Unordered Tree Matching and Strict Unordered Tree Matching: the Evaluation of Tree Pattern Queries

Yangjun Chen, Donovan Cooke

Dept. Applied Computer Science, University of Winnipeg

515 Portage Ave., Winnipeg, Manitoba, Canada R3B 2E9

y.chen@uwinnipeg.ca

***Abstract* — In this paper, we consider two kinds of *unordered tree matchings* for evaluating tree pattern queries in XML databases. For the first kind of unordered tree matching, we propose a new algorithm, which runs in O(|*D*||*Q*|) time, where *Q* is a tree pattern and *D* is a largest data stream associated with a node of *Q*. It can also be adapted to an indexing environment with XB-trees being used to speed up disk access. Experiments have been conducted, showing that the new algorithm is promising. For the second of tree matching, the so-called *strict unordered tree matching*, we show that the problm is *NP-complete* by a reduction from the satisfiability problem.**

Key words: *Tree mapping, twig pattern, XML databases, query evaluation, tree encoding*

I. INTRODUCTION

In XML [33, 34], data are represented as a tree; associated with each node of the tree is an element name tag 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 [15], XQuery [34], XML-QL [14], and Quilt [6, 7]), we express queries as tree 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 Fig. 1.

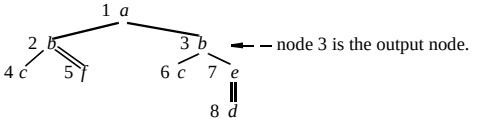

Fig. 1. A query tree

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 following XPath expression:

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

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 tree patterns, 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 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 label*: For each $u \in Q$, *label*(*u*) = *label*(*f*(*u*)).

(ii) 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*.

This definition allows a path to match a tree as illustrated in Fig. 2.

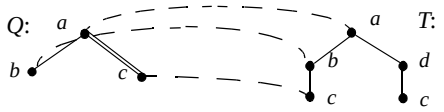

Fig. 2. A tree matches a path

It even allows to map several nodes with the same tag name in a query to the same node in a database. In fact, it is a kind of unordered tree matching, by which the order of siblings is not significant. Almost all the existing strategies for evaluating tree pattern queries are designed according to this definition [5, 9, 10, 11, 12, 14, 22, 24, 26, 27, 28, 29, 34, 35]. Such kind of tree matchings can be solved in polynomial time.

However, if we require that each query node in *Q* maps to a different document node in *T* and no siblings map to those nodes which are related by ancestor/descendant or parent/child relationships, the problem becomes very difficult. We refer to it as *strict unordered tree matching*.

In this paper, we first present a new algorithm for evaluating tree pattern queries according to Definition 1, which runs in O(|*D*|·|*Q*|) time and O(|*D*|·|*Q*|) space, and can adapted to an indexing environment with XB-trees being used, where *D* is a

largest data stream associated with a node q of Q. Then, we show that the strict unordered tree matching is *NP-complete*.

The remainder of the paper is organized as follows. In Section 2, we restate a tree encoding [36], which facilitates the recognition of different relationships among the nodes of a tree. In Section 3, we discuss our algorithm for the unordered tree matching according to Definition 1. In Section 4, we show the *NP-completeness* of the strict unordered tree matching. Section 6 is devoted to the implementation and experiments. Finally, a short conclusion is set forth in Section 6.

II. TREE ENCODING

In [36], an interesting tree encoding method was discussed, which can be used to identify different relationships among the nodes of a tree.

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 v, respectively; and *LevelNum* is the nesting depth of v in the document. (See Fig. 3 for illustration.) By using such a data structure, the structural relationship between the nodes in an XML database can be simply determined [36]:

(i) *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 $r_1 > r_2$.

(ii) *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 $ln_2 = ln_1 + 1$.

(iii) *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 $d_1 = d_2$, $r_1 < l_2$.

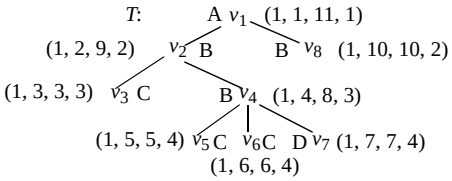

Fig. 3. Illustration for tree encoding

In Fig. 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 $\alpha_1 = (d_1, l_1, r_1, ln_1)$ and $\alpha_2 = (d_2, l_2, r_2, ln_2)$, we have $d_1 = d_2$, $l_1 < l_2$, and $r_1 > r_2$, we say that α_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 $\alpha(v)$ interchangeably.

We can also assign LeftPos and RightPos values to the query nodes in Q for the same purpose as above.
Finally we use $T[v]$ to represent a subtree rooted at v in T.

III. MAIN ALGORITHM

In this section, we discuss our algorithm according to Definition 1.

