TREE EMBEDDING AND XML QUERY EVALUATION

Yangjun Chen
Dept. Applied Computer Science, University of Winnipeg, Canada

Keywords: XML databases, Trees, Paths, XML pattern matching, Twig joins.

Abstract: Tree pattern matching is one of the most fundamental tasks for XML query processing. Prior work has typically decomposed the tree pattern into binary structural (parent-child and ancestor-descendent) relationships or paths, and then stitch together these basic matches by join operations. In this paper, we propose a new algorithm that explores both the document tree and the twig pattern in a bottom-up way and show that the join operation can be completely avoided. The new algorithm runs in $O(|T|\cdot|Q|)$ time and $O(|Q|\cdot leaf_T)$ space, where T and Q are the document tree and the tree pattern query, respectively; and $leaf_T$ represents the number of leaf nodes in T.

1 INTRODUCTION

In XML, data is represented as a tree; associated with each node of the tree is an element type from a finite alphabet $\sum$. The children of a node are ordered from left to right, and represent the content (i.e., list of subelements) of that element.

To abstract from existing query languages for XML (e.g. XPath, XQuery, XML-QL, and Quilt), we express queries as twig patterns (or say, tree patterns) where nodes are types from $\sum \cup \{*\}$ (* is a wildcard, matching any node type) and string values, and edges are *parent-child* or *ancestor-descendant* relationships. As an example, consider the query tree shown in Fig. 1, which asks for any node of type *b* (node 2) that is a child of some node of type *a* (node 1). In addition, the *b* type (node 2) is the parent of some *c* type (node 4) and an ancestor of some *d* type (node 5). Type *b* (node 3) can also be the parent of some *e* type (node 7). The query corresponds to the following XPath expression:

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

In this figure, there are two kinds of edges: child edges (*c*-edges) for parent-child relationships, and descendant edges (*d*-edges) for ancestor-descendant relationships. A *c*-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 *c-child* of v. A *d*-edge is denoted $v \Rightarrow u$ in the text, and represented by a double arc; u is called a *d-child* of u.

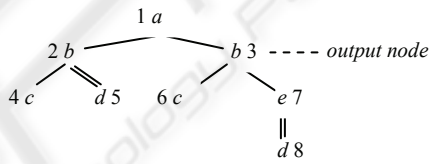

Figure 1: A query tree.

Definition 1. An embedding of a tree 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:

(i) Preserve node type: For each $u \in Q$, u and $f(u)$ are of the same type. (or more generally, u's node test is satisfied by $f(u)$.)

(ii) Preserve *c*/*d*-child 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 exits a mapping from Q into T, we say, Q can be imbedded into T, or say, T contains Q. In addition, if label(T's root) = label(Q's root), we say that the embedding is *root-preserving*.

As an example, see the document tree and the tree pattern query shown in Fig. 2(a).

There exits a mapping from Q to T as illustrated by the dashed lines, by which each node of Q is mapped to a different node of T. However, according to the definition, an embedding could map several nodes of Q (of the same type) to the same node of T, as shown in Fig. 2(b), by which nodes q_2 and q_5 in Q are mapped onto a single node v_2 in T, and q_3 and q_4 are mapped onto a single node v_3 in T.

Chen Y. (2008).
TREE EMBEDDING AND XML QUERY EVALUATION.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - DISI*, pages 173-178
DOI: 10.5220/0001690301730178

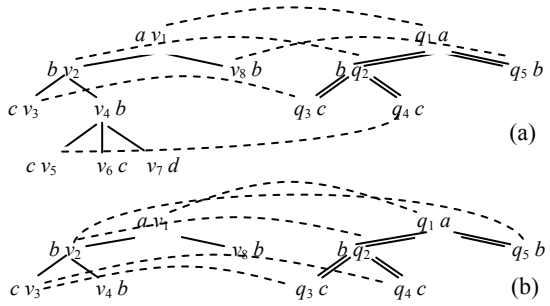

Figure 2: Illustration for tree embedding.

For the purpose of query evaluation, either of the mappings is recognized as a tree embedding.

In fact, almost all the existing strategies are designed to work in this way.

In this paper, we discuss a new algorithm, which works in a bottom-up way and shows that the join or join-like operations can be completely avoided. The algorithm works in $O(|T|\cdot|Q|)$ time and $O(|Q|\cdot leaf_T)$ space, where $leaf_Q$ is the number of the leaf nodes of Q.

