On the XML Data Stream and Xpath Queries

Yangjun Chen
University of Winnipeg, y.chen@uwinnipeg.ca

62F. On the XML Data Stream and Xpath Queries

Yangjun Chen
Dept. of Computer Science, University of Winnipeg
y.chen@uwinnipeg.ca

Abstract

With the growing importance of XML in data exchange, much research has been done in providing flexible query mechanisms to extract data from XML documents. In this paper, we focus on the query evaluation in an XML streaming environment, in which data streams arrive continuously and queries have to be evaluated even before all the data of an XML document is available. We will propose an algorithm for this issue, working in $O(|T|\cdot Q_{leaf})$ time and $O(|T|\cdot Q_{leaf})$ space, where T_{leaf} stands for the number of the leaf nodes in a document tree T and Q_{leaf} for the number of the leaf nodes in a query tree Q.

Keywords

XML databases, Trees, Paths, XML pattern matching, XML streams.

1. Introduction

XPath is a simple language for querying XML data. It has been used in many XML applications and also in some languages for querying an XML tree and returning a set of answer nodes, such as in XQuery (World Wide Web Consortium, 2005), XML-QL (Dutch et al., 1999), and Quilt (Chamberlin et al., 2000; Chamberlin et al., 2002).

In this paper, we consider a subset of XPath (World Wide Web Consortium, 2007) queries, which is frequently used in practice. This subset consists of node tests (or say element tag name tests), the child axis (/), the descendant axis (//), wildcards (*), and predicates (or filter, denoted [...]).
We call this class of queries $XP^{\{/,//,*,[]\}}$. Figure 1 shows its grammar.

Path	:=	Step | Path Step
Step	:=	Axis | NodeTest | Axis NodeTest '[' Predicate ']'
Axis	:=	'/' | '//'
NoteTest	:=	name | '*'
Predicate	:=	Path | Path ComOp Constant | Predicate '$\wedge$' Predicate | Predicate '$\vee$' Predicate | '$\neg$' Predicate
ComOp	:=	'=' | '!=' | '>' | '>=' | '<' | '<='

Figure 1. Language grammar

In the absence of '$\vee$' and '$\neg$', an Xpath expression can be represented as a tree, called a twig pattern. For example, the XPath expression: *a*[*b*[*c* and .//*f*]]/*b*[*c* and *e*//*d*] can be represented as a tree *Q* shown in Figure 2.

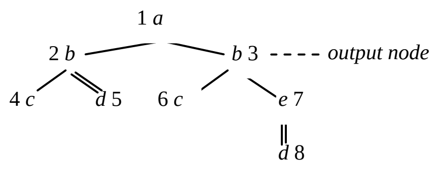


Figure 2. A query tree

In such a tree structure, a nodes v is labeled with an element name, a wild card '*' (that matches any element in a document tree T), or a string values, denoted as *label*(v). In addition, there are two kinds of edges: child edges (/-edges) for parent-child relationships, and descendant edges (//-edges) 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. For any node v in Q, we use $Q[v]$ to represent the subtree rooted at v. We also denote the query size, i.e., the number of nodes in Q, by $|Q|$; and the document size, i.e., the number of elements in T, by $|T|$.

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. 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:

(i) Preserve node label: For each $u \in Q$, *label*(u) = *label*($f(u)$) (or say u matches $f(u)$).

(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.

In an XML streaming environment, an XML document tree T is modeled as a stream S of modified SAX events: *startElement*(*tag*, *level*, *id*) and *endElement*(*tag*, *level*), where *tag* is the tag of the node being processed, *level* is the level at which the node appears, and *id* is the unique identifier assigned to the node. A node in T exactly corresponds to a *startElement* and (the corresponding *endElement* event) in S. In addition, if an element e has no subelement, a text is possibly associated with its *startElement*. These events are the input to our query evaluation processor.

In this paper, we propose a new algorithm to evaluate queries in such an environment, which runs in $O(|T| \cdot Q_{leaf})$ time and $O(T_{leaf} \cdot Q_{leaf})$ space, where T_{leaf} and Q_{leaf} represent the numbers of the leaf nodes in a document tree T and in a query tree Q, respectively.

