Science in China Series F: Information Sciences

© 2009 SCIENCE IN CHINA PRESS

Springer

www.scichina.com
info.scichina.com
www.springer.com/scp
www.springerlink.com

Efficient processing of partially specified twig pattern queries

ZHOU JunFeng[1,2], MENG XiaoFeng[1†] & LING TokWang[3]

[1] School of Information, Renmin University of China, Beijing 100872, China;

[2] Department of Computer Science and Technology, Yanshan University, Qinhuangdao 066004, China;

[3] School of Computing, National University of Singapore, 117590, Singapore

As huge volumes of data are organized or exported in tree-structured form, it is quite necessary to extract useful information from these data collections using effective and efficient query processing methods. A natural way of retrieving desired information from XML documents is using twig pattern (TP), which is, actually, the core component of existing XML query languages. Twig pattern possesses the inherent feature that query nodes on the same path have concrete precedence relationships. It is this feature that makes it infeasible in many actual scenarios. This has driven the requirement of relaxing the complete specification of a twig pattern to express more flexible semantic constraints in a single query expression. In this paper, we focus on query evaluation of partially specified twig pattern (PSTP) queries, through which we can reap the most flexibility of specifying partial semantic constraints in a query expression. We propose an extension to XPath through introducing two Samepath axes to support partial semantic constraints in a concise but effective way. Then we propose a stack based algorithm, pTwigStack, to process a PSTP holistically without deriving the concrete twig patterns and then processing them one by one. Further, we propose two DTD schema based optimization methods to improve the performance of pTwigStack algorithm. Our experimental results on various datasets indicate that our method performs significantly better than existing ones when processing PSTPs.

XML database, query processing, partially specified twig pattern, holistic twig join, XPath

1 Introduction

As a de facto standard for information representation and exchange over the Internet, XML has been used extensively in many applications. Query capabilities are provided through twig patterns (TPs), which are the core components for standard XML query languages, e.g. XPath (http://www.w3.org/TR/xpath20/) and XQuery (http://www.w3.org/TR/xquery/). A TP can be naturally represented as a node-labeled tree, where each edge denotes either Parent-Child (P-C) or Ancestor-Descendant (A-D) relationship. For example, the TP written in XPath format, Q_1: "//book[.//author/name= 'Mike']/title", selects

Received January 18, 2008; accepted November 5, 2008
doi: 10.1007/s11432-009-0152-3

†Corresponding author (email: xfmeng@ruc.edu.cn)

Supported partially by the National Natural Science Foundation of China (Grant No. 60833005), the National High-Tech Research & Development Program of China (Grant Nos. 2007AA01Z155, 2009AA011904), and the National Basic Research Program of China (Grant No. 2003CB317000)

title elements which are children of some book elements written by an author named "Mike". While many existing algorithms[1−4] can efficiently process a given TP, an inherent restrictive feature of TP is that a concrete precedence order between the nodes in every path of the query expression should be clearly specified. In Q_1, for example, book should be an ancestor of author; thus Q_1 can only be used to retrieve information from D_2 in Figure 1, not D_1.

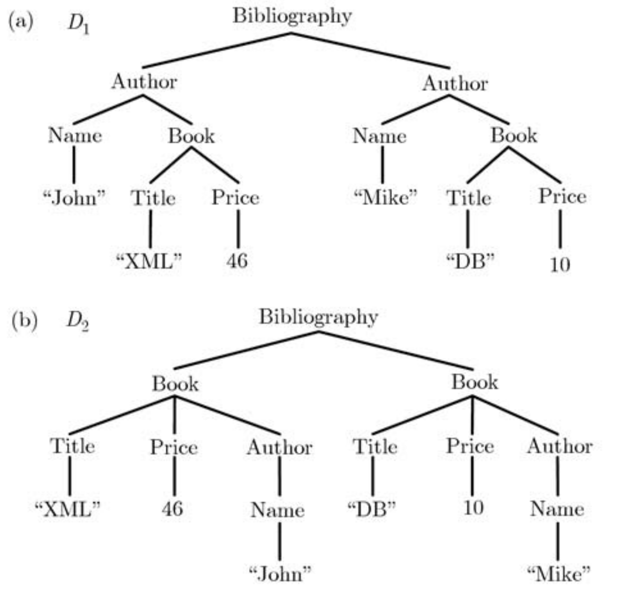


Figure 1 An example data organization of different hierarchical structures.

