World Wide Web (2008) 11:407–425
DOI 10.1007/s11280-008-0047-z

Holistically Stream-based Processing Xtwig Queries

Guoren Wang · Bo Ning · Ge Yu

Received: 1 February 2008 / Revised: 5 May 2008 /
Accepted: 17 May 2008 / Published online: 13 June 2008

Abstract Unlike a twig query, an Xtwig query contains some selection predicates with reverse axes which are either *ancestor* or *parent*. To evaluate such queries in the stream-based context, some rewriting rules have been proposed to transform the paths with reverse axes into equivalent reverse-axis-free ones. However, the transformation method is expensive due to multiple scanning input streams and the generation of unnecessary intermediate results. To solve these problems, a holistic stream-based algorithm *XtwigStack* is proposed for Xtwig queries. Experiments show that *XtwigStack* is much more efficient than the transformation method.

Keywords XML · query processing · twig pattern · Xtwig pattern

1 Introduction

XML has become the de-facto standard for exchanging data and representing information over the Web, and database servers are employed to store large amounts of XML documents, for example, in XML dissemination systems [7] and XML digital libraries [24]. XML data may be very complex and deeply nested, consequently there is a lot of interest in query processing over data that conforms to a tree-structured data model. XML Queries typically contain some selection predicates with forward axes. However, it is reasonable that XML queries may be more complex and contain some selection predicates with ancestor and/or parent reserve axes. The following

G. Wang (🖂) · B. Ning · G. Yu
School of Information Science and Engineering, Northeastern University,
Shenyang 110004, China
e-mail: wanggr@mail.neu.edu.cn
URL: http://mitt.neu.edu.cn/wanggr

XPath [26] expression Q_1 is an example of such query based on the sample XML document in Figure 1a.

$$Q_1 : book\ [\ ancestor :: Computer][ancestor :: Addison - Wesley]$$
$$[\ title = \text{“}XML\text{”}]//author[//name = \text{“}John\text{”}]$$

Expression Q_1 matches book elements that (i) have a child subelement *title* containing “XML”, (ii) have a subelement *author* whose name is “John”, (iii) are descendants of *Addison-Wesley* elements, and (iv) are descendants of *computer* elements. This expression can be represented as a complex node-labeled pattern in which elements and string values are regarded as node labels, as shown in Figure 1b. Q_1 involves the ancestor and descendant nodes of *book* elements.

Compared to DOM-based processing methods [16, 23], stream-based processing methods [3, 5, 8, 9, 13–15, 19, 25] have been widely used recently, and there is increasing interest in the work of evaluating queries with reverse axes directly in the stream-based context. Olteanu et al. [18] propose some rewriting rules to transform absolute XPath location paths with reverse axes into equivalent reverse-axis-free ones. There are lots of works that evaluate queries with only forward axes, such as path pattern [3, 25] and twig pattern [19]. With the rewriting rules, Q_1 can be transformed into a twig query which can be evaluated using existing algorithms such as *PathStack* [13], *TwigStack* [5], *TJFast* [14], *PPS* [9], *Extended Dewey* [15], CCPI [22] or *Twig2Stack* [8]. But the transformed query patterns contain equality joins of nodes, and remain expensive to evaluate. When there are multiple ancestor axes in the query pattern, the processing cost increases.

The other method is to decompose the Xtwig pattern into a set of twig patterns, each of which can be matched against the XML database, then these twig matches are joined to get the final results.

Both of the above mentioned methods cannot avoid the join operations caused by transformation. In this paper, a new algorithm is developed to match Xtwig queries holistically without unnecessary joins and therefore avoid the extra processing overhead of intermediate results, and a new data structure called *XCube* is proposed to conduct the matching of vpatterns, which is the core problem of the Xtwig pattern matching.

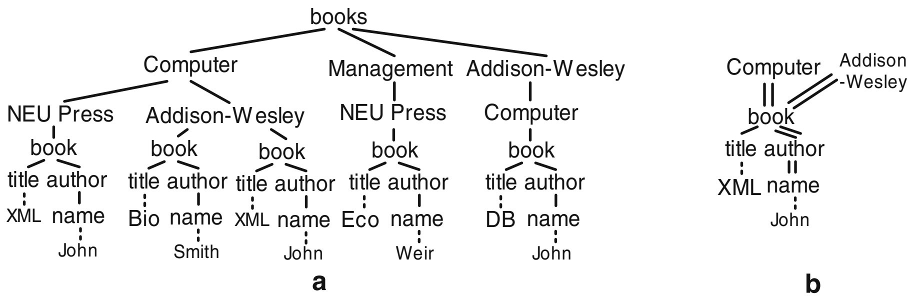


Figure 1 Example of an Xtwig query. **a** XML document. **b** Xtwig query pattern.

The contributions of this paper are summarized as follows.

- We propose a new query pattern Xtwig. In real applications, XML queries are more complex and may contain predicates with ancestor axes, but there is not any research work on stream-based processing the queries with ancestor axes so far.
- We analyze the characteristics of the Xtwig pattern matching, and propose a new data structure called *XCube* to solve the relations of predicates with ancestor axes. We then propose algorithm called *XtwigStack* to solve the Xtwig pattern matching. It is a holistic join algorithm for Xtwig pattern matching.
- Finally we present the experimental results on a range of real and synthetic data to complement our analytical results. The performance study shows that our algorithms perform well on execute time and the amount of intermediate results.

The rest of this paper is organized as follows. Section 2 introduces the XML data model and the Xtwig query pattern. Section 3 explains in detail the matching of Xtwig patterns and presents the *XtwigStack* algorithm. Section 4 shows our experiment results and performance evaluation. Related work and conclusions are given in Sections 5 and 6 respectively.

2 Preliminaries

2.1 XML data model

An XML document can be modeled as an ordered tree, where nodes represent elements, attributes and text data, and edges correspond to parent-child(p-c) relationships between nodes. The structural relationship between any two nodes can be captured by a region-based encoding scheme [11, 20, 28], in which each node can be represented with a tuple (*docID, startpos, endpos, level*), based on its position and level in the ordered tree. In this scheme, the ancestor-descendant(a-d) relationship between two nodes can be determined by comparing their start and end positions. For example, node u is an ancestor of v if u.startpos $<$ v.startpos and v.endpos $<$ u.endpos. Furthermore, if $u.level = v.level - 1$, then u is the parent of v.