The remainder of the paper is organized as follows. In Section 2, we review the related work. In Section 3, we discuss our main algorithm. In Section 4, we extend this algorithm to general cases that '$\vee$' and '$\neg$' logic operators are included. Finally, a short conclusion is set forth in Section 5.

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, such as the methods discussed in (Li and Moon, 2001; Goldman and Widom, 1997; Cooper and *et al*., 2001; Chung and *et al*., 2002; Kaushik and *et al*., 2002; Wang and *et al*., 2003; Wang and Meng, 2005).

All the above mentioned methods need to decompose a tree 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.* (Bruno and *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 *d*-edges. In the case that a query contains both *c*-edges and *d*-edges, some useless path matchings have to be performed. In addition, in the worst case, *TwigStack* needs $O(|D|^{|Q|})$ time for doing the merge joins as shown by Chen *et al*. (see page 287 in (Chen and *et al*., 2006)), where D is a largest data stream associated with a node q in Q, which contains all the document nodes that match q. Since then, several methods that improve *TwigStack* in some way have been reported. For instance, *iTwigJoin* (Chen and *et al*., 2005) exploits different data partition possibilities while *TJF*ast (Lu and *et al*., 2005) accesses only leaf nodes of document trees by using Dewey IDs. But both of them still need to do some useless matchings as shown by the theoretical analysis made in (Choi and et al., 2003). *Twig*^{2}*Stack* (Chen and *et al*., 2006) is the most recent method that improves *TwigStack*. By this method, the stack encoding is replaced with the *hierarchical stack encoding*, by which each stack associated with a query node contains an ordered sequence of stack trees. In this way, the path joins are replaced by the so called *result enumeration*. In (Chen and *et al*., 2006), it is claimed that *Twig*^{2}*Stack* needs only $O(|D|\cdot|Q| + |subTwigResults|)$ time. But a careful analysis shows that the time complexity of the method 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. The number of these stacks is $|Q|$ (see Fig. 4 in (Chen and *et al*., 2006) to know the working process.) The bottom-up method discussed in (Chen, 2007) needs no join operations.

In this paper, we improve the method proposed in (Chen, 2007) by removing all the merging operations, which are needed by that method to form matching sets associated with each node in T. In addition, the method is extended to handle general cases.

3 ALGORITHM

In this section, we discuss our algorithm according to Definition 1. The main idea of this algorithm is to search both T and Q bottom-up and checking the subtree embedding by generating dynamic data structures. In the process, a tree labeling technique is

used to facilitate the recognition of nodes' relationships. Therefore, in the following, we will first show the tree labeling in 3.1. Then, in 3.2, we discuss the main algorithm.

3.1 Tree Labeling

Before we give our main algorithm, we first restate how to label a tree to speed up the recognition of the relationships among the nodes of trees.

Consider a tree T. By traversing T in *preorder*, each node v will obtain a number (it can an integer or a real number) $pre(v)$ to record the order in which the nodes of the tree are visited. In a similar way, by traversing T in *postorder*, each node v will get another number $post(v)$. These two numbers can be used to characterize the ancestor-descendant relationships as follows.

Proposition 1. Let v and v' be two nodes of a tree T. Then, v' is a descendant of v iff $pre(v') > pre(v)$ and $post(v') < post(v)$.

Proof. See Exercise 2.3.2-20 in (Knuth, 1969).

If v' is a descendant of v, then we know that $pre(v') > pre(v)$ according to the preorder search. Now we assume that $post(v') > post(v)$. Then, according to the postorder search, either v' is in some subtree on the right side of v, or v is in the subtree rooted at v', which contradicts the fact that v' is a descendant of v. Therefore, $post(v')$ must be less than $post(v)$. The following example helps for illustration.

Example 1. See the pairs associated with the nodes of the tree shown in Fig. 3. The first element of each pair is the preorder number of the corresponding node and the second is its postorder number. With such labels, the ancestor-descendant relationships can be easily checked.

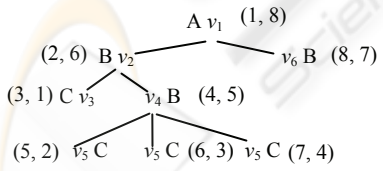

Figure 3: Illustration for tree encoding.