As with *TwigStack* [5], each node q in a tree pattern (or say, a query tree) Q is associated with a data stream $B(q)$, which contains the positional representations (quadruples) of the database nodes v that match q (i.e., $label(v) = label(q)$). All the quadruples in a data stream are sorted by their (DocID, LeftPos) values. For example, in Fig. 4, we show a query tree containing 5 nodes and 4 edges and each node is associated with a list of matching nodes of the document tree shown in Fig. 3, sorted according to their (DocID, LeftPos) values. For simplicity, we use the node names in a list, instead of the node's quadruples.

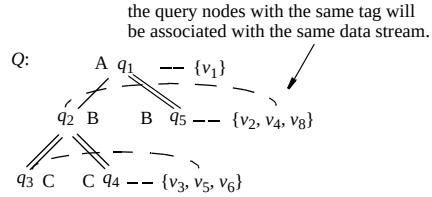

Fig. 4. Illustration for $L(q_i)$'s

Therefore, if each time we choose a node with the least LeftPos from data streams, T is in fact traversed in *preorder* (top-down). However, our algorithm needs to visit the tree nodes in *postorder* (bottom-up). For this purpose, we maintain a global stack S to make a data stream transformation as described in Algorithm *stream-transformation*( ) shown in Fig. 5. In S, each entry is a pair (q, v) with $q \in Q$ and $v \in T$.

Algorithm *stream-transformation*($B(q_i)$'s)
input: all data streams $B(q_i)$'s, each sorted by LeftPos.
output: new data streams $L(q_i)$'s, each sorted by RightPos.
begin
1. **repeat until** each $B(q_i)$ becomes empty
2. {identify q_i such that the first element v of $B(q_i)$ is of the minimal LeftPos value; remove v from $B(q_i)$;
3. **while** S is not empty and $S.top$ is not v's ancestor **do**
4. {$x \leftarrow S.pop()$; Let $x = (q_j, u)$;
5. put u at the end of $L(q_i)$; }
6. $S.push(q_i, v)$;
7. }
end

Fig. 5. Algorithm for stream transformation

In the algorithm, the stack S is used to keep all the nodes on a document path until we meet a node v that is not a descendant of $S.top$ (see line 3). Then, we pop up all those nodes that are not v's ancestor, and push v into S (see lines 4 - 5). The output of the algorithm is a set of data streams $L(q_i)$'s with each being sorted by (DocID, RightPos). But we notice that the popped nodes themselves are in postorder (see line 3). So we can handle the nodes in this order without explicitly generating $L(q_i)$'s. However, in the following discussion, we assume that all $L(q_i)$'s are completely generated for ease of explanation. We also note that the data streams associated with different nodes in Q may be the same. So we use $\boldsymbol{q}$ to represent the set of such query nodes and denote by $L(\boldsymbol{q})$ the data stream shared by them. For example, in Q shown in Fig. 4, $L(\{q_2, q_5\}) = \{v_4, v_2, v_8\}$. Without loss of generality, assume that the query nodes in $\boldsymbol{q}$ are sort-

ed by their RightPos values.

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 a common data stream.

We first observe that iterating through $L(\boldsymbol{q}_1), ..., L(\boldsymbol{q}_l)$ corresponds to a navigation of T in postorder. However, in this process, any node v in T, which does not match any $q \in Q$ (i.e., $label(v) \neq label(q)$), is not accessed. So only a subtree T' of T is navigated. If we are able to construct T' explicitly in this process, we will get a tree structure with each node v associated with a query node stream $S(v)$, as illustrated in Fig. 6. For each $q \in S(v)$, we have $label(v) = label(q)$.

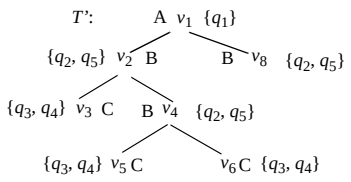

Fig. 6. Illustration for generating *QS*'s

If we check, before a q is inserted into the corresponding $S(v)$, whether $Q[q]$ (the subtree rooted at q) can be embedded into $T'[v]$, we get in fact an algorithm for tree pattern matching. The challenge is how to conduct such a checking efficiently.

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 $a_0, a_1, ..., a_m$ for some m in sequence, where $a_0 = \phi$ and a_i's (i = 1, ..., 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, ..., m) when the following conditions are satisfied.

i) v is the node currently encountered.
ii) q appears in $S(u)$ for some child node u of v.
iii) 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 $S(v)$, we will do the following checking:

1. Let $q_1, ..., q_k$ be the child nodes of q.
2. If for each q_i (i = 1, ..., k), $\chi(q_i)$ is equal to v and $label(v) = label(q)$, insert q into $S(v)$.