2.2 XPath and axes

Usually, query pattern can be expressed with XPath. The primary purpose of XPath is to address parts of an XML document, by navigating through the XML document structure. There are 13 axes in XPath, such as *ancestor*, *descendant*, *following*, and *following-sibling*. An axis is either forward or reverse. An axis is reverse if it only contains the context node or nodes that are before the context node in the document order. Thus, *parent*, *ancestor*, *ancestor-or-self*, *preceding*, and *preceding-sibling* are reverse axes, while all the other axes are forward. The following gives two example queries containing reserve axes.

$$Q_2\text{:}A//C[ancestor::B] \text{ and } Q_3\text{:}A//C[parent::B]$$

To simplify XPath expressions with reverse axes, we introduce symbol **$** for **parent::** and symbol **%** for **ancestor::**. Then Q_2 and Q_3 can be expressed in the abbreviated form:

Q_2: A//C[% B] and Q_3: A//C[$B]

2.3 Xtwig pattern

An **Xtwig pattern** is a query that contains parent or ancestor axes in its predicates. In the Xtwig pattern, a node is called a **vnode** if it has more than two in-edge reserve axes, and the parent of a vnode is called a **vpnode** of the vnodes. For example, in Figure 2a node *V* is a vnode and nodes *A*, *B* and *C* are its vpnodes. It is noted that the ancestors of the parents of a vnode *v* are not vpnodes of *v*. For example, in Figure 2a node *V* is a vnode and nodes *A*, *B* and *C* are its vpnodes, but node *D* is not a vpnode of vnode *V*. An Xtwig pattern is called a **vpattern** if it contains a leaf vnode. A vpattern only consisting of vnodes and vpnodes is called a **simple vpattern**. The number of the vpnodes in a vpattern is called the **Reverse Fan-out Degree(RFOD)** of the vpattern. For example, Figure 2a is a simple vpattern and its RFOD is 3.

In a vpattern, the maximal subtree including a vpnode as a leaf is called a **vtwig** of the vpnode. Similarly the maximal path including a vpnode as a leaf is called a **vpath** of the vpnode. For example, in Figure 2b *D//A* is a vpath of vpnode *A* while in Figure 2c *C[//E]//B//D* is a vtwig of vpnode *D*.

A more complex vpattern can be gotten by the combination of a simple vpattern, vpaths, and/or vtwigs at vpnodes. For example, the vpattern in Figure 2d can be gotten by the combination of the simple vpattern *V[%B][%A]* and the vtwig *C[//E]//B//D* at vpnode *B*. Similarly, a more complex Xtwig pattern can be gotten by the combination of a simple vpattern, paths, and/or twigs at the vnode. For example, in Figure 2, (e) can be gotten by the combination of the simple vpattern *V[%A][%B][%C]* and the path *V//D//E* at vnode *V* and (f) can be gotten by the combination of the simple vpattern *V[%A][%B][%C]* and the twig *V//D[//E]//F//G* at vnode *V*.

Given a query Xtwig pattern *Q* and an XML database *D*, a matching of *Q* in *D* is identified by a mapping from nodes in *Q* to nodes in *D* such that (i) query node predicates are satisfied by the corresponding database nodes, and (ii) the structural (parent-child and ancestor-descendant) relationships between query nodes are satisfied by the corresponding database nodes. We call the matching of *Q* in *D* **Xtwig Pattern Matching**.

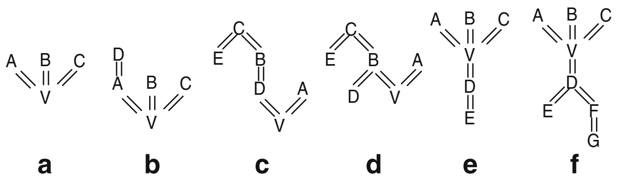


Figure 2 Examples of Xtwig patterns (**a**–**f**).

3 Xtwig pattern matching

In this section, we first analyze the limitations of the two basic methods, vertical decomposition and horizontal decomposition. Then we study the matching of vpatterns which is the core problem of the Xtwig pattern matching. We present a new data structure called *XCube* to compactly represent the results of a vpattern matching. With the new stack structure, we present *XtwigStack*, an algorithm for finding all the matching results of an Xtwig pattern against an XML document.

3.1 Limitations of basic decomposition methods

There are two kinds of decomposition methods to match an Xtwig query pattern. The straightforward one is, at each vnode, to vertically decompose the Xtwig pattern into multiple twig patterns. Each individual decomposed twig pattern can be evaluated using the traditional twig algorithm, then they are join-merged at the vnodes to get the final results. This method is referred to as *VertiDec*. Processing decomposed twig patterns needs multiple scanning of the streams. Furthermore, many intermediate results may not be part of the final results. For example, if *John* published another *XML* book at another publisher, this would be an unnecessary intermediate result for Q_1. The other method is, at each vnode, to horizontally decompose the Xtwig pattern into vpatterns and twig patterns. The decomposed twig patterns and vpatterns are processed separately, and join-merged at the vnodes to get the final results. This method is referred to as *HoriDec*. Compared with *VertiDec*, *HoriDec* scans fewer element streams, but still has to scan the vnode tag stream twice at least, that is because each vnode belongs to both the decomposed vpattern and twig pattern. Moreover, it has the same problem of unnecessary intermediate results as *VertiDec*.

For example, Q_1 can be decomposed into Q_4 and Q_5 by *VertiDec* (R is the root of the XML document):

Q_4*:R[//publisher="Addison-Wesley"]//book[title="XML"]//author[@name="John"]*

Q_5*:R[//catalog="computer"]//book[title="XML"]//author[@name="John"]*

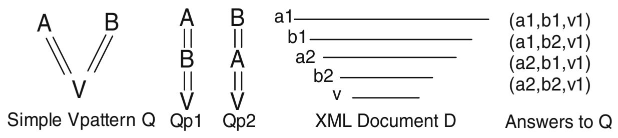
Q_1 can be decomposed into Q_6 and Q_7 by *HoriDec*:

Q_6*:R//book[title="XML"]//author[@name="John"]*

Q_7*:R//book[%publisher="Addison-Wesley"][%catalog="computer"]*

Besides these two basic decomposition methods, we can also solve the Xtwig pattern matching by rewriting rules [4, 18] which can transform an Xtwig pattern into an equivalent reverse-axis-free twig pattern. The idea of rewriting is that, instead of looking back from the context node, one can look forward from the beginning of the document to match the tag T in predicates with reverse axes. Instead of checking whether the context node has an ancestor T or not, one looks for a T node, then

Figure 3 Example of Vbranches matching.



for a following node that is identical to the context node. For example, Q_8 can be equivalently transformed into Q_9 which is reverse-axis-free.