In fact, XQuery and XPath allow no concrete precedence order between query nodes. For example,
Q_2: //author[child::name="Mike"]/descendant-or-self::*/ancestor-or-self::book/child::title,
can also be used to find title elements which are children of book elements which are written by an author named "Mike" from D_2. Although book and author should be on the same path, we know nothing about which one is the ancestor of the other or vice versa. The benefits of using Q_2 is obvious; it can be used to retrieve useful information from both D_1 and D_2 without considering their structural heterogeneity. However, such a query cannot be easily evaluated. Although Olteanu et al.[5,6] show that using special rules, XPath queries with reverse axes, e.g. Q_2, can be equivalently rewritten as a set of TPs, they also show that this transformation may lead to an exponential blowup of the number of TPs. Further, Gottlob et al.[7] show that the combined complexity of XPath is P-hard (i.e., hard for polynomial time).

Usually in many scenarios, we cannot specify the precedence relationships of query nodes when we formulate query expression. 1) The document structure is not available. 2) Extracting desired information from XML documents of structural heterogeneity. It is complex to use TPs in conjunction with data integration mapping rules between a global schema and local schema, and may cause errors since maintaining the mapping relationship may involve extensive manual intervention. 3) The change of business strategy and corporate environments may cause the data to be organized with a different structure, which makes existing path expression that depends on particular hierarchical structure no longer feasible.

Although keyword based methods[8,9] can be used freely without schema knowledge, only limited semantic constraints can be contained in such a query expression. Query relaxation based methods[10,11] will also produce a large number of relaxed query expressions by relaxation operations, which will further result in too many approximate answers.

In ref. [12], the notion of partially specified twig pattern (PSTP) was proposed to tackle this problem. Compared with TP, PSTP provides us with the most flexibility: 1) we can specify the full structural constraints if the schema or document structure is available; 2) we can specify just keywords to retrieve desired information if the schema or document structure is not available; and 3) we can make full use of whatever partial knowledge we have to specify more flexible semantic constraints.

As a PSTP may correspond to multiple TPs, a naive evaluation method[12] for PSTP is as follows:

Let Q be a PSTP, $Q_1, Q_2, \ldots, Q_n$ be TPs derived from Q, $R, R_1, R_2, \ldots, R_n$ be the answer sets of $Q, Q_1, Q_2, \ldots, Q_n$ on an XML document D, respectively. Then $R = \bigcup_{i\in[1,n]} R_i$.

While PSTP can express more flexible semantic information, it is not feasible to directly apply it in practice, because n may be too large and thus has great impact on query performance. Our contributions are as follows:

1. We propose an extension to XPath by introducing two Samepath axes to enhance the expressiveness of XPath.

2. We give a detailed analysis of the challenges of evaluating PSTP, and then propose an efficient algorithm, pTwigStack, to process a PSTP holistically. Our method possesses the following three features: scanning only once; no redundant output; and bounded space complexity.

3. We implemente related algorithms and make comparison between our methods and existing ones. Experimental results demonstrate that our method is efficient in terms of various evaluation metrics.

2 Preliminaries

2.1 Data model and numbering schemes

An XML document can be modeled as a node-labeled tree, where nodes represent elements, attributes and text data, while edges represent direct nesting relationship between nodes in the tree. Formally, tree $T = (V, E, \Sigma, M)$, where V is the node set and there is a unique root node R in V, E is the edge set, and no cycle among the edges is permitted, Σ is an alphabet of labels and text values, M is a function that maps each node to its label.