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

In terms of the above discussion, we give our algorithm for evaluating tree pattern queries. The algorithm mainly consists of a main procedure and a subprocedure. The task of the main procedure is to construct T' while the subprocedure is invoked to check tree embedding. In the main procedure, each node that is created for a quadruple v from a $L(\boldsymbol{q})$ is associated with two links, denoted respectively *left-sibling*(v) and *parent*(v), to mainly the tree structure of T' as follows:

1. Identify a data stream $L(\boldsymbol{q})$ with the first element being of the minimal RightPos value. Choose the first element v of $L(\boldsymbol{q})$. Remove v from $L(\boldsymbol{q})$.
2. Generate a node for v.
3. If v is not the first node, we do the following:
Let v' be the node created just before v. If v' is not a child (descendant) of v, create a link from v to v', called a *left-sibling* link and denoted as *left-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 left-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. Finally, set *left-sibling*(v) ← v''.

Fig. 7 is a pictorial illustration of this process.

In Fig. 7(a), we show the navigation along a left-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. Fig. 8(b) shows that the left-sibling link of v is set to v'', which is previously pointed to by the left-sibling link of v's left-most child.

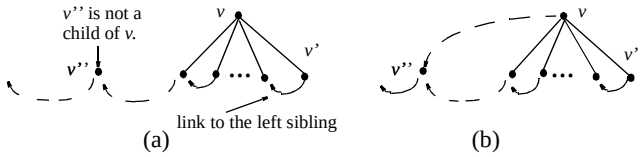

Fig. 7. Illustration for the construction of a matching subtree

In Fig. 8, we give the main algorithm, by which a quadruple is removed in turn from the data streams $L(\boldsymbol{q})$'s and a node v for it is generated and inserted into T'.

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.

The algorithm is designed for queries containing /-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 $S(v)$, where $\boldsymbol{q}$ is a subset of query nodes such that $L(\boldsymbol{q})$ contains v. Let v_1, v_2 be two children of a node v. Let $S_1 = S(v_1)$ and $S_2 = S(v_2)$. *merge*(S_1, S_2) puts S_1 and S_2 together with any duplicate being removed. Since both S_1 and S_2 are sorted by RightPos values, *merge*(S_1, S_2) works in a way like the *sort-merge join* and needs only O($max\{|S_1|, |S_2|\}$) time. We define *merge*(S_1, ..., S_{k-1}, S_k) to be *merge*(*merge*(S_1, ..., S_{k-1}), S_k).

The output of *tree-matching*( ) is D_{root} and D_{output}. Based on them, we can find all the answers by generating a subtree in a way similar to the construction of T'.

Algorithm *tree-matching*( ) 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 $S(v')$, where v' is a child node of v, satisfying the conditions i) - iii) given above. In lines 24 - 28, we use the merging operation to construct $S(v)$. In Function *subsumption-check*( ), we check whether any q in $\boldsymbol{q}$ can be inserted into S 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 - 8). In the Appendix, we prove the correctness of *tree-matching*( ).

```
Algorithm tree-matching(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 first node v of L(q) is of the minimal
   RightPos value; remove v from L(q); generate node v;
3. if v is the first node created then
4. {S(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.   {left-sibling(v) ← v';
9.    S(v) ← subsumption-check(v, q); }
10. else
11. {v'' ← v'; w ← v';
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'' ← left-sibling(v'');
20.  remove left-sibling(w);
21. }
22. left-sibling(v) ← v'';
23. }
24. S ← subsumption-check(v, q);
25. let v_1, ..., v_j be the child nodes of v;
26. S' ← merge(S(v_1), ..., S(v_j));
27. remove S(v_1), ..., S(v_j);
28. S(v) ← merge(S, S');
29. }}
end
```

```
Function subsumption-check(v, q)
(*v satisfies the node name test at each q in q.*)
begin
1. S ← Φ;
2. for each q in q do {
3. let q_1, ..., q_j be the child nodes of q.
4. if for each /-child q_i χ(q_i) = v and for each //-child q_i
     χ(q_i) is subsumed by v then
5. {S ← S ∪ {q};
6.  if q is the root of Q then
7.  D_root ← D_root ∪ {v};
8.  if q is the output node then D_output ← D_output ∪ {v};}}
9.  return S;
end
```

Fig. 8. Algorithm *tree-matching*

The algorithm handles wildcards in the same way as any non-wildcard nodes. But a wildcard matches any tag name. Therefore, $L(*)$ should contain all the nodes in T. However, as with *twigStack* [5], we establish an XB-tree over $B(q)$'s and take an element from it as it is needed. Recall that the input of our algorithm is in fact $B(q)$'s which are transformed to $L(q)$'s by using a global stack (see Fig. 5).