For instance, by checking the label associated with v_2 against the label for v_6, we see that v_2 is an ancestor of v_6 in terms of Proposition 1. Note that v_2's label is (2, 6) and v_6's label is (6, 3), and we have $2 < 6$ and $6 > 3$. We also see that since the pairs associated with v_8 and v_5 do not satisfy the condition given in Proposition 1, v_8 must not be an ancestor of v_5 and *vice versa*.

Definition 2. (*label pair subsumption*) Let (p, q) and (p', q') be two pairs associated with nodes u and v. We say that (p, q) is subsumed by (p', q'), denoted $(p, q) \sqsubset (p', q')$, if $p > p'$ and $q < q'$. Then, u is a descendant of v if (p, q) is subsumed by (p', q').

In the following, we also use $T[v]$ to represent a subtree rooted at v in T.

3.2 Algorithm for Twig Pattern Matching

Now we discuss our algorithm for twig pattern matching. During the process, both T and Q are searched bottom-up. That is, the nodes in T and Q will be accessed along their postorder numbers. Therefore, for convenience, we refer to the nodes in T and Q by their postorder numbers, instead of their node names.

In each step, we will check a node j in T against all the nodes i in Q.

In order to know whether $Q[i]$ can be embedded into $T[i]$, we will check whether the following two conditions are satisfied.

1. label(j) = label(i).
2. Let $i_1, ..., i_k$ be the child nodes of i. For each i_a ($a = 1, ..., k$), if (i, i_a) is a c-edge, there exists a child node j_b of j such that $T[j_b]$ contains $Q[i_a]$; if (i, i_a) is a d-edge, there is a descendent j' of j such that $T[j']$ contains $Q[i_a]$.

To facilitate this process, we will associate each j in T with a set of nodes in Q: $\{i_1, ..., i_j\}$ such that for each $i_a \in \{i_1, ..., i_j\}$ $Q[i_a]$ can be root-preservingly embedded into $T[j]$. This set is denoted as $M(j)$. In addition, each i in Q is associated with a value $\beta(i)$, defined as below.

i) Initially, $\beta(i)$ is set to ϕ.
ii) During the computation process, $\beta(i)$ is dynamically changed. Concretely, each time we meet a node j in T, if i appears in $M(j_b)$ for some child node j_b of j, then $\beta(i)$ is changed to j.

In terms of above discussion, we give the following algorithm.

Algorithm *tree-matching*(T, Q)
Input: tree T (with nodes 0, 1, ..., $|T|$) and tree Q (with nodes 1, ..., $|Q|$)
Output: a set of nodes j in T such that $T[j]$ contains Q.
begin
1. **for** $j := 1, ..., |T|$ **do**
2. {let $j_1, ..., j_k$ be the children of j;
3. **for** $l := 1, ..., k$ **do**
4. {**for** each $i' \in M(j_l)$ **do** $\beta(i') \leftarrow j$;
5. remove $M(j_l)$;}

5. **for** i := 1, ..., $|Q|$ **do**
6. **if** label(i) = label(j) **then**
7. {let i_1, ..., i_g be the children of i;
8. **if** for each i_l (l = 1, ..., g) we have
9. (i, i_l) is a c-edge and $\beta(i_l) = j$, or
10. (i, i_l) is a d-edge and $\beta(i_l)$ is subsumed by j;
11. **then** {insert i into $M(j)$;
12. **if** i is the root of Q, **then** report the subtree rooted at j as an answer;}
13. }

end

In the above algorithm, each time we meet an j in T, we will establish the new β values for all those nodes of Q, which appear in $M(j_1)$, ..., $M(j_k)$, where j_1, ..., j_k represent the child nodes of j (see lines 1 - 4). Then, all $M(j_l)$'s (l = 1, ..., k) are removed. In a next step, we will check j against all the nodes i in Q (see lines 5 - 13). If label(i) = label(j), we will check $\beta(i_1)$, ..., $\beta(i_g)$, where i_1, ..., i_g are the child nodes of i. If (i, i_l) ($l \in \{1, ..., g\}$) is a c-edge, we need to check whether $\beta(i_l) = j$ (see line 9). If (i, i_l) ($l \in \{1, ..., g\}$) is a d-edge, we simply check whether $\beta(i_l)$ is subsumed by j (see line 10). If all the child nodes of i survive the above checking, we get a root-preserving embedding of the subtree rooted at i into the subtree rooted at j. In this case, we will insert j into $M(j)$ (see line 11) and report j as one of the answers if i is the root of Q (see line 12).