Q_8*:book[ancestor::Computer]*

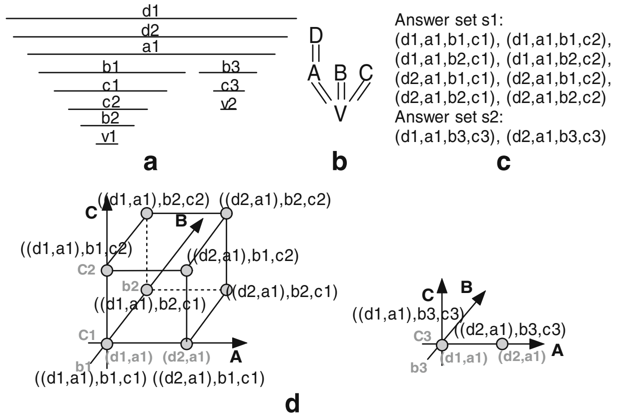
Q_9*:book[/descendant::Computer/book::node()==self::node()]*

Although there is no ancestor axis in Q_9, evaluating Q_9 is rather expensive because the streams have to be scanned twice and there is an equality join caused by the rewriting. For an Xtwig pattern containing multiple predicates with ancestor axes, the evaluation cost of the rewriting way increases.

3.2 Matching of vpatterns with vpaths only

Consider the simple vpattern A//V[%B] in Figure 3. In this pattern, A and B are ancestors of V. There are two possible cases, A is an ancestor of B or B is an ancestor of A. Therefore the answer to the simple vpattern Q is the union of Q_{p_1} and Q_{p_2} in Figure 3. Similarly, for any vpattern, the results of the vpattern matching are the permutation of the vpnode elements which have the a-d relationships and satisfy respective vpattern structure relationships. *XCube* is proposed to express this permutation. For a given vpattern, an *XCube* is a hyper cube, and the dimensionality of the *XCube* is the number of vpnodes in the vpattern. Each type of *XCube* is corresponding to a vpattern. If there are n vpatterns in an Xtwig pattern, there are n kinds of *XCubes*. Every coordinate axis is calibrated by the answer list of corresponding vpnode. Every node in the hyper cube is just one answer to the vpattern, and each *XCube* is a set of answers to the corresponding vpattern.

Figure 4 Examples of XCube. **a** XML document. **b** Query. **c** Answers. **d** XCube xc1 and xc2 to answer set s1 and s2.


d1
d2
a1
b1
b3
c1
c3
c2
v2
b2
v1
a
D
A
B
C
V
b
Answer set s1:
(d1,a1,b1,c1), (d1,a1,b1,c2),
(d1,a1,b2,c1), (d1,a1,b2,c2),
(d2,a1,b1,c1), (d2,a1,b1,c2),
(d2,a1,b2,c1), (d2,a1,b2,c2)
Answer set s2:
(d1,a1,b3,c3), (d2,a1,b3,c3)
c
((d1,a1),b2,c2) ((d2,a1),b2,c2)
C
B
((d1,a1),b1,c2)
((d2,a1),b1,c2)
C2
b2
((d1,a1),b2,c1)
(d2,a1),b2,c1)
C1
b1
(d1,a1) (d2,a1)
A
((d1,a1),b1,c1) ((d2,a1),b1,c1)
C
B
((d1,a1),b3,c3)
((d2,a1),b3,c3)
C3
b3
(d1,a1) (d2,a1)
A
d

Consider the vpattern query in Figure 4b on the XML document in Figure 4a. The results of the query are given in Figure 4c. These two result sets can be expressed by *XCubes* xc_1 and xc_2, as shown in Figure 4d. Because the query pattern has three vpnodes, the dimensionality of the *XCube* is 3. Every coordinate axis is calibrated by the element list and all the nodes on the *XCube* are the answers to the vpattern. For example, for the answer set S_1, coordinate axis A is calibrated by the answer list of vpath $D//A$. Since the answer list of $D//A$ is $\{(d_1,a_1), (d_2,a_1)\}$, coordinate axis A is calibrated by these two values, as shown in *XCube* xc_2 of Figure 4d. Coordinate axes B and C can be calibrated in a similar way.

Algorithm 1 gives the computation of matching vpatterns with vpaths only. It associates each node q in the vpattern with a stream T_q containing all the elements of tag q and each stream has an imaginary cursor, which can either move to the next element or read the current element. The operations over streams are **eof**, **advance**, **next**, **nextL** and **nextR**. Operation *advance*(T_q) moves the current cursor to the next element in tag stream T_q, and operation *next*(T_q) returns the current element in tag stream T_q. The last two operations return the startpos and endpos of current element in the stream. At the same time, it associates a single stack S for all vpnodes, and associates a separate stack S_T for each ancestor node T of respective vpnodes to maintain their relationships in the vpath. The operations for two kinds of stacks are **empty**, **popStack**, **pushStack**, **topL** and **topR**. The last two operations return the startpos and endpos of current element in the stack. All the elements of vpnodes will be pushed into stack S to maintain the a-d relationships of all the vpnode elements, while the other tag elements will be pushed into respective stacks to preserve the answers of the tag itself. In Algorithm 1, line 2 returns the tag, the next element of which has minimal startpos among the streams of the vpnodes. Line 5 pops stack S until the top is the ancestor of the next(T_q). When the algorithm runs to line 11, the next element is not an ancestor of the top of stack S, and it is going to output the answers. Before outputting the answers, the algorithm has to determine whether or not the current stack S contains elements of all the vpnodes. Line 12 outputs the answer set in the form of an XCube by scanning stack S.

Consider the vpattern query in Figure 4b on the XML document in Figure 4a. There are three vpaths in the vpattern, $D//A$, B, and C, as shown in Figure 5a. Node D is associated with stack S_D as node D is an ancestor of vpnode A, and vpnodes A, B and C all are associated with stack S. The first two elements d_1 and d_2 in stream T_d, as shown in Figure 5b, are both pushed into S_D, as shown in Figure 5c, because d_2 is a child of d_1. For element a_1, since A is a vpnode it is pushed to stack S and its parent pointer is set to d_2, which is the top of stack S_D. Then b_1, c_1, c_2 and b_2 are pushed into S too, as shown in Figure 5. When element b_3 comes, it is confirmed that the current stack S contains elements from all the vpnodes, then the algorithm can output the XCube shown in Figure 5d and pop stack S until the top of S is an ancestor

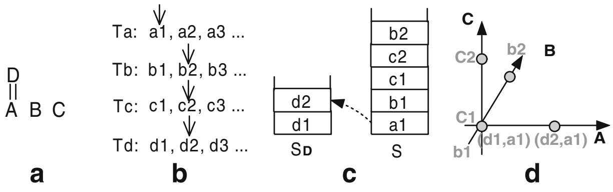


Figure 5 Illustration of algorithm BPCube. **a** Vpaths. **b** Streams. **c** Stacks. **d** XHyperCube.