Example 2 Applying Algorithm *tree-matching* to the data streams shown in Fig. 4, we will find that the document tree shown in Fig. 3 contains the query tree shown in Fig. 4. We trace the computation process as shown in Fig. 9.

In the first three steps, we will generate part of the matching subtree as shown in Fig. 9(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 Fig. 9(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 Fig. 9(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 Fig. 9(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 Fig. 9(e)), leading to the insertion of q_1 into $QS(v_1)$. □

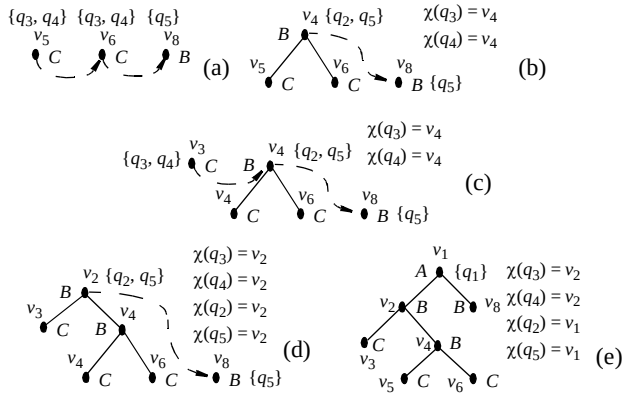

Fig. 9. Sample trace

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.RightPos > v_2.RightPos > v_1.RightPos. 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: i) v_2 is to the right of v_1, and ii) v_2 is an ancestor of v_1. In case (i), we have v_1.LeftPos < v_2.LeftPos. So we have v_3.LeftPos < v_1.LeftPos < v_2.LeftPos. This shows that v_2 is a descendant of v_3. In case (ii), 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 Fig. 10, which is helpful for understanding the proof of Proposition 1 given below.

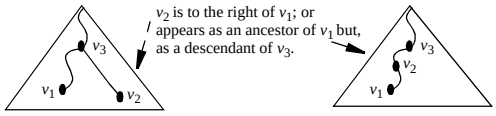

Fig. 10. Illustration for Lemma 1

Proposition 1 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 *tree-matching*( ). 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 *tree-matching*( ).)

(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 *tree-matching*( ).)

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 $h \le k$. Consider the nodes v at height $h = k + 1$. Assume 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 (i) $\chi(q_i) = \phi$, or (ii) $\chi(q_i)$ is not subsumed by v when q is checked against v. Obviously, case (i) 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 (ii) 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 $v_1, ..., v_k$ for some $k \ge 1$ with v_1.RightPos < v_2.RightPos < ... < v_k.RightPos such that q_i appears in $QS(v_1), ..., QS(v_k)$. In terms of the induction hypothesis, $v' = v_j$ for some $j \in \{1, ..., k\}$. Let l be the largest integer $\le k$ such that v_l.LeftPos > v.LeftPos. Then, for each v_p ($j \le p \le l$), we have

v'.RightPos ≤ v_l.RightPos < v.RightPos.

In terms of Lemma 1, each v_p ($j \le p \le l$) 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 (ii). The above explanation shows that case (ii) 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 [5], as well as the result enumeration [12], can be completely avoided. More importantly, the theoretical time complexity is reduced by one order of magnitude.

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 $O(|D|\cdot|Q|)$.
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 $O(\sum_i c_{j_i})$ 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 $O(\sum_j \sum_i c_{j_i}) \le O(|D| \cdot \sum_k^{|Q|} c_k) = O(|D|\cdot|Q|)$.
3. The third part is the time for establishing χ values, which is the same as the second part since for each q in a $QS(v)$ its χ value is assigned only once.

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

The space overhead of the algorithm is easy to analyze. 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 $O(|D|\cdot|Q| + |D| + |Q|) = O(|D|\cdot|Q|)$.

VI. About Strict Unordered Tree Matching

Definition 2 A strict embedding of a tree pattern Q into an XML document T is a mapping $f: Q \to T$, from the nodes of Q to the nodes of T, which satisfies the following conditions:

(i) same as (i) in Definition 1.

(ii) same as (ii) in Definition 1.

(iii) For any two nodes $v_1 \in Q$ and $v_2 \in Q$, if v_1 and v_2 are not related by an ancestor-descendant relationship, then $f(v_1)$ and $f(v_2)$ in T are not related by an ancestor-descendant relationship. □

This is a much more difficult problem than the tree matching discussed in the previous section.

To facilitate the algorithm description, we will use some concepts from the hypergraph theory [1].

Definition 3 Let $U = \{u_1, \ldots, u_n\}$ be a finite set of nodes. A hypergraph on U is a family $H = \{E_1, \ldots, E_l\}$ of subsets of U such that

(1) $E_i \ne \varnothing$ ($i = 1, 2, \ldots, l$)

(2) $\bigcup_{i=1}^{l} E_i = U$.