Most XML query processing algorithms use a special positional representation to represent an element; we use pre(v), which is compatible with preorder numbering, to denote the numerical id assigned to node v, in the sense that if a node v_1 precedes a node v_2 in the preorder left-to-right depth-first traversal of the tree, then pre($v1$) <pre($v2$). This positional representation can be easily implemented using either region encoding[13] or Dewey ID[14]. In the first case, pre(v) equals a tuple of three fields: $(start; end; level)$. We say that element u is an ancestor of element v if and only if $u.start < v.start < u.end$. u is the parent of v if and only if $u.start < v.start < u.end$ and $u.level = v.level - 1$. In the second case, if u is the root node, $label(u) = 1$, otherwise, $label(u) = label(v).x$, where u is the xth child of v, and "." in "$label(v).x$" is the concatenation operator which is different from the "." in $u.start$, $u.end$ and $u.level$ in the previous sentences.

2.2 Twig pattern (TP) matching

TPs are used to match data fragments from XML data. The edges in a TP indicate either Parent-Child (P-C) or Ancestor-Descendant (A-D) relationship of query nodes. For convenience, we use "node" to denote query node and "element" to denote data element in an XML document.

Matching a TP against an XML document is to find all occurrences of the TP in the database. Formally, given a TP Q and an XML document D, a match of Q in D is identified by a mapping from nodes in Q to elements in D, such that: i) the query node predicates are satisfied by the corresponding database elements; ii) the structural relationships (P-C or A-D) among query nodes are satisfied by the corresponding database elements. The answer to query Q with n nodes can be represented as a n-array tuple $(e_1, e_2, \ldots, e_n)$ which consists of the database elements that identify a distinct match of Q in D.

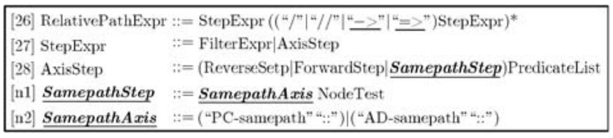
3 The Samepath axis

Definition 1 (SamepathStep). A SamepathStep returns a sequence of nodes that are reachable from the context node via a specified axis (PC-samepath or AD-samepath axis). The SamepathStep has two parts: an axis, which defines the "direction of movement" for the step, and a node test, which selects nodes based on their kind and name. The resulting node sequence is returned in document order.

Definition 2 (PC-samepath axis ("$\rightarrow$")). The PC-samepath axis contains the set of data elements that are children and parent of the context node.

Definition 3. (AD-samepath axis ("$\Rightarrow$")). The AD-samepath axis is the transitive closure of the PC-samepath axis; it contains the set of data elements that are descendant and ancestor of the context node.

There are totally 82 rules in the current XPath grammar, among which only rules 26th and 28th have to be modified. As shown in Figure 2, SamepathStep is further defined by the new rules [n1] and [n2]. It consists of two axes, i.e. PC-samepath and AD-samepath. Except the two axes, we also introduce two separators, "$\rightarrow$" and "$\Rightarrow$", to indicate the semantic constraints of being on the same path. Thus a path expression consisting of a series of step expressions may be separated by "$\rightarrow$" or "$\Rightarrow$". For example, "book$\Rightarrow$author" is short for "child::book/AD-samepath::author" and will return all authors that are descendant or ancestor of book. As a result, there are totally 84 rules in the extended XPath grammar with two of them modified and two newly added. This extension to XPath provides users with the ability to specify the semantic constraints of two nodes on the same path in a very simple way. The Samepath axis is similar to the current XPath axes insofar as it returns a set of nodes corresponding to the context node. We can use a node test and predicates to filter those undesired nodes.



```
[26] RelativePathExpr ::= StepExpr (("/"|"//"|"->"|"=>")StepExpr)*
[27] StepExpr         ::= FilterExpr|AxisStep
[28] AxisStep         ::= (ReverseSetp|ForwardStep|SamepathStep)PredicateList
[n1] SamepathStep     ::= SamepathAxis NodeTest
[n2] SamepathAxis     ::= ("PC-samepath" "::")|("AD-samepath" "::")
```

Figure 2 EBNF grammar for the extended XPath.