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

Recently, a great many strategies have been proposed to evaluate XPath queries in an XML streaming environment (Avila *et al.*, 2002; Chen *et al.*, 2006; Ives *et al.*, 2002; Koch *et al.*, 2004; Ludascher *et al.*, 2002; Peng and Chawathe, 2003; Peng *et al.*, 2003). The methods discussed in (Avila *et al.*, 2002; Ives *et al.*, 2002) are based on finite state automata (*FSA*), but only able to handle single path queries, i.e., a query containing branching cannot be processed, as observed in (Peng and Chawathe, 2003). The method proposed in (Peng and Chawathe, 2003) is a general strategy, but requires exponential time ($O(|T| \times 2^{|Q|})$) in the worst case, as analyzed in (Peng *et al.*, 2003). The methods discussed in (Koch *et al.*, 2004; Ludascher *et al.*, 2002) do not support *d*-edges. If we extend them to general cases, exponential time is required. Up to now, the research culminates in *TwigM* presented in (Chen *et al.*, 2006). It is not only a general-case algorithm, but also works in polynomial time. In the worst case, its time complexity is bounded by $O(T_h Q_d |Q||T| + |Q|^2|T|)$, where T_h is the height of T and Q_d is the largest outdegree of a node in Q. By this method, each node q of Q is associated with a boolean array of length Q_d and a stack of size T_h, in which each element is a node v from T such that its relationship with the nodes in the stack associated with q's parent q' satisfies the relationship between q and q'. Therefore, each time to figure out a stack and push a node into it, $O(T_h Q_d |Q|)$ time is required, leading to a time complexity of $O(T_h Q_d |Q||T| + |Q|^2|T|)$. See Theorem 4.4 in (Chen *et al.*, 2006).

3. Main Algorithm

Weremark that in a streaming environment, the input to the XML query processor is a steam of modified SAX events; and an event is either startElement(*tag*, *level*, *id*) or *endElement*(*tag*, *level*). In order to evaluate a query Q, we have to scan a stream S from the beginning to the end and report any *startElement* event once the corresponding subtree is found containing Q.

For this purpose, we will maintain a global *stack* structure with each entry in it being a triplet: <*e*, *p*, *c*>, where *e* is a *startElement* event, *p* is a pointer to an entry in *stack* where its parent *startElement* is stored and *c* a pointer to the head of a linked list containing all the nodes constructed for its child elements, as illustrated in Figure 3.

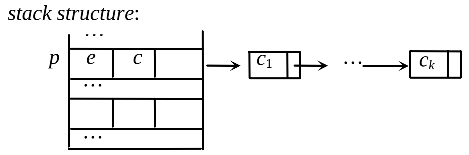


Figure 3. Illustration for *stack* structure

During the process, two other data structures are also maintained and computed to facilitate the discovery of subtree matchings according to Definition 1.

- Each node v (corresponding to a *startElement* event in S) in a document tree T is associated with a set, denoted $\alpha(v)$, contains all those nodes q in Q such that $Q[q]$ can be imbedded into $T[v]$.
- Each q in Q is associated with a value $\delta(q)$, defined as follows.

Initially, for each $q \in Q$, $\delta(q)$ is set to ϕ. During the tree matching process, $\delta(q)$ is dynamically changed as below.

(i) Let v be a node in T with parent node u.

(ii) If q appears in $\alpha(v)$, change the value of $\delta(q)$ to u.

Then, each time before we insert q into $\alpha(v)$, we will do the following checkings:

1. Check whether $label(q) = label(v)$.
2. Let $q_1, ..., q_k$ be the child nodes of q. For each q_i ($i = 1,..., k$), check whether $\delta(q_i)$ is equal to v.

If both (1) and (2) are satisfied, insert q into $\alpha(v)$.

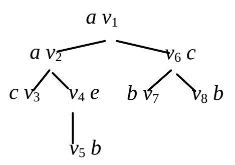
Below is the algorithm, which takes an event stream S and a twig pattern Q as the input. During the process, S is scanned from the beginning to the end and once a *startElement* event is found such that the subtree rooted at the corresponding node contains Q it will be reported.