Algorithm 1: BPCube(v)

```
Data: Query $V$ of vpatterns with vpaths only, its vnode is $v$, and tag streams $T_q$ with
      $q \in V$.
Result: The matching of $V$ against the tag streams.
1  while not at end of any stream do
2      $q_{act} = \ q_i,\ q_i \in Tags\ of\ Vbranches$ such that $nextL(T_{q_i})$ is minimal;
3      if $q_{act}$ is vpnode then
4          if $q_{act}$ is not a root then
5              Pop stack $S_{parent(q_{act})}$, until the top is an ancestor of next($q_{act}$);
6          if $q_{act}$ is a root or $S_{parent}(q_{act})$ is not empty then
7              if stack S is empty then
8                  Push element next($T_{q_{act}}$) to stack S;
9              else if nextL($T_{q_{act}}$)>topL(S) and nextR($T_{q_{act}}$)<topR(S) then
10                 Push element next($T_{q_{act}}$) to stack S;
11             else if stack contains elements of all the tags of vpnodes then
12                 Output the XCube by scanning stack S and respective stack $S_q$;
13                 Pop stack S, until the top is an ancestor of next($T_{q_{act}}$);
14                 Push element next($T_{q_{act}}$) to stack S;
15     if $q_{act}$ is not vpnode then
16         if $q_{act}$ is not a root then
17             Pop stack $S_{parent(q_{act})}$, until the top is an ancestor of next($q_{act}$);
18         if $q_{act}$ is a root or $S_{parent}(q_{act})$ is not empty then
19             Pop stack $S_{act}$, until the top is an ancestor of next($q_{act}$);
20             Move the current element of $Tq_{act}$ to $S_{q_{act}}$, set pointer to the top of $S_{parent(q_{act})}$;
21     advance($T_{q_{act}}$);
```

of b_3. Note that the algorithm does not pop stack S_D until the next element of tag D is not a descendant of the top of S_D.

3.3 Matching vpatterns with vtwig

In this section, we study the matching of vpatterns with vtwig. Figures 2c and 2d are such query patterns. First, we propose a new stack structure called *Stick Stack* to store the temporary answers caused by the sticks, then propose a stream-based algorithm called *BCube* to process the matching of vpatterns with vtwig based on the new proposed stick stack structure.

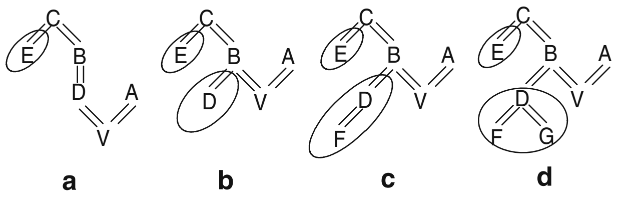


Figure 6 Examples of stick patterns (**a**–**d**).

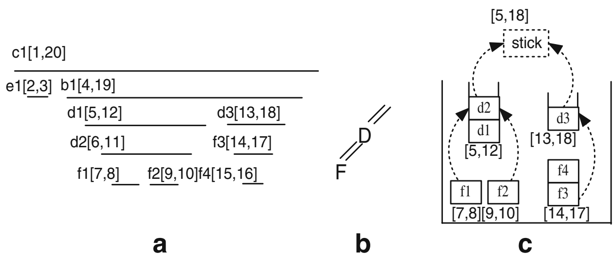


Figure 7 Illustration of Stick Stack for a stick path. **a** Fragment of XML document. **b** Stick path. **c** Stick stack.

3.3.1 Stick stack

In a vpatttern with vtwig, a *stick* node is a node not included in any vpath of the vpattern. For example, in Figure 6 pattern (a) has a stick node E; (b) has two stick nodes E and D; (c) has three stick nodes D, E and F, and D and F forms a stick path *D//F*; (d) has four stick nodes E, D, F and G, and D, F and G forms a stick twig *D[//F]//G*. Note that a simple stick node, a stick path, or a stick twig are usually referred to as a stick patten.

To solve the matching of vpatterns with vtwig, a new structure called *Stick Stack* is proposed to store the temporary answers of stick patterns in a vpattern with vtwig. During the matching of such vpattern, a *Stick Stack* is associated with each stick pattern that consists of node stacks. Those node stacks are built for stick nodes. For a stick node A, the elements of A with an ancestor-descendant relationship are grouped into the same node stack and an upper element is a descendant of the lower elements in the same node stack. It is obvious that a stick node may have multiple node stacks. Moreover, each node stack is associated with a code, which can be easily determined by the code of the bottom element of the node stack. Based on the region encoding, we can easily determine the relationship among node stacks. For two node stacks s_1 and s_2, if $s_1.top.startpos < s_2.top.startpos$ and $s_2.top.endpos < s_1.top.endpos$, then s_1 is called the ancestor node stack of s_2. Moreover, each element e in a node stack has a pointer to the parent element of e in an ancestor node stack. For each *Stick Stack*, there is a root node *stick* with a region encode which is defined as the minimal startpos and maximal endpos among the elements in all the node stacks. Figure 7c is an example of the stick stack of stick path $D//F$ of Figure 6c. The region code of the *Stick Stack* is [5, 18].

3.3.2 BCube: matching vpatterns with vtwig

Algorithm *BCube* is designed to deal with the matching of vpatterns with vtwig and presented in Algorithm 2.

There are three kinds of stacks in Algorithm *BCube*. The first one is stack S, which is used to preserve the relationships of vpnode elements, and all the vpnode elements are pushed to S if they are the descendant of a top stack element. The second one is the stick stack, which is designed to store the temporary answers of stick patterns. The last one is a stack associated to those nodes which are not vnodes, vpnodes, and stick nodes. They are similar to the stacks in Algorithm *TwigStack*.