If both A and B are query nodes, we use "$A \rightarrow B$" to denote that A is the parent node of B or vice versa, "$A \Rightarrow B$" denotes that A and B are on the same path. If e_A and e_B are data elements of tag A and B, "$e_A \rightarrow e_B$" or "$e_A \Rightarrow e_B$" denotes that e_A and e_B satisfy the structural constraints of "$A \rightarrow B$" or "$A \Rightarrow B$", respectively.

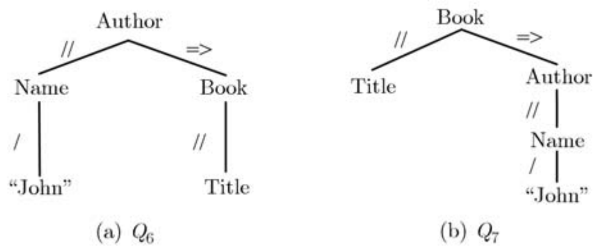


Figure 3 Two partially specified twig patterns.

Example 1. Consider query Q_3: "find the title of the books that have an author named 'John' from the two documents in Figure 1". For D_1, the query should be Q_4: //author[.//name="John"]//book//title; for D_2, the query should be Q_5: //book[.//author//name="John"]//title. With Samepath axis, Q_4 and Q_5 can be replaced by a PSTP with Samepath axis in either Figure 3(a) or Figure 3(b), which can be written in extended XPath format as Q_6: author[.//name="John"]$\Rightarrow$book//title or Q_7: book[.$\Rightarrow$author//name="John"]//title. Further, PSTP expression can be seamlessly incorporated into XQuery. For Q_3, our solution using the Samepath axis is as follows.

```
for $a in doc()//author,
      $t in $a⇒book//title
where $a//name="John"
return $t
```

Although the Samepath axis can be used to specify the semantic constraints of two nodes on the same path, it only involves the relationship of two nodes. Considering the PSTP in Figure 4(a), we can easily understand that the semantics of $A \Rightarrow B$ equals $A//B$ or $B[.//A]$. Further, considering the PSTP expression $A \Rightarrow B \Rightarrow C$ and its derived TPs in Figure 4(b), where A and B should be on the same path and B and C should be on the same path, but not necessary for A and C. $A \Rightarrow B \Rightarrow C$ may correspond to $B[.//A]//C$ since elements with tag A and C are not required to be on the same path. The PSTP in Figure 4(c) has not been specified with a concrete root node, and the related TPs are not shown due to limited space. A problem we should notice is that node B in the PSTP in Figure 4(c) should be on the same path with A, and D should be on the same path with C, which means that either B or D or both of them may be ancestors of A and C.

4 Problems and our solutions

As we know from the above description that some PSTPs may not have been specified with a concrete root node, like the one in Figure 4(c), which corresponds to a keyword-like query, the difference is that each part of the PSTP is also a path expression that may contain partial semantic constraints.

Such a query can be evaluated easily by extending existing keyword based methods. We present in this paper only query processing method for PSTPs that are specified with concrete root nodes, like the ones in Figure 4(a) and (b). Moreover, all results presented in this section are based on A-D and AD-samepath relationships. In this section, we first analyze existing stack based TP matching method, and then show challenges of evaluating PSTP.

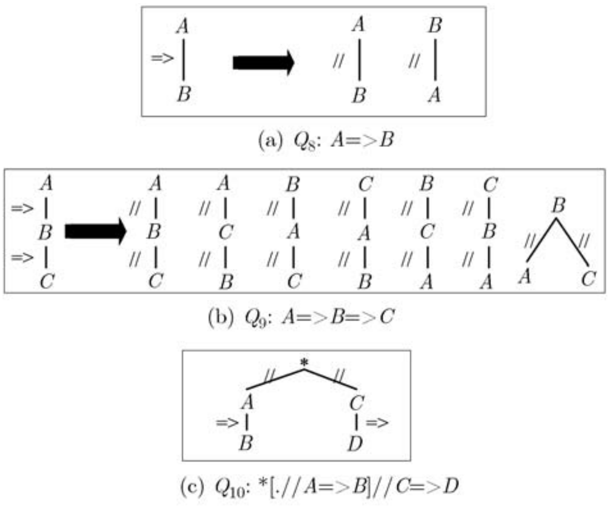


Figure 4 Three PSTPs and their corresponding TPs.