In the algorithm, a virtual *startElement* event is used, which is considered to be the parent of the first *startElement* event in S (which corresponds to the root of T). The *level* number of the virtual event is set to be -1, and its *tag* and *id* are both set to be *nil*. Two variables E and E' are used. E' is for the current *startElement* event being processed while E is to store the parent of the current *startElement* event. In addition, each time a node v is constructed, a subprocedure *containment-check*(v, Q) is invoked to find all those $q \in Q$ such that $T[v]$ contains $Q[q]$ and store them in $\alpha(v)$.

```
Algorithm query-evaluation(S, Q)
input: S - an XML stream; Q - a twig pattern.
output: report any startElement such that for the corresponding node v, T[v] contains Q.
begin
1. push(the first element of S, stack);
2. E := virtual event;
3. while stack is not empty do {
4.   E' := top(stack);
     (*check the top element in stack*)
5.   E'.p := address of E;                (*establish parent link for E'*)
6.   let e be the next element in S;
7.   if  e is a startElement event then {
8.       E := E';
9.       push(e, stack);
10.      }
11.      else (*e is an endElement event.*)
12.          {E'' := pop(stack);          (*pop the top element out of stack*)
13.           generate node v for E''; E := E''.p;
14.           append v to the end of (E''.p).c;
15.           call containment-check(v, Q);
16.          }
17. }
end
```

The above algorithm processes the events in *S* one by one. Therefore, the corresponding document tree *T* is searched in the depth-first traversal fashion. Each time a *startElement* event is encountered, it will be pushed into *stack* (see line 1 and lines 6 - 9) and stay there until its corresponding *endElement* is encountered (see lines 11 - 12). In this case, it will be popped out of *stack* and a node *v* for it will be constructed (see line 13), for which a containment check will be performed (see line 15).

Example 1. Consider the document tree *T* in Figure 4(a). Its XML stream *S* is shown in Figure 4(b). Applying the algorithm *query-evaluation*( ) to *S*, we will regain *T* if line 15 is not executed. In Figure 5, we trace the first 8 steps of the execution process.

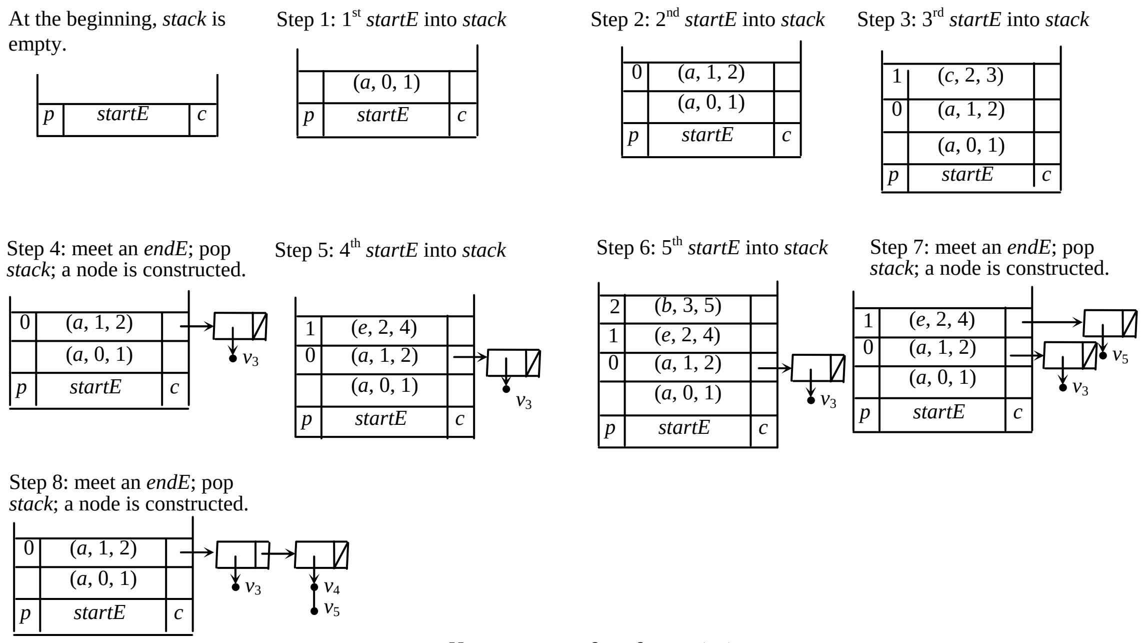


(a)