In Algorithm *BCube*, *getMinX(v)* returns the tag q_{act} of which stream is going to be processed. If q_{act} is a vpnode, the next element of stream $T_{q_{act}}$ is pushed to stack S,

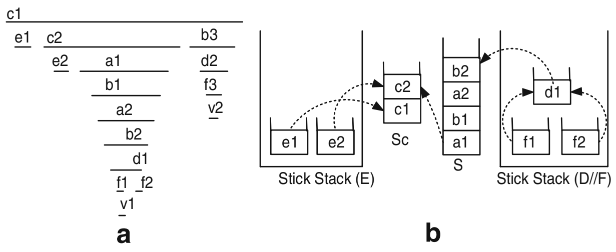


Figure 8 Illustration of algorithm BCube. **a** XML document. **b** All kinds of stacks.

as in Algorithm *BPCube*. When the next element *e* is not a descendent of top element of *S*, as stack *S* is popped as the *XCube* is generated to output the answers, until *e* is a descendent and it is pushed to *S*. If q_{act} belongs to a stick pattern, the next element *e* is pushed to the stick stack to store the temporary answers to the stick pattern. Otherwise, the next element of q_{act} is pushed to respective stacks as in Algorithm *TwigStack*.

Consider the vpattern in Figure 6c. Figure 8 illustrates the execution process of Algorithm *BCube*. In the algorithm, there are two stick stacks *stick stack(E)* and *stick stack(D//F)*, and there is a stack *S* to preserve the relationships of elements from vpnode streams. There is also a stack S_c. The next tag *getMinX(v)* returns is *b*, and *next*(T_b) is b_3. Because *B* is a vpnode and b_3 is not a descendant of the top element of *S*, the current final answers can be output and stack *S* is popped until *S* is empty.

3.4 XtwigStack: matching Xtwig patterns

So far we have addressed the matching of vpatterns with vpaths only and vpatterns with vtwig. Next we discuss the matching of more general forms of Xtwig patterns such as in Figure 2e and f.

In order to solve the matching of Xtwig patterns, Algorithm *XtwigStack* is proposed by applying Algorithm *BCube* to *PathStack* or *TwigStack*. Function *BCubeStep*, which is invoked once, is defined to match a vpattern. By regarding the set of a vpattern as a virtual node, the *advance* function in *PathStack* or *TwigStack* can be changed in order to process the matching of the vpattern. In the *advance* function, if the tag node is a virtual node composed of many vpatterns, *BCubeStep* is invoked to move the relative streams for finding the next answer set for the virtual node. In this case, each stream is scanned only once, and algorithm *XtwigStack* is therefore I/O and CPU optimal. Note that a twig pattern is a special case of Xtwig pattern, so the algorithm XtwigStack can conduct the matching of twig pattern too.

By defining the symbol *XCube*(vb_1, vb_2...) which represents vpatterns vb_1,vb_2 and so on, the processing of Xtwig pattern matching in Figure 2 can be evaluated by the patterns in Table 1.

4 Performance evaluation

In this section we present experimental results on the performance of the proposed holistic stream-based Xtwig matching algorithms using both real and synthetic data sets.

Table 1 Matching Xtwig pattern.

Pattern	Processing Xtwig pattern
(a)	XCube(A, B, C)//V
(b)	XCube(D//A, B, C)//V
(c)	XCube(C[//E]//B//D, A)//V
(d)	XCube(C[//E]//B[//D], A)//V
(e)	XCube(A, B, C)//V//D//E
(f)	XCube(A, B, C)//V//D[//E]//F//G

4.1 Experimental settings

We implement all the algorithms in Java 1.5. The system and hardware environment is a PC with 2.4 GHz Pentium CPU and 512 MB RAM running Windows XP operating system. The following three real-world and synthetic data sets are used for our experiments:

(1) DBLP [21], the well-known XML data set records large amounts of information on authors, papers, and other publications. The DTD of DBLP is not recursive;
(2) XMark[27], XMark is a well known XML benchmark. The XMark data set contains information about an auction site. It is "information oriented", and has many repetitive structures and fewer recursions;
(3) A synthetic XML data set with deep recursions is generated by using the following DTD.

```
<!ELEMENT a (b)* >
<!ELEMENT b (c|a)* >
<!ELEMENT c (d|b)* >
<!ELEMENT d (e)* >
<!ELEMENT e (f)* >
<!ELEMENT f (#PCDATA) >
```

The algorithms of *VertiDec*, *HoriDec*, and Rewriting Rules Method (*RRM*) [18] are implemented to evaluate the performance of the proposed algorithms in this paper for the matching of Xtwig patterns.

4.2 Performance study

4.2.1 Influence of RFOD

In order to evaluate the influence of reserved fan-out degree on performance, we design three vpattren queries *Query1–Query3* on the XMark data set. The RFODs of these vpatterns are 2–4. We also design four vpattern queries *Query4–Query8* on the synthetic data set. The RFODs of these queries are 2–6. These 8 queries are given as follows.

Query1: /regions//from[%asia]

Query2: /regions//from[%asia][%mailbox]

Query3: /regions//from[%asia][%mailbox][%mail]

Query4: /A//F[%B]

Query5: /A//F[%B][%C]

Query6: /A//F[%B][%C][%D]

Query7: /A//F[%B][%C][%D][%E]

Query8: /A//F[%B][%C][%D][%E][%F]

Figures 9 and 10 show the influence of RFOD change on the performance of algorithms *VeriDec*, *RRM* and *XtwigStack*. From these figures, we can see that *XtwigStack* performs better than *VeriDec*, and is independent of RFOD, while the performance of *VeriDec* gets worse as RFOD increases. *RRM* is not independent of RFOD and its performance is the worst. As RFOD increases, *VertiDec* has to scan more elements and merge more intermediate results, while *RRM* has to scan streams several times and evaluates multiple join operations.

4.2.2 Scalability vs document size

In order to examine scalability performance vs document size, two vpattern queries *Query*9 and *Query*10 are designed on the DBLP data set and the synthetic data set respectively. These two vpattern queries are given as follows.

Query9: /document//year[%proceedings]

Query10: /A//E[%C][%D]

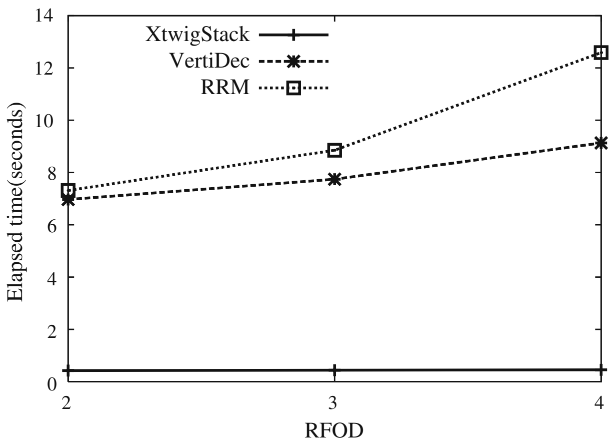


Figure 9 Response time vs RFOD on XMark.