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

1. The first part is the time spent on generating β values (see lines 2 - 5). For each node j in T, we will access $M(j_l)$ for each child node j_l of j. Therefore, this part of cost is bounded by
$$O(\sum_{j=1}^{|T|} d_j \cdot |M(j)|) \le O(\sum_{j=1}^{|T|} d_j \cdot |Q|) = O(|T| \cdot |Q|),$$
where d_j is the outdegree of j.
2. The second part is the time used for constructing $M(j)$'s. For each node j in T, we need $O(\sum_i c_i)$ time to do the task, where c_i is the outdegree of i in Q, which matches j. So this part of cost is bounded by
$$O(\sum_j \sum_i c_i) \le O(\sum_{j=1}^{|T|} |Q|) = O(|T| \cdot |Q|).$$

The space overhead of the algorithm is easy to analyze. During the processing, each j in T will be associated with a $M(j)$. But $M(j)$ will be removed later once j's parent is encountered and for each $i \in M(j)$ its β value is changed. Therefore, the total space is bounded by $O(leaf_T|Q| + |T| + |Q|)$, where $leaf_T$ represents the number of the leaf nodes of T. It is because at any time point for any two nodes on the same path in T only one is associated with a M.

4 GENERAL CASES

In this section, we extend the algorithm discussed in the previous section to handle queries containing '$\wedge$', '$\vee$' and '$\neg$' logic operators.

Without loss of generality, we assume that in an XPath expression a predicate is a path, or a conjunctive normal form. As an example, consider the following XPath expression:

a[*b*[*c* and .//*f*]]/*b*[*c* or *e*//*]/*g*[not *c*].

This expression can be represented as an And-Or tree Q shown in Fig. 4.

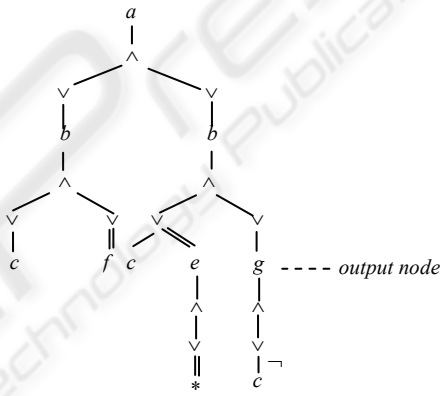

Figure 4: A query tree with different logic operators.

In such a tree, we distinguish between two kinds of nodes:

- name nodes: nodes corresponding to the node test.
- operator nodes: nodes labeled with $\wedge$ or $\vee$.

As with a simple twig pattern, it may contain two kinds of edges: /-edges and //-edges; but an edge may be labeled with '$\neg$'. If an edge (q, q') is labeled with '$\neg$', q' is called a negative node; otherwise, q' is called a positive node.

In an And-Or tree Q, the following conditions always hold:

1. The child nodes of any $\vee$-node are name nodes.
2. The child nodes of any $\wedge$-node are $\vee$-nodes.
3. Any name node has no children or has only one node which is a $\wedge$-node.

According to the above properties, the tree embedding of Q into a document tree T can be defined as follows.

Let i be a node in Q with child nodes i_1, ..., i_k. Let j be a node in T with child nodes j_1, ..., j_l.

(i) If i is a $\vee$-node, $T[j]$ contains $Q[i]$ if one of the following conditions holds:
- There exists a positive //-child q_a ($1 \le a \le k$) such that $T[j]$ contains $Q[i_a]$.
- There exists a positive /-child i_a ($1 \le a \le k$) such that $T[j]$ contains $Q[i_a]$ and $label(j) = label(i_a)$.
- There exists a negative //-child i_a ($1 \le a \le k$) such that $T[j]$ does not contain $Q[i_a]$.
- There exists a negative /-child i_a ($1 \le a \le k$) such that $T[j]$ does not contain $Q[i_a]$ or $T[j]$ contains $Q[i_a]$ but $label(j) \neq label(i_a)$.

(ii) If i is a $\wedge$-node, $T[j]$ contains $Q[i]$ if the following conditions hold:
- for every positive node i_a ($1 \le a \le k$), there exists a j_b ($1 \le b \le l$) such that $T[j_b]$ contains $Q[i_a]$.
- for every nagative node i_a ($1 \le a \le k$), there exists no j_b ($1 \le b \le l$) such that $T[j_b]$ contains $Q[i_a]$.