1. *startE*(*a*, 0, 1)	9. *endE*(*a*, 1)
2. *startE*(*a*, 1, 2)	10. *startE*(*c*, 1, 6)
3. *startE*(*c*, 2, 3)	11. *startE*(*b*, 2, 7)
4. *endE*(*c*, 2)	12. *endE*(*b*, 2)
5. *startE*(*e*, 2, 4)	13. *startE*(*b*, 2, 8)
6. *startE*(*b*, 3, 5)	14. *endE*(*b*, 2)
7. *endE*(*b*, 3)	15. *endE*(*c*, 1)
8. *endE*(*e*, 2)	16. *endE*(*a*, 0)

(b)

Figure 4. A document tree and its XML stream



At the beginning, *stack* is empty.
Step 1: 1st *startE* into *stack*
Step 2: 2nd *startE* into *stack*
Step 3: 3rd *startE* into *stack*
Step 4: meet an *endE*; pop *stack*; a node is constructed.
Step 5: 4th *startE* into *stack*
Step 6: 5th *startE* into *stack*
Step 7: meet an *endE*; pop *stack*; a node is constructed.
Step 8: meet an *endE*; pop *stack*; a node is constructed.
p
startE
c
(a, 0, 1)
(a, 1, 2)
(c, 2, 3)
(e, 2, 4)
(b, 3, 5)
v_3
v_4
v_5

Figure 5. Illustration for for $L(q_i)$'s

From the above discussion, we can see that a document tree can always be constructed by scanning the corresponding XML stream *S*. For the purpose of query evaluation, however, we have to check the containment each time a node of *T* is constructed. This is done by calling *containment-check*(*v*, *Q*), in which another two functions are invoked to do different checkings:

- *element-check*(*u*, *q*): *u* is an element containing subelements. It checks whether *T*[*u*] contains *Q*[*q*]. If it is the case, return {*q*}. Otherwise, it returns an empty set ∅.

- *bottom-element-check*(u, Q): u is an element containing no subelement. It returns a set of nodes in Q: $\{q_1, ..., q_k\}$ such that for each q_i ($1 \le i \le k$) the following conditions are satisfied.
  (i) $label(u) = label(q_i)$.
  (ii) if q_i has a child, then the child must be a text and matches the text associated with u.

```
Algorithm containment-check(v, Q)
input: v - a node in T; Q - a twig pattern.
output: a(v) - a set of query node q such that T[v] contains Q[q].
begin
1.  $C := \emptyset$; $C_1 := \emptyset$; $C_2 := \emptyset$;
2.  if v.c is not nil then                    (*v has some subelements.*)
3.   {let $v_1, ..., v_k$ be the child nodes of v;
4.    $\alpha := \alpha(v_1) \cup ... \cup \alpha(v_k)$;
5.    for each $q \in \alpha$ do
6.        {$\delta(q) := v$; $C := C \cup \{q$'s parent$\}$;}
7.        remove all $\alpha(v_j)$ ($j$ = 1, ..., $k$);
8.        for each q' in C do
9.          $C_1 := C_1 \cup$ element-check(v, q');
10.   }
11. $C_2$ := bottom-element-check(v, Q);
12. $\alpha(v) := \alpha \cup C_1 \cup C_2$;
end
```

```
Function element-check(u, q)
begin
1. $C_1 := \emptyset$;
2. if label(q) = label(u) then          (*If q is '*', the checking is always successful.*)
3. {let $q_1, ..., q_k$ be the child nodes of q;
4.  if for each $q_i$ ($i$ = 1, ..., $k$) d($q_i$) is equal to u
5.  then {$C_1 := \{q\}$;
6.  if q is root then report u;}}
7. return $C_1$;
end
```

```
Function bottom-element-check(u, Q)
begin
1. $C_2 := \emptyset$; flag := false;
2. for each leaf node q in Q do {
3.   if q is a text then {
4.       let q' be the parent of q;
5.       if label(q') = label(u) and
         q matches the text associated with u then {$C_2 := C_2 \cup \{q'\}$; flag := true;
6.    }
7.   else {
```

```
8.          if $label(q) = label(u)$ then {
9.              $C_2 := C_2 \cup \{q\}$; $flag := true$;
10.     }
11.     if $q$ is $root$ and $flag := true$ then report $u$;
12.         $flag := false$;
13. }
14. return $C_2$;
end
```