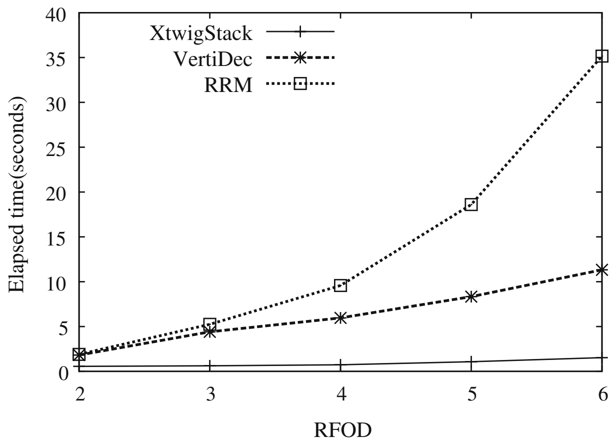


Figure 10 Response time vs RFOD on the synthetic data set.

Figures 11 and 12 show the scalability performance vs different document size of algorithms *XtwigStack*, *VeriDec* and *RRM*. From these figures we can see that *VeriDec* and *RRM* have similar scalability performance vs document size and *XtwigStack* has much better scalability performance vs document size than *VeriDec* and *RRM*. This is because the larger the document is, the more time is consumed for *VertiDec* and *RRM* to merge the intermediate results. Figures 13 and 14 show that the useful intermediate result rate of *VertiDec* decreases rapidly as the document size increases.

4.2.3 Scalability vs length of vpattern

In order to evaluate the scalability performance vs length of vpattern, four vpattern queries with different length, *Query11–Query14*, are designed on the XMark data

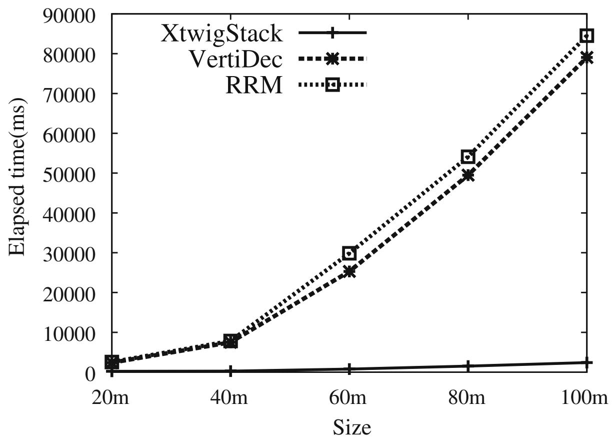
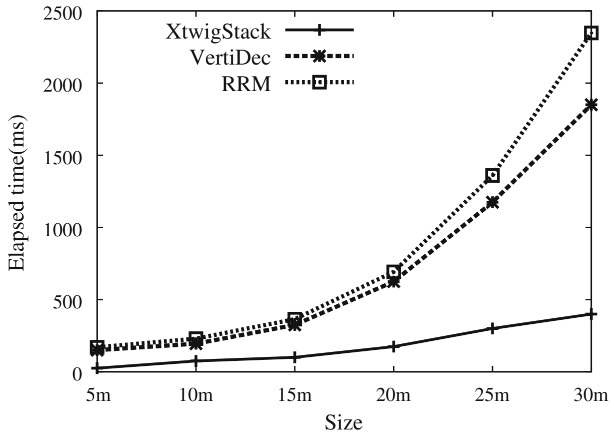


Figure 11 Scalability vs document size on DBLP.

Figure 12 Scalability vs document size on the synthetic data set.



set. These vpattern queries are given as follows.

Query11: /regions//from[%mailbox]

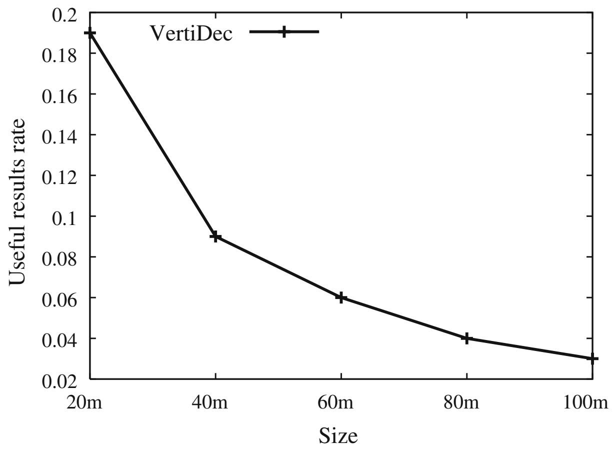
Query12: /people//person//street[%address]

Query13: /open_auctions//open_auction//time[%bidder]

Query14: /closed_auctions//closed_auction//description[%annotation]

Figures 15 and 16 show the scalability performance vs different length of vpattern. From Figure 15 we can see that *VertiDec* and *RRM* have similar performance, and are the worst, while they are slower than *XtwigStack*. In Figure 16, *XtwigStack* scans the fewest elements, while *VertiDec* and *RRM* scan the most. *VertiDec* scans many times the streams of the nodes under vpatterns when matching respective twig patterns, and scans lots of intermediate results for merge-joining. Since *HoriDec* avoids the scanning of the streams of the nodes under vpatterns, it performs better than *VertiDec* and *RRM*. *XtwigStack* scans the streams only once and does not generate any intermediate results, so it has the best performance.

Figure 13 Useful result rate vs document size on DBLP.



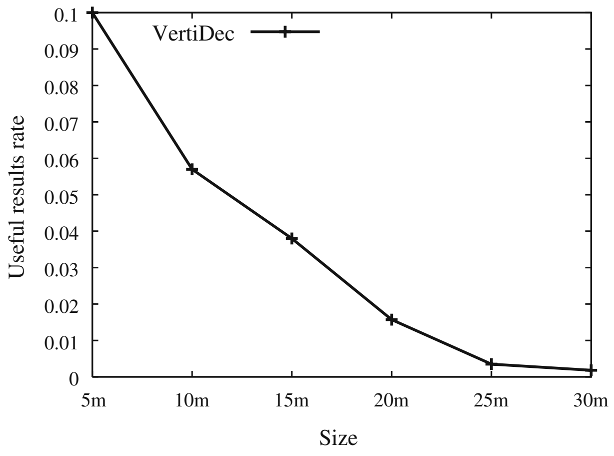


Figure 14 Useful result rate vs document size on the synthetic data set.

4.2.4 Xtwig patterns

Finally, we design the following xwtig query *Query*15 to compare the performances of *XtwigStack*, *VertiDec*, *HoriDec* and *RRM* on the XMark data set.