(iii) If i is a name node, $T[j]$ contains $Q[i]$ if the following conditions hold:
- $T[j]$ contains $Q[i_1]$ (i has only one child node i_1.)
- $label(j) = label(i)$.

In the following, we give an algorithm to check the embedding of an And-Or tree Q into a document tree T. For this purpose, we associate with each j in T two sets: (j) and $H(j)$. $F(j)$ contains all those name nodes i in Q such that $Q[i]$ can be imbedded into $T[j]$; and $H(j)$. contains all those $\vee$-nodes i in Q such that $Q[i]$ can be imbedded into $T[j]$. Besides, in order to calculate $H(j)$, we maintain an array N_Q containing all the negative nodes in Q.

With $F(j)$ and $H(j)$, we design our general algorithm, in which three functions are called:

- *general-node-check*(j, i): It checks whether $T[j]$ contains $Q[i]$. If it is the case, return $\{i\}$. Otherwise, it returns an empty set $\varnothing$.
- *leaf-node-check*(j): It returns a set of leaf nodes in Q: $\{i_1, ..., i_k\}$ such that for each i_a ($1 \le a \le k$) $label(j) = label(i_a)$.
- *calculate-H*(j, $F(j)$): It compute $H(j)$ based on $F(j)$ and N_Q. It is done exactly according to the conditions given above for checking $\vee$-node containment. Especially, in the presence of '$\neg$', we have to check each negative node in N_Q to see whether it appears in $F(j)$.~

Algorithm *general-tree-embedding*(v)
Input: tree T (with nodes 0, 1, ..., $|T|$) and tree Q (with nodes 1, ..., $|Q|$)
Output: a set of nodes j in T such that $T[j]$ contains Q.

begin
1. **for** j := 1, ..., $|T|$ **do**
2. {**if** j is not a leaf node in T **then**
3. {let j_1, ..., j_k be the children of j;
3. **for** l := 1, ..., k **do**
4. {**for** each $i' \in H(j_l)$ **do** $\beta(i') \leftarrow j$;}
5. $F \leftarrow merge(F(j_1), ..., F(j_k))$; (*See (Chen, 2007) for the definition of the merge operation.*)
6. assume that $F = \{ i_1, ..., i_c\}$;
7. $S_1 := \varnothing$; $S_2 := \varnothing$;
5. **for** i := 1, ..., $|Q|$ **do**
6. $S_1 := S_1 \cup general\text{-}node\text{-}check(j, i)$;
15. }
16. $S_2 := leaf\text{-}node\text{-}check(j)$;
17. $F(j) := merge(F, S_1, S_2)$;
18. call *calculate-H*(j, $F(j)$);
end

Function *leaf-node-check*(j)
begin
1. $S_2 := \varnothing$;
2. **for** each leaf node i in Q **do**
3. {**if** $label(i) = label(j)$ **then** $\{S_2 := S_2 \cup \{i\}$;
4. **if** i is *root* **then** mark j;}
5. return S_2;
end

Function *general-node-check*(j, i)
begin
1. $S_1 := \varnothing$;
2. **if** *label*(i's parent) = $label(j)$ **then**
(*If i is *, the checking is always successful.*)
3. { let i_1, ..., i_k be the child nodes of i;
4. **if** for each i_a (a = 1, ..., k) $\beta(i_a)$ is equal to j
5. **then** $\{S_1 := \{i\}$;
6. **if** i's parent is *root* **then** mark j;}}
7. return S_1;
end

Function *calculate-H*(j, F)
begin
1. $H := \varnothing$; $A := \varnothing$;
2. **for** each $i \in F$ do {
3. **if** ((i is a /-child and $label(i) = label(j)$) or
4. i is a //-child)
5. **then** $H := H \cup \{i$'s parent$\}$);
6. }
7. **for** each $i' \in N_Q$ **do** {
8. **if** ($i' \notin F$ or ($i' \in F$ and
i' is a /-child with $label(i') \neq label(j)$))
9. **then** $A := A \cup \{i$'s parent$\}$;}
10. return merge(H, A);
end

Algorithm *general-tree-embedding*( ) is similar to Algorithm *tree-embedding*( ). The only difference is that $M(j)$ in *tree-embedding*( ) is replaced with $F(j)$

in *general-tree-embedding*( ). In $F(j)$, we maintain all those query nodes i such that $Q[i]$ can be embedded (not only root-preservingly embedded) in $T[j]$. Although more time is needed for this, the whole time complexity remains unchanged. See line 5, in which the merge operation is first introduced in (Chen, 2007). The time complexity of *merge*($F(j_1)$, ..., $F(j_k)$) is bounded by $O(k \cdot leaf_Q)$.