One of the inputs to the algorithm *containment-check*( ) is a node v constructed in the execution of *query-evaluation*(S, Q). If v corresponds to an element that has no subelement, the function *bottom-element-check*( ) is called (see line 11), by which a(v) will be established by checking it against all the leaf nodes of Q. Otherwise, $\alpha(v_i)$ will be checked for all the child nodes v_i of v (see lines 3 -6). Concretely, for each q in α (= $\alpha(v_1) \cup ... \cup \alpha(v_k)$), the value of $\delta(q)$ will be changed to v. Meanwhile, q's parent will be stored in a temporary variable C. Then, all the nodes q' in C are the candidates to be further checked. This is done by calling *element-check*(v, q') to see whether $T[v]$ contains $Q[q']$ (see lines 8 -9). Special attention should be paid to the fact that *bottom-element-check*( ) should also be applied to v to find all the leaf nodes of Q which matche v.

Finally, we notice that in the execution of *element-check*( ), $\delta(q)$'s are utilized to facilitate the checkings (see lines 3 - 5 in *element-check*( )).

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. 6.

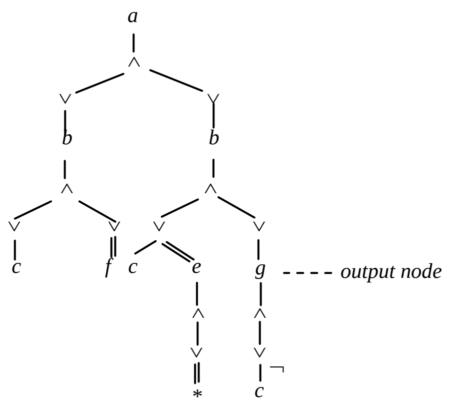


Figure 6. A query tree with different logic operators

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 q be a node in Q with child nodes q_1, ..., q_k. Let v be a node in T with child nodes v_1, ..., v_l.

(i) If q is a $\vee$-node, $T[v]$ contains $Q[q]$ if one of the following conditions holds:

- There exists a positive //-child q_i ($1 \leq i \leq k$) such that $T[v]$ contains $Q[q_i]$.
- There exists a positive /-child q_i ($1 \leq i \leq k$) such that $T[v]$ contains $Q[q_i]$ and *label*(v) = *label*(q_i).
- There exists a negative //-child q_i ($1 \leq i \leq k$) such that $T[v]$ does not contain $Q[q_i]$.
- There exists a negative /-child q_i ($1 \leq i \leq k$) such that $T[v]$ does not contain $Q[q_i]$ or $T[v]$ contains $Q[q_i]$ but *label*(v) $\neq$ *label*(q_i).

(ii) If q is a $\wedge$-node, $T[v]$ contains $Q[q]$ if the following conditions hold:

- for every positive node q_i ($1 \leq i \leq k$), there exists a v_j ($1 \leq j \leq l$) such that $T[v_j]$ contains $Q[q_i]$.
- for every nagative node q_i ($1 \leq i \leq k$), there exists no v_j ($1 \leq j \leq l$) such that $T[v_j]$ contains $Q[q_i]$.

(iii) If q is a name node, $T[v]$ contains $Q[q]$ if the following conditions hold:

- $T[v]$ contains $Q[q_1]$ (q has only one child node q_1.)
- *label*(v) = *label*(q).

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 v in T two sets: $\alpha(v)$ and $\beta(v)$. $\alpha(v)$ is defined in the same way as in Section 3; and $\beta(v)$ contains all those $\vee$-nodes q in Q such that $Q[q]$ can be imbedded into $T[v]$. Besides, in order to calculate $\beta(v)$, we maintain an array N_Q containing all the negative nodes in Q.

With $\alpha(v)$ and $\beta(v)$, we need to slightly change the algorithms *containment-check*( ) and *element-check*( ) discussed in 4.1. But Algorithm *bottom-element-check*( ) needn't be modified.