Query15:/regions//item[%asia]//name

Since *VertiDec* scans some streams many times and has a lot of intermediate results, it performs worst. *HoriDec* is better than *VertiDec*, because it scans the tag streams only once, except for the vnode stream. However, it still has lots of useless intermediate results. *XtwigStack* is a holistic algorithm, and it scans all the streams only once, and has no intermediate results. Figure 17 shows the advantages of *XtwigStack* in terms of elapsed time. Figure 18 shows that as the document size increases, the number of elements scanned by *VertiDec* increases fastest, while the number of elements scanned by *HoriDec* are less than that by *VertiDec*, and larger than that by *XtwigStack*.

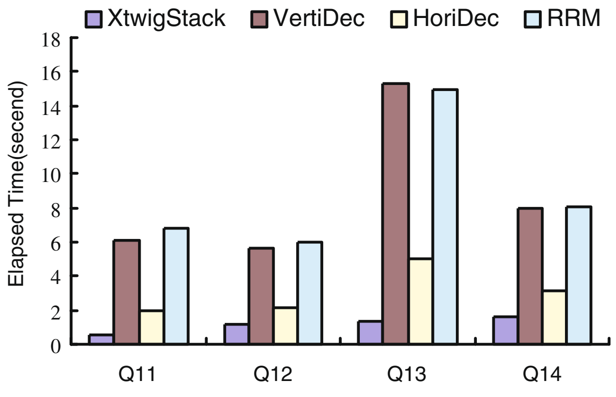


Figure 15 Elapsed time vs different length of vpattern.

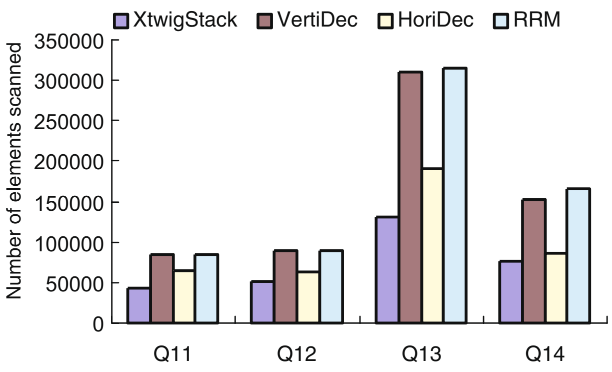


Figure 16 Number of scanned elements vs different length of vpattern.

5 Related work

With the increasing popularity of XML, query processing and optimization for XML databases has attracted a lot of research interest. The work of Lore [17], Timber [12], and Natix [10] has considered various aspects of retrieving such data.

Structural join is essential to XML query processing because XML queries usually impose certain structural relationships (e.g. p-c or a-d relationships). For binary structural join, Zhang et al [2] propose a multi-predicate merge join (MPMGJN) algorithm based on region labelling of XML elements. The later work by Al-Khalifa et al [1] gives a stack-based binary structural join algorithm, called Stack-Tree-Desc/Anc which is optimal for an a-d and p-c binary relationship. A twig pattern is a query with selection predicates only containing descendant and child axes. There are some work to conduct the matching of twig patterns. N. Bruno et al [6] propose a holistic twig join algorithm, namely TwigStack, to avoid producing large intermediate results. Extended Dewey [15], TJFast [14] and Twig2Stack [8] are newly proposed to evaluate twig patterns, and they are nearly the best methods to solve this series of problems of twig pattern matching. A twig pattern only contains predicates with

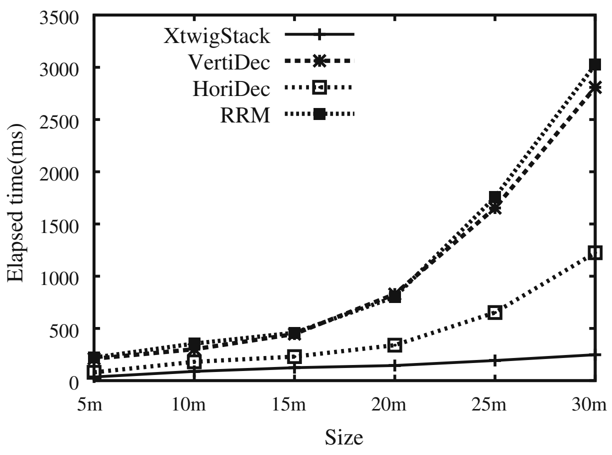
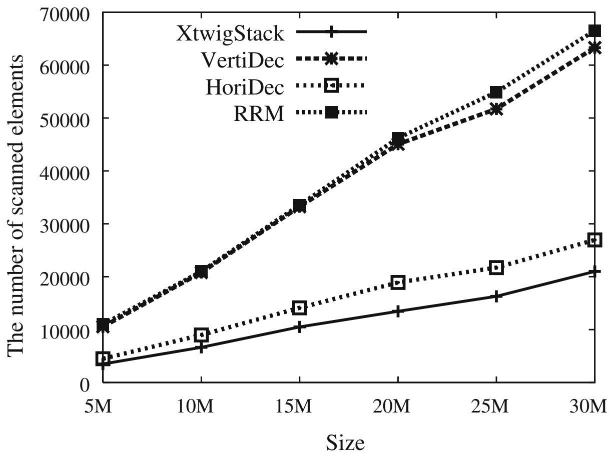


Figure 17 Elapsed time vs document size.

Figure 18 Number of scanned elements vs document size.



descendant and child axes, while an Xtwig pattern is a more complex query which may contain predicates with ancestor and/or parent axes. For the work of reverse axes, Olteanu et al [18] propose rewriting rules to transform absolute XPath location paths with reverse axes into equivalent reverse-axis-free ones, and rewrite all axes to a minimum set of forward axes.

6 Conclusions

XML queries are more complex and may contain predicates with ancestor or parent axes, but there is no research work on stream-based processing queries with ancestor or parent axes so far. We propose a new query pattern Xtwig to meet the requirement and analyze the Xtwig pattern in details. We analyze the characteristics of Xtwig pattern matching, and propose a new data structure called *XCube* to conduct the vpattern matching, which is the core problem of the Xtwig pattern matching. We also develop an *XtwigStack* algorithm to solve the queries with ancestor or parent axes in predicates. Finally we present experimental results on a range of real and synthetic data, to complement our analytical results. The performance study shows that *XtwigStack* is I/O and CPU optimal, and every stream is scanned only once, and there are not useless intermediate results.

Acknowledgements This research was supported by the National Natural Science Foundation of China (Grant No. 60573089 and 60773221) and National Basic Research Program of China (Grant No. 2006CB303103).