4.1 Insight into the TwigStack algorithm

In the TwigStack algorithm, each query node q in a TP is associated with a stack S_q, a cursor C_q and a data stream T_q. C_q can point to some elements in T_q, especially, we say that C_q is NULL if all elements in T_q are processed, and C_q is also used to denote the element it points to. Before executing, all cursors point to the first elements in each data stream. We use Advance(C_q) to make C_q pointing to the next element. The self-explaining functions isRoot(q) and isLeaf(q) are used to determine whether q is a root node or a leaf node. The function children(q) is used to return all the child nodes of q and parent(q) is used to return the parent node of q.

TwigStack works in two steps. In the first step, it repeatedly calls getNext(root) to get a query node q with Solution Extension[1)], and then C_q is processed by either being pushed into stack as a useful element, or being skipped as a useless element. Such operations will repeat until all elements of leaf nodes are processed. At the end of this step, TwigStack will produce all path solutions. In the second step, all produced path solutions are merge-joined to get the final answers. When all edges in the TP are A-D edges, TwigStack guarantees that both its time and I/O complexity are independent of the size of partial matches to any root-to-leaf path.

In the TwigStack algorithm, the objective of getNext(root) is finding the first element that may participate in final answers from the elements that are still not being processed, and Solution Extension is used here to guide the execution of getNext(root). As shown in Figure 5, Q_{AD} is a TP with just A-D edges. Suppose B is returned by getNext(A). From the definition of Solution Extension we know that it guarantees that the structural constraints below B are satisfied, which is denoted by (1) with arrows in Figure 5(a). Thus whether C_B is a useful element is determined by just checking whether the top element in S_A is an ancestor of C_B, which is denoted by (2) with an arrow. Obviously, Solution Extension, (1) with arrows, works in down direction in TwigStack; while useful element, (2) with an arrow, works in up direction.

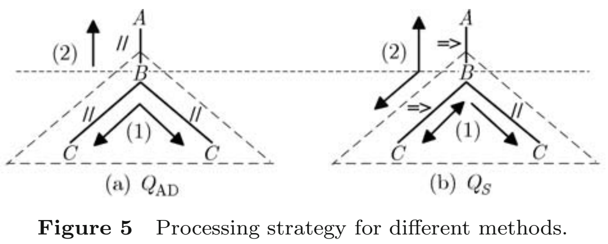


Figure 5 Processing strategy for different methods.

Example 2. For D_3 and Q_{11} in Figure 6, the first call of getNext(A) in TwigStack will return B with cursor C_B pointing to b_1, which means that all descendant nodes of B have Solution Extension. Among all cursors of descendant of B, C_B has the smallest preorder value. However, as no element in S_A is ancestor of b_1, b_1 is skipped as a useless

1) A node q has a Solution Extension if there is a solution for the sub query rooted at q composed entirely of the cursor elements of the query nodes in the sub query. Note that if q has a Solution Extension, C_q is the ancestor of all cursor elements in the sub query tree nodes, and pre(C_q) is smaller than all other elements of query nodes in the subtree rooted at q, based on the strictly nested property of XML data.

element instead of being pushed into S_B. In this example, the useful elements are a_1, b_2, b_3 and c_1. After c_1 is processed, the stack encoding is shown in Figure 6(c) and then two path solutions are produced, they are (a_1, b_2, c_1) and (a_1, b_3, c_1).

Thus we have the following observations.

1. Query nodes in a TP are processed with a special order in existing methods, i.e. left-to-right depth-first traversing the TP. For Q_{11}, the order is A, B, C.

2. Query node q returned by getNext(root) must have a Solution Extension, from which we can get an element C_q for further processing. C_q can be either pushed into stack if it can participate in final answers, or skipped directly as a useless one.