Algorithm *general-containment-check*(v, Q)

input: v - a node in T; Q - a twig pattern.

```
output: a($v$) - a set of query node $q$ such that $T[v]$ contains $Q[q]$.
begin
1. $S := \emptyset$; $S_1 := \emptyset$; $S_2 := \emptyset$;
2. if $v.c$ is not nil then                    (*$v$ has some subelements.*)
3.   {let $v_1, ..., v_k$ be the child nodes of $v$;
4.    $\alpha := \alpha(v_1) \cup ... \cup \alpha(v_k)$;
5.       for each $q \in \alpha$ do {
6.          for $q \in \beta(v)$ do {$\delta(q) := v$;}
7.       assume that $\alpha = \{q_1, ..., q_j\}$;
8.       for $i = 1$ to $j$ do {
9.          $S := S \cup \{q_i$'s parent$\}$;
10.      remove all $\alpha(v_j)$ ($j = 1, ..., k$);
11.      for each $q$ in $S$ do
12.         $S_1 := S_1 \cup$ general-element-check($v, q$);
13.   }
14.   $S_2 :=$ bottom-element-check($v$);
15.   $\alpha(v) := \alpha \cup S_1 \cup S_2$;
16.   call calculate-$\beta$($v$, $\alpha(v)$);
end
```

```
Function general-element-check($u$, $q$)
begin
1.       $S_1 := \emptyset$;
2.      if label($q$'s parent) = label($u$) then (*If $q$ is *, the checking is always successful.*)
3.       {let $q_1, ..., q_k$ be the child nodes of $q$;
4.        if for each $q_i$ ($i = 1, ..., k$) $\delta(q_i)$ is equal to $u$
5.        then {$S_1 := \{q\}$;
6.                  if $q$'s parent is root then mark $u$};}
7.       return $S_1$;
end
```

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

Algorithm *general-containment-check*( ) is similar to Algorithm *containment-check*( ). The only difference is lines 6, 12, and 16. In line 6, we establish δ values for query nodes based on $\beta(v)$ instead of $\alpha(v)$. In addition, different treatments of /-child and //-child nodes are shifted to Function *calculate*-β( ) (see line 16.) In line 12, we call Function *general-element-check*( ) instead of *element-check*( ). In line 16, we call Function *calculate*-β( ) to generate $\beta(v)$.

We also notice that Function *general-element-check*( ) is a little bit different from Function *element-check*( ). It corresponds to the checking of $\wedge$-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*-$\beta(v, S)$, we compute $\beta(v)$ based on $\alpha(v)$. 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 S (see lines 7 - 9 in this function). It needs $O(|N_Q|\cdot\log S)$ 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. Conclusion

In this paper, an efficient algorithm for the query evaluation in an XML streaming environment is presented. The algorithm runs in $O(|T|\cdot Q_{leaf})$ time and $O(|T|\cdot Q_{leaf})$ space, where T_{leaf} stands for the number of the leaf nodes in a document tree T and Q_{leaf} for the number of the leaf nodes in a query tree Q. This computational complexity is much better than any existing strategy for this problem. In addition, this method can be extended to handle general queries.

References

I. Avila-Campillo, T.J. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suciu (2002), XMLTK: An XML Toolkit for Scalable XML Stream Processing, in *Programming Langauge Technologoes for XML(PLAN-X)*, 2002.

D.D. Chamberlin, J.Clark, D. Florescu and M. Stefanescu (2002) XQuery1.0: An XML Query Language, http:/ /www.w3.org/TR/query-datamodel/.

D.D. Chamberlin, J. Robie and D. Florescu (2000) Quilt: An XML Query Language for Heterogeneous Data Sources, *WebDB 2000*.

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

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

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

Z.G. Ives, A.Y. Halevy, and D.S. Weld (2002), An XML query engine for network-bound data, *VLDB Journal*, 11(4), 2002.

D.E. Knuth (1969), *The Art of Computer Programming, Vol.1*, Addison-Wesley, Reading, 1969.

C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier (2004), Schema-based Scheduling of Event Processor and Buffer Minimization for Queries on Structured Data Stream, in: *Proc. of VLDB*, 2004.

B. Ludascher, P. Mukhopadhayn, and Y. Papakonstantinou (2002), A Transducer-based XML Query Processor, in: *Proc. of VLDB*, 2002.

F. Peng and S.S. Chawathe (2003), XPath queries on streaming data, in: *Proc. of SIGMOD*, 2003.

F. Peng and S.S. Chawathe (2003), XSQ: A Streaming XPath Engine, Technical Report CS-TR-4493, University of Maryland, 2003.

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

World Wide Web Consortium (2005). XQuery 1.0: An XML Query Language, W3C Recommendation, Version 1.0, 2005. See http://www.w3.org/TR/xquery.