A simple hypergraph (or Sperner family) is a hypergraph H = $\{E_1, \ldots, E_l\}$ such that

(3) $E_i \subset E_j \Rightarrow i = j$. □

As for a graph H, the order of H, denoted by $n(H)$, is the number of nodes. The number of edges will be denoted by $m(H)$ and the $rank(H)$ is defined to be $r(H) = max_j|E_j|$. It can be proved that $m(H) \le \binom{n}{\lfloor n/2 \rfloor}$ [1].

Let $A \overline{\subset} U$ be a subset. We call the family

$H_A = \{E_i \cap A \mid 1 \le i \le l, E_i \cap A \ne \varnothing\}$

the sub-hypergraph induced by A.

Definition 4 Let $H = \{E_1, \ldots, E_l\}$ be a hypergraph on U and $H' = \{F_1, \ldots, F_{l'}\}$ be another hypergraph on V. The product of H and H', denoted as $H \times H'$, is a hypergraph, whose nodes are the elements of the Cartesian product $U \times V$, and whose edges are the sets $E_i \times F_j$ with $1 \le i \le l$ and $1 \le j \le l'$. Obviously, $n(H \times H') = n(H)n(H')$ and $m(H \times H') = m(H)m(H')$. However, if $U = V$, we have $n(H \times H') = n(H)$ and $m(H \times H') \le \binom{n}{\lfloor n/2 \rfloor}$. □

As with the ordered tree embedding, we will maintain two matrices $Q(P, T)$ and $S(P, T)$ to control the computation.

1. In $Q(P, T)$, an entry q_{ij} is 1 if the subtree rooted at j in T includes the subtree rooted at i in P. Otherwise, it is 0.
2. In $S(P, T)$, each entry s_{ij} is defined as follows. Let $i_1, i_2, \ldots, i_k$ be the child nodes of i. s_{ij} is a hypergraph $H_i = \{E_1, \ldots, E_l\}$over $\{i_1, i_2, \ldots, i_k\}$ such that $T[j]$, the subtree rooted at j, includes each E_g ($g = 1, \ldots, l$}, that is, for each E_g, the subtree rooted at j includes all the subtrees rooted at the nodes in E_g. But $T[j]$ cannot include $E_i \cup E_j$ for any $i, j \in \{1, \ldots, l\}$.

Algorithm *unordered-embedding*(T, P)
Input: tree T (with nodes 1, ..., n) and tree P (with nodes 1, ..., m)
Output: $Q(P, T)$, which shows the tree embedding.
begin
1. **for** v := 1, ..., n **do**
2. {**for** u := 1, ..., m **do**
3. {**if** $q_{uv} = 0$ **then**
4. {let $v_1, \ldots, v_h$ be the child nodes of v;
5. $H := s_{uv_1} \times s_{uv_2} \times \ldots \times s_{uv_h}$;
6. let $u_1, \ldots, u_k$ be the child nodes of u;
7. **if** $\{u_1, \ldots, u_k\} \in H$ **then** set q_{uv} to 1;
8. **else** $s_{uv} := H_A$;
9. }
10 }
11. let $u_1, \ldots, u_l$ be nodes such that $P[u_1], \ldots, P[u_l]$ are included in $T[v]$;
12. for each ancestor v' of v, $q_{u_i v'} := 1$ for $i = 1, \ldots, l$;
13. construct hypergraph $H = \{\{u_1\}, \ldots, \{u_l\}\}$;
14. for u := 1, ..., m do
15. {let A be the set of u's child nodes;
16. $s_{uv} := H_A$;
17. }
20.}
End

The execution of line 5 will dominate the running time of the algorithm. Let k be the largest out-degree of any node in P. Then, the size of each s_{uv} is bounded by $\binom{k}{\lfloor k/2 \rfloor} < 2^k$. (is asymptotic to $(2/\pi)^{1/2}2^k k^{-1/2}$ [44]). Especially, the size of any product hypergraph of the form $s_{uv} \times s_{uv'}$ is bounded by 2^k (and so is $s_{uv} \times s_{uv'} \times s_{uv''}$, …, and so on.) So the time complexity of the algorithm is on the order of $O(|T| \cdot |P| \cdot 2^k)$.

V. EXPERIMENTS

In this section, we report the test results. We conducted our experiments on a DELL desktop PC equipped with Pentium(R) 4 CPU 2.80GHz, 0.99GB RAM and 20GB hard disk. The code was compiled using Microsoft Visual C++ compiler version 6.0, running standalone.