References

1. AL-Khalifa, S., Jagadish, H.V., Kouda, N., Patel, J.M., Srivastava, D., Wu, Y.: Structural joins: a primitive for efficient XML query pattern matching. In: Proc. 18th Int. Conf. Data Engineering (ICDE'02), pp. 141–152. IEEE Computer Society, San Jose (2002)

2. Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Zhang, C., Naughton, J.F., DeWitt, D.J., Luo, Q., Lohman, G.M.: On supporting containment queries in relational database management systems. In: Proc. 27th ACM SIGMOD Int. Conf. Management of Data (SIGMOD'01), pp. 425–436. ACM, Santa Barbara (2001)
3. Arion, A., Bonifati, A., Manolescu, I., Pugliese, A.: Path summaries and path partitioning in modern XML databases. World Wide Web **11**(1), 117–151 (2008)
4. Barton, C., Charles, P., Goyal, D., Raghavachari, M., Fontoura, M., Josifovski, V.: Streaming XPath processing with forward and backward axes. In: Proc. 19th Int. Conf. on Data Engineering (ICDE'03), Bangalore, pp. 455–466. IEEE Computer Society, Bangalore (2003)
5. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: Proc. 28th ACM SIGMOD Int. Conf. Management of Data (SIGMOD'02), pp. 310–321. ACM, Madison (2002)
6. Bruno, N., Srivastava, D., Koudas, N.: Holistic twig joins: optimal XML pattern matching. In: Proc. 28th ACM SIGMOD Int. Conf. Management of Data (SIGMOD'02), pp. 310–321. ACM, Madison (2002)
7. Chan, C.Y., Ni, Y.: Piggyback optimization of XML data dissemination. In: Proc. 23rd Int. Conf. Data Engineering (ICDE'07), pp. 1454–1455. IEEE Computer Society, Istanbul (2007)
8. Chen, S., Li, H., Tatemura, J., Hsiung, W., Agrawal, D., Candan, K.S.: Twig2Stack: bottom-up processing of generalized treepattern queries over XML documents. In: Proc. 32nd Int. Conf. Very Large Data Bases (VLDB'06), pp. 283–294. ACM, Seoul (2006)
9. Chen, T., Ling, T.W., Chan, C.: Prefix path streaming: a new clustering method for optimal XML twig pattern matching. In: Proc. 15th Int. Conf. Database and Expert Systems Applications (DEXA'04), pp. 801–811. Springer, Zaragoza (2004)
10. Fiebig, T., Helmer, S., Kanne, C.C., Moerkotte, G., Neumann, J., Schiele, R., Westmann, T.: Anatomy of a native XML base management system. VLDB J. **11**(4), 292–314 (2003)
11. Florescu, D., Kossmann, D.: Storing and querying xml data using an rdmbs. IEEE Data Eng. Bull. **22**(3), 27–34 (1999)
12. Jagadish, H.V., AL-Khalifa, S., Chapman, A., Lakshmanan, L.V., Nierman, A., Paparizos, S., Patel, J.M., Srivastava, D., Wu, Y., Yu, C.: TIMBER: a native XML database. VLDB J. **11**(4), 274–291 (2002)
13. Jiao, E., Ling, T.W., Chan, C.Y.: PathStack : a holistic path join algorithm for path query with not-predicates on XML data. In: Proc. 10th Int. Conf. Database Systems for Advanced Applications (DASFAA'05), pp. 113–124. Springer, Beijing (2005)
14. Lu, J., Chen, T., Ling, T.W.: TJFast: effcient processing of XML twig pattern matching. In: Proc. 14th Int. Conf. World Wide Web (WWW'05), pp. 1118–1119. ACM, Chiba (2005)
15. Lu, J., Ling, T.W., Chan, C.-Y., Chen, T.: From region encoding to extended dewey: on efficient processing of XML twig pattern matching. In: Proc. 31st Int. Conf. Very Large Data Bases (VLDB'05), pp. 193–204. ACM, Trondheim (2005)
16. Lv, J., Wang, G., Yu, J.X., Yu, G., Lu, H., Sun, B.: Performance evaluation of a DOM-based XML database: storage, indexing and query optimization. In: Proc. 3rd Int. Conf. Web-Age Information Management (WAIM'02), pp. 13–24. Springer, Beijing (2002)
17. McHugh, J., Abiteboul, S., Goldman, R., Quass, D., Widom, J.: Lore: a database management system for semistructured data. SIGMOD Rec. **26**(3), 54–66 (1997)
18. Olteanu, D., Meuss, H., Furche, T., Bry, F.: XPath: looking foward. In: Proc. the EDBT Workshop on XML Data Management, pp. 109–127. Matfyzpress, Prague (2002)
19. Qin, L., Yu, J.X., Ding, B.: TwigList: make twig pattern matching fast. In: Proc. 12th Int. Conf. Database Systems for Advanced Applications (DASFAA'07), pp. 850–862. Springer, Bangkok (2007)
20. Tatarinov, I., Viglas, S., Beyer, K., Shekita, E., Shanmugasundaram, J., Zhang, C.: Storing and querying ordered XML using a relational database system. In: Proc. 28th ACM SIGMOD Int. Conf. Management of Data (SIGMOD'02), pp. 204–215. ACM, Madison (2002)
21. University of Washington (2002) University of Washington XML Repository. http://www.cs.washington.edu/research/xmldatasets/
22. Wang, H., Li, J., Wang, H.: Clustered chain path index for XML document: efficiently processing branch queries. World Wide Web **11**(1), 153–168 (2008)
23. Wang, G., Sun, B., Lv, J., Yu, G.: RPE query processing and optimization techniques for XML databases. J. Comput. Sci. Technol. **19**(2), 224–237 (2004)
24. Wang, Y., Xing, C., Zhou, L.: Managing and querying of videos by semantics in digital library—a semantic model SemTTE and its XML-based implementation. In: Proc. 9th Int. Conf. Asian Digital Libraries (ICADL'06), pp. 519–522. Springer, Kyoto (2006)

25. Wong, K.-F., Yu, J.F., Tang, N.: Answering XML queries using path-based indexes: a survey. World Wide Web **9**(3), 277–299 (2006)
26. W3C (1999) XPath. http://www.w3.org/TR/xpath
27. XMARK (2003) XMARK. http://monetdb.cwi.nl/xml
28. Yoshikawa, M., Amagasa, T.: XRel: a path-based approach to storage and retrieval of XML documents using relational databases. ACM Trans. Internet Technol. **1**(1), 110–141 (2001)