method discussed in Kilpelainen and Mannila (1995), in which each node in Q is associated with an array of size $|T|$. So its space complexity is in the order of $O(|T|\cdot|Q|)$. Especially, that method cannot be adapted to an indexing environment since an index is always established over T.

6 Conclusions

In this paper, two new algorithms A1 and A2 are discussed, according to two different definitions of tree embedding. By the first definition, we consider only the ancestor-descendant relationship among the nodes in a tree structure. By the second definition, not only the ancestor-descendant relationship but also the order of siblings is taken into account. Almost all the existing strategies are designed according to the first definition. We provide the second definition as an option in the case that the user wants to do so. Both A1 and A2 have the best worst-case time complexities for this problem. Especially, we show that for the twig pattern matching problem, neither the join nor the result enumeration (a join-like operation) is necessary. Our experiments demonstrate that our methods are both effective and efficient for the evaluation of twig pattern queries.

References

Abiteboul, S., Buneman, P. and Suciu, D. (1999) *Data on the Web: from Relations to Semistructured Data and XML*, Morgan Kaufmann Publisher, Los Altos, CA 94022, USA.

Aghili, A., Li, H., Agrawal, D. and Abbadi, A.E. (2006) TWIX: twig structure and content matching of selective queries using binary labeling, *INFOSCALE*.

Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Srivastava, D. and Wu, Y. (2002) 'Structural joins: a primitive for efficient XML query pattern matching', *Proc. of IEEE Int. Conf. on Data Engineering*.

Bruno, N., Koudas, N. and Srivastava, D. (2002) 'Holistic twig joins: optimal XML pattern matching', *Proc. SIGMOD Int. Conf. on Management of Data*, June, pp.310–321, Madison, Wisconsin.

Chamberlin, D.D., Clark, J., Florescu, D. and Stefanescu, M. (2007) *XQuery1.0: An XML Query Language*, available at http://www.w3.org/TR/query-datamodel/.

Chamberlin, D.D., Robie, J. and Florescu, D. (2000) 'Quilt: an XML query language for heterogeneous data sources', *WebDB*.

Chen, Y., Davison, S.B. and Zheng, Y. (2006a) 'An efficient XPath query processor for XML streams, *Proc. ICDE*, 3–8 April, Atlanta, USA.

Chen, S., Li, H-G., Tatemura, J., Hsiung, W-P., Agrawa, D. and Canda, K.S. (2006b) 'Twig^{2}Stack: bottom-up processing of generalized-tree-pattern queries over XML documents', *Proc. VLDB*, September, pp.283–294, Seoul, Korea.

Chen, T., Lu, J. and Ling, T.W. (2005) 'On boosting holism in XML twig pattern matching', *Proc. SIGMOD*, pp.455–466.

Chen, Y. (2006a) 'Evaluating tree pattern queries based on tree embedding', *Proc. Int. Conf. Software Engineering and Data Technologies (ICSOFT'2006)*, 11–14 September, Vol. II, pp.79–85, Springer Verlag, Setubal, Portugal.

Chen, Y. (2006b) 'On the stack encoding and twig joins', *WSEAS Transactions on Information Science & Applications*, October, Vol. 3, No. 10, pp.1865–1872.

Chen, Y. (2007) 'An efficient algorithm for tree matching in XML databases', *Journal of Computer Science*, Vol. 3, No. 7, pp.487–493, Science Publication.

Chen, Y. (2008) 'On the XML data stream and XPath queries', *Proc. 19th Information Resources Management Association Intl. Conference*, 18–20 May, pp.62–72, Niagara, Ontario, Canada.

Chen, Y. and Che, D. (2006a) 'Efficient processing of XML tree pattern queries', *Journal of Advanced Computational Intelligence and Intelligent Informatics*, Vol. No. 5, pp.738–743.

Chen, Y. and Che, D. (2006b) 'Efficient processing of XML tree pattern queries in the presence of integrity constraints', *Journal of Advanced Computational Intelligence and Intelligent Informatics*, Vol. No. 5, pp.744–751.

Chung, C., Min, J. and Shim, K. (2002) 'APEX: an adaptive path index for XML data', *ACM SIGMOD*, June.

Cooper, B.F., Sample, N., Franklin, M., Hialtason, A.B. and Shadmon, M. (2001) 'A fast index for semistructured data', *Proc. VLDB*, September, pp.341–350.

Dutch, A., Fernandez, M., Florescu, D., Levy, A. and Suciu, D. (1999) 'A query language for XML', *Proc. 8th World Wide Web Conf.*, May, pp.77–91.

Florescu, D. and Kossman, D. (1999) 'Storing and querying XML data using an RDMBS', *IEEE Data Engineering Bulletin*, Vol. 22, No. 3, pp.27–34.

Goldman, R. and Widom, J. (1997) 'DataGuide: enable query formulation and optimization in semistructured databases', *Proc. VLDB*, August, pp.436–445.

Gottlob, G., Koch, C. and Pichler, R. (2005) 'Efficient algorithms for processing XPath queries', *ACM Transaction on Database Systems*, June, Vol. 30, No. 2, pp.444–491.

Götz, M., Koch, C. and Martens, W. (2007) 'Efficient algorithms for the tree Homeomorphism problem', *Pro. Int. Symposium on Database Programming Language*.

Gou, G. and Chirkova, R. (2007) 'Efficient algorithms for evaluating XPath over streams', *Proc. SIGMOD*, 12–14 June.

Hoffmann, C.M. and O'Donnell, M.J. (1982) 'Pattern matching in trees', *J. ACM*, Vol. 29, No. 1, pp.68–95.

Kaushik, R., Bohannon, P., Naughton, J. and Korth, H. (2002) 'Covering indexes for branching path queries', *ACM SIGMOD*, June.

Kilpelainen, P. and Mannila, H. (1995) 'Ordered and unordered tree inclusion', *SIAM Journal of Computing*, Vol. 24, pp.340–356.

Koch, C. (2003) 'Efficient processing of expressive node-selecting queries on XML data in secondary storage: a tree automata-based approach', *Proc. VLDB*, September.

Li, Q. and Moon, B. (2001) 'Indexing and querying XML data for regular path expressions', *Proc. VLDB*, September, pp.361–370.

Lu, J., Ling, T.W., Chan, C.Y. and Chan, T. (2005) 'From region encoding to extended Dewey: on efficient processing of XML twig pattern matching', *Proc. VLDB*, pp.193–204.

Miklau, G. and Suciu, D. (2004) 'Containment and equivalence of a fragment of XPath', *J. ACM*, Vol. 51, No. 1, pp.2–45.

Ramanan, P. (2007) 'Holistic join for generalized tree patterns', *Information Systems*, Vol. 32, pp.1018–10

Wang, H. and Meng, X. (2005) 'On the sequencing of tree structures for XML indexing', *Proc. Conf. Data Engineering*, April, pp.372–385, Tokyo, Japan.

Wang, H., Park, S., Fan, W. and Yu, P.S. (2003) 'ViST: a dynamic index method for querying XML data by tree structures, *SIGMOD Int. Conf. on Management of Data*, June, San Diego, CA.

World Wide Web Consortium (2007) 'XML path language (XPath)', *W3C Recommendation*, available at http://www.w3.org/TR/ xpath20.

World Wide Web Consortium (2007) 'XQuery 1.0: an XML query language', *W3C Recommendation*, January, Version 1.0, available at http://www.w3.org/TR/xquery.

Zhang, C., Naughton, J., Dewitt, D., Luo, Q. and Lohman, G. (2001) 'Supporting containment queries in relational database management systems', *Proc. of ACM SIGMOD*.