Special attention should be paid to Function *general-node-check*( ). It is used to check $\wedge$-nodes in Q. Since each name node has only one $\wedge$-node as its child, the checking of name nodes is integrated into this process to simplify the procedure (see line 2 in this function.)

In Function *calculate-H*(j, $F(j)$), we compute $H(j)$ based on $F(j)$. It is done exactly according to the conditions given above for checking $\vee$-node containment. Especially, in the presence of '$\neg$', we have to check each negative node in N_Q to see whether it appears in $F(j)$. (see lines 7 - 9 in this function). It needs $O(|N_Q| \cdot \log|F(j)|)$ time. So the total time of the algorithm is bounded by $O(|T| \cdot leaf_Q + |N_Q| \cdot |T| \cdot \log leaf_Q)$.

5 CONCLUSIONS

In this paper, a new algorithm is proposed to evaluate twig pattern queries in XML document databases. The algorithm works in a bottom-up way, by which an important property of the postorder numbering is used to avoid join or join-like operations. The time complexity and the space complexity of the algorithm are bounded by $O(|T| \cdot |Q|)$ and $O(|Q| \cdot leaf_T)$, respectively, where T is the document tree and Q the twig pattern query, and $leaf_T$ represents the number of leaf nodes in T. Experiments have been done to compare our method with some existing strategies, which demonstrates that our method is highly promising in evaluating twig pattern queries.

ACKNOWLEDGEMENTS

The work is supported by NSERC 239074-01 (242523) (Natural Science and Engineering Council of Canada).

REFERENCES

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

N. Bruno, N. Koudas, and D. Srivastava (2002) Holistic Twig Hoins: Optimal XML Pattern Matching, in *Proc. SIGMOD Int. Conf. on Management of Data*, Madison, Wisconsin, June 2002, pp. 310-321.

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

S. Chen et al. (2006). *Twig²Stack*: Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents, in *Proc. VLDB*, Seoul, Korea, Sept. 2006, pp. 283-323.

Y. Chen (2007). A New Algorithm for Tree Mapping in XML Databases, in Proc. of the Internet and Multimedia Systems and Applications Conference (IMSA 2007), Honolulu, Hawaii, USA.

B.F. Cooper, N. Sample (2001). M. Franklin, A.B. Hialtason, and M. Shadmon, A fast index for semistructured data, in: *Proc. VLDB*, Sept. 2001, pp. 341-350.

R. Goldman and J. Widom (1997). DataGuide: Enable query formulation and optimization in semistructured databases, in: *Proc. VLDB*, Aug. 1997, pp. 436-445.

G. Gottlob, C. Koch, and R. Pichler (2005). Efficient Algorithms for Processing XPath Queries, *ACM Transaction on Database Systems*, Vol. 30, No. 2, June 2005, pp. 444-491.

C.M. Hoffmann and M.J. O'Donnell (1982). Pattern matching in trees, *J. ACM*, 29(1):68-95, 1982.

Q. Li and B. Moon (2001) Indexing and Querying XML data for regular path expressions, in: *Proc. VLDB*, Sept. 2001, pp. 361-370.

J. Lu, T.W. Ling, C.Y. Chan, and T. Chan (2005). From Region Encoding to Extended Dewey: on Efficient Processing of XML Twig Pattern Matching, in: *Proc. VLDB*, pp. 193 - 204, 2005.

G. Miklau and D. Suciu (2004) Containment and Equivalence of a Fragment of XPath, *J. ACM*, 51(1):2-45, 2004.

H. Wang, S. Park, W. Fan, and P.S. Yu (2003) ViST: A Dynamic Index Method for Querying XML Data by Tree Structures, *SIGMOD Int. Conf. on Management of Data*, San Diego, CA., June 2003.

H. Wang and X. Meng (2005), On the Sequencing of Tree Structures for XML Indexing, in *Proc. Conf. Data Engineering*, Tokyo, Japan, April, 2005, pp. 372-385.

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