- *Tested methods*

In the experiments, we have tested four methods:

- *TwigStack* (*TS* for short) [5],
- *Twig²Stack* (*T²S* for short) [12],
- *Twig-List* (the method discussed in [15], *TL* for short),
- *One-Phase Holistic* (the method discussed in [42]; *OPH* for short),
- *Tree-matching* (the method dscussed in Section 3; *TM* for short),

and compare their execution times, as well as the runtime space usage. The theoretical computational complexities of these methods are summarized in Table 1.

Table 1. Time and space complexities

methods	query time	runtime space usage
TwigStack	$O(\|D\|^{\|Q\|})$	$O(\|D\|\|Q\|)$
Twig²Stack	$O(\|D\| \cdot \|Q\|^2 + \|subTwigResults\|)$	$O(\|D\| \cdot \|Q\|)$
OPH	$O(\|D\| \cdot \|Q\|^2)$	$O(\|D\| \cdot \|Q\|)$
TL	$O(\|D\| \cdot \|Q\|^2)$	$O(\|D\| \cdot \|Q\|)$
TM	$O(\|D\| \cdot \|Q\|)$	$O(\|D\| \cdot \|Q\|)$

Table 2. Data sets for experimental evaluation

	TreeBank	DBLP	XMark
Data size	82 (MB)	127 (MB)	113 (MB)
Number of nodes	2437k	3332k	1666k
Max/Avg. depth	36/7.9	6/2.9	12/5.5

- *Data*

The data sets used for the tests are TreeBank data set [30], DBLP data set [30] and a synthetic XMARK data set [35]. The TreeBank data set is a real data set with a narrow and deeply recursive structure that includes multiple recursive elements. The DBLP data set is another real data set with high similarity in structure. It is in fact a wide and shallow document. The XMark (with factor = 1) is a well-known benchmark data set. The important parameters of these data sets are summarized in Table 2.

- *Queries*

As we know, XPath allows for the formulation of straight-line queries as well as, in terms of XPath predicates, twigs that actually contain branches. XPath further allows the specification of value-based predicates. To study the performance impact of such characteristics, we have tested 10 queries against DBLP database, which are divided into two groups. In the first group all the 5 queries are with a constant while in the second group (another 5 queries) no parameter is specified. Over XMARK database, we have also tested 10 queries, divided into 2 groups with each containing 5 queries. In the first group, each query contains a constant. In the second group, for each query no constant is specified. All the queries are shown in Table 3 - Table 6.

Table 3: Group I - DBLP queries

Query	XPath Expression
Q1	//inproceedings [author]//year [text() = '2004']
Q2	//inproceedings [author and title]//year [text() = '2004']
Q3	//inproceedings [author and title and .//pages]//year [text() = '2004']
Q4	//inproceedings [author and title and .//pages and .//url] //year [text() = 2004']
Q5	//articles [author and title and .//volume and .//pages and //url]//year [text() = '2004']

Table 4: Group II - DBLP queries

Query	XPath Expression
Q6	//inproceedings[author/* and ./*]/year
Q7	//inproceedings[author/* and title and ./*]/year
Q8	//inproceedings[author/* and title and .//pages and ./*]/year
Q9	//inproceeding[author/* and title and .//pages and .//url and ./*]/year
Q10	//articles[author/* and title and .//volume and .//pages and .//pages and .//url and ./*]/year

Table 5: Group III - XMark queries

Query	XPath Expression
Q11	/site//open_auction[.//seller/person]//date [text() = '10/23/2006']
Q12	/site//open_auction[./seller/person and .//bidder]//date [text() = '10/23/2006']
Q13	/site//open_auction[./seller/person and /.bidder/increase] //date [text() = '10/23/2006']
Q14	/site//open_auction[./seller/person and .//bidder/increase and .//initial]//date [text() = '10/23/2006']
Q15	/site//open_auction[./seller/person and .//bidder/increase and //initial and .//description]//date [text() = '10/23/2006']

Table 6: Group IV - XMark queries

Query	XPath Expression
Q16	/site//open_auction[.//seller/person/* and ./*]/date
Q17	/site//open_auction[.//seller/person/* and .//bidder and ./*] /date
Q18	/site//open_auction[.//seller/person/* and .//bidder/increase] /date
Q19	/site//open_auction[.//seller/person/* and .//bidder/increase and .//initial and ./*]/date
Q20	/site//open_auction[.//seller/person/* and .//bidder/increase and .//initial and .//description and ./*]/date

- *Test results*

Now we demonstrate the execution times of all the four strategies when they are applied to the above queries.

In Fig. 11(a), we show the test results of the first group. From these we can see that our algorithm outperforms all the other strategies. It is because this algorithm works only in one scan of the data streams and neither the path join nor the result enumeration is involved. More importantly, for each element from an XB-tree, our algorithm only checks *QS*s for the child nodes of current query node. But *Twig*^{2}*Stack* needs to check all the stacks associated with all the query nodes. Both *OPH* and *TL* have the same problems *Twig*^{2}*Stack*, but work a little bit better than *Twig*^{2}*Stack*. *OPH* does no result enumeration is involved while *TL* does less checkings. *TwigStack* has the worst performance.

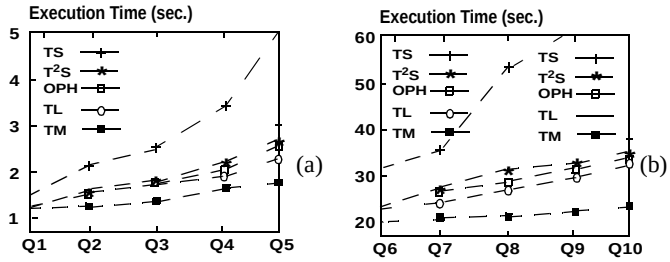

Fig. 11. Results of GroupI and II

Fig. 11(b) shows the test results of the second group. The execution time of all the strategies are much worse than Group 1 since the queries are all of quite low selectivity and thus almost all the data set has to be downloaded into main memory. In this case, I/O dominates the cost. Again, our algorithm has the best performance. Especially, when the size of queries becomes larger, this algorithm is 3 - 4 times better than *Twig*^{2}*Stack*, *TL* and *OPH*. First, the time for constructing a matching subtree is much less than that for constructing the hierarchical stacks. Secondly, the space used by our algorithm is much smaller than any of the three methods. *TwigStack* shows an exponential-time behavior since for each path in a query a great many matching paths will be produced and the cost of join operations increases exponentially.

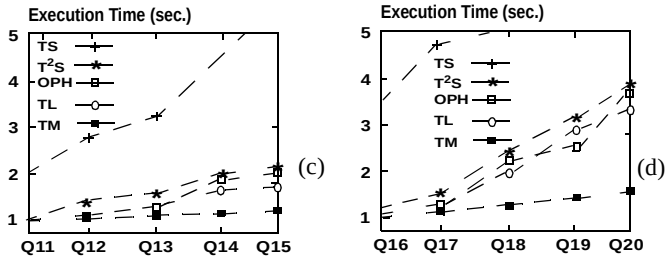

Fig. 12. Results of Group III and IV

In Fig. 12(a) and (b), the test results over the XMARK database are demonstrated. From these, we can see that our algorithm still has the best performance for this data set.

In Fig. 13, we compare the runtime memory usage of all the four tested approaches for the second group of queries.

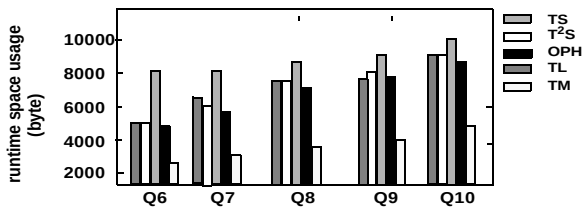

Fig. 13. Runtime memory usage of Group II

By the memory usage, we mean the intermediate data structures, not including data stream (concretely, path stacks

for *TwigStack*; hierarchical stacks for *Twig^{2}Stack*, *TL* and *OPH*; and *QS*s for ours.)

VI. CONCLUSION

In this paper, a new algorithms is presented to evaluate twig pattern queries based on unordered tree matching. The main idea is a process for tree reconstruction from data streams, during which each node *v* that matches a query node will be inserted into a tree structure and associated with a query node stream *QS*(*v*) such that for each node *q* in *QS*(*v*) *T*[*v*] embeds *Q*[*q*]. Especially, by using an important property of the tree encoding, this process can be done very efficiently, which enables us to reduce the time complexity of the existing methods (such as *Twig^{2}Stack* [12], *Twig-List* [15], and *One-Phase Holistic* [42]) by one order of magnitude. Our experiments demonstrate that the new algorithm is both effective and efficient for the evaluation of twig pattern queries.

REFERENCES

[1] C. Berge, Hypergraphs, Elsevier Science Publisher, Amsterdam, 1989.

[2] S. Abiteboul, P. Buneman, and D. Suciu, *Data on the web: from relations to semistructured data and XML*, Morgan Kaufmann Publisher, Los Altos, CA 94022, USA, 1999.

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

[4] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, and Y. Wu, Structural Joins: A primitive for efficient XML query pattern matching, in *Proc. of IEEE Int. Conf. on Data Engineering*, 2002.

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

[6] D. D. Chamberlin, J.Clark, D. Florescu and M. Stefanescu. "XQuery1.0: An XML Query Language," http://www.w3.org/TR/query-datamodel/.

[7] D. D. Chamberlin, J. Robie and D. Florescu. "Quilt: An XML Query Language for Heterogeneous Data Sources," *WebDB 2000*.

[8] T. Chen, J. Lu, and T.W. Ling, On Boosting Holism in XML Twig Pattern Matching, in: *Proc. SIGMOD*, 2005, pp. 455-466.

[9] B. Choi, M. Mahoui, and D. Wood, On the optimality of holistic algorithms for twig queries, in: *Proc. DEXA*, 2003, pp. 235-244.

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

[11] Y. Chen, S.B. Davison, Y. Zheng, An Efficient XPath Query Processor for XML Streams, in *Proc. ICDE*, Atlanta, USA, April 3-8, 2006.

[12] S. Chen, H-G. Li, J. Tatemura, W-P. Hsiung, D. Agrawa, and K.S. Canda, *Twig^{2}Stack*: Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents, in *Proc. VLDB*, Seoul, Korea, Sept. 2006, pp. 283-294.

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

[14] A. Deutch, M. Fernandez, D. Florescu, A. Levy, D.Suciu, A Query Language for XML, in: *Proc. 8th World Wide Web Conf.*, May 1999, pp. 77-91.

[15] Qin, L., Yu, J. X., and Ding, B., "TwigList: Make Twig Pattern Matching Fast," In Proc. 12th Int'l Conf. on Database Systems for Advanced Applications (DASFAA), pp. 850-862, Apr. 2007.

[16] G. Gou and R. Chirkova, Efficient Algorithms for Evaluating XPath over Streams, in: *Proc. SIGMOD*, June 12-14, 2007.

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

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

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

[20] C. Koch, Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata-based Approach, in: *Proc. VLDB*, Sept. 2003.

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

[22] J. McHugh, J. Widom, Query optimization for XML, in *Proc. of VLDB*, 1999.

[23] C. Seo, S. Lee, and H. Kim, An Efficient Index Technique for XML Documents Using RDBMS, *Information and Software Technology* 45(2003) 11-22, Elsevier Science B.V.

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

[25] Montanari, U., Networks of constraints: Fundamental properties and applications to pictureprocessing, *Inform. Sci.* **7** (1974) 95-132.

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

[27] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. Dewitt, and J.F. Naughton, Relational databases for querying XML documents: Limitations and opportunities, in *Proc. of VLDB*, 1999.

[28] U. of Washington, The Tukwila System, available from http://data.cs.washington.edu/integration/tukwila/.

[29] U. of Wisconsin, The Niagara System, available from http://www.cs.wisc.edu/niagara/.

[30] U of Washington XML Repository, available from http://www.cs.washington.edu/research/xmldatasets.

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

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

[33] World Wide Web Consortium. XML Path Language (XPath), W3C Recommendation, 2007. See http://www.w3.org/TR/xpath20.

[34] World Wide Web Consortium. XQuery 1.0: An XML Query Language, W3C Recommendation, Version 1.0, Jan. 2007. See http://www.w3.org/TR/xquery.

[35] XMARK: The XML-benchmark project, http://monetdb.cwi.nl/xml, 2002.

[36] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman, on Supporting containment queries in relational database management systems, in *Proc. of ACM SIGMOD*, 2001.

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

[38] A.R. Schmidt, F. Waas, M.L. Kersten, D. Florescu, I. Manolescu, M.J. Carey, and R. Busse, The XML benchmark project, Technical Report INS-Ro1o3, Centrum voor Wiskunde en Informatica, 2001.

[39] M. Götz, C. Koch, and W. Martens, Efficient Algorithms for the tree Homeomorphism Problem, in *Pro. Int. Symposium on Database Programming Language*, 2007.

[40] P. Ramanan, Holistic Join for Generalized Tree Patterns, *Information Systems* 32 (2007) 1018-1036.

[41] P. Rao and B. Moon, Sequencing XML Data and Query Twigs for Fast Pattern Matching, *ACM Transaction on Database Systems*, Vol. 31, No. 1, March 2006, pp. 299-345.

[42] Jiang, Z., Luo, C., Hou, W.-C., Zhu, Q., and Che, D., "Efficient Processing of XML Twig Pattern: A Novel One-Phase Holistic Solution," In Proc. the 18th Int'l Conf. on Database and Expert Systems Applications (DEXA), pp. 87-97, Sept. 2007.

[43] Z. Bar-Yossef, M. Fontoura, and V. Josifovski, On the memmory requirements of XPath evaluation over XML streams, *Journal of Computer and System Sciences* 73 (2007) 391-441.

[44] C. Beeri, M. Down, R. Fagin, and R. Statman, On the structure of Armstrong relations for functional dependencies, J. ACM, 31 (1984), pp. 30-46.