3. If q is the root node or otherwise, C_q satisfies the structural relationship of edge $<\text{parent}(q), q>$ with the top element in stack $S_{\text{parent}(q)}$ (if such element exists), then C_q is a useful element, which means that C_q can participate in final answers.

4. All elements in the same stack (from bottom to top) are guaranteed to lie on a root-to-leaf path according to the given XML document, and elements in different stacks are linked together through pointers (from descendant to ancestor).

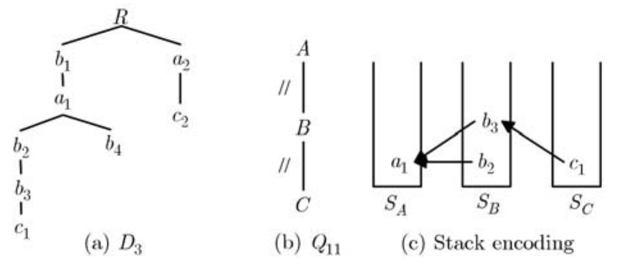


Figure 6 An example of query processing of the TwigStack algorithm.

4.2 Challenges and our solutions

Although TwigStack guarantees that all elements are scanned only once and no redundant output, the four aspects described above hold only for a TP, not a PSTP. Similarly, we need to resolve the following problems when evaluating a PSTP.

4.2.1 Query node processing order. As a PSTP may correspond to multiple TPs, e.g. Q_9 in Figure 4 corresponds to 7 TPs, a naive way is processing each one of them using existing methods. Obviously, this will cause high processing cost. In our method, we process a Samepath axis without decomposing it into two P-C or A-D axes, thus we can process a PSTP without considering the derived TPs. In this way, the query node processing order for Q_9 is A, B, C.

4.2.2 Returning node. For the same reason, when processing a PSTP, Solution Extension cannot be used correctly in getNext as an indicator to tell whether an element is useless. For example, consider Q_{11} and D_3 in Figure 6. If the current cursors C_A, C_B and C_C point to a_2, b_4 and c_2, we can skip b_4 directly since b_4 appears before c_2 and it is not the ancestor of c_2. But for Q_9 in Figure 4, Q_{11} is just one of the derived TPs of Q_9. Although b_4 is useless for Q_{11}, for Q_9, however, we cannot say that b_4 is useless by just checking Q_{11}, b_4 may be useful for other derived TPs since B also appears at leaf node in two derived TPs of Q_9. In this case, we should return B for further processing to avoid losing answers of Q_9. For this problem, we propose a notion partial solution extension (PSE) to guide the execution of getNext. Intuitively, if q has a PSE, q corresponds to at least one derived TP in which q has a Solution Extension, which means that C_q may participate in final answers of these derived TPs.

4.2.3 Pushing element. In TwigStack, an element C_q corresponding to q returned by getNext(root) can be pushed into stack if C_q can participate in at least one final result, i.e. C_q is a useful element. Similarly, in our method, C_q will be pushed into stack if and only if it is a useful element. Then what is a useful element for PSTP? In TwigStack, we only need to check whether the top element in $S_{\text{parent}(q)}$ satisfies the structural relationship of edge $<\text{parent}(q), q>$ with C_q. However, we need to check other related elements for PSTP so as not to lose any result. For example, consider Q_9 in Figure 4(b) and D_4 in Figure 7(a). Obviously, a_1, b_1 and c_1 are useful elements since $B[.//A]//C$ and $B//C[.//A]$ are two derived TPs of Q_9; thus the three elements are pushed into S_A, S_B and S_C. After that, C_A, C_B and C_C point to a_2, b_2 and c_2, respectively. Because A appears at leaf node of three derived TPs of Q_9, after a_2 is returned by getNext(root), we need to check whether a_2 can participate in final answers with elements in stacks.

Obviously, a_2 is a useful element for Q_9. For another example, consider D_3 in Figure 6 and Q_9 in Figure 4. Suppose C_A, C_B and C_C point to a_2, b_4 and c_2, respectively. Because B also appears at leaf node of some derived TPs, b_4 is first returned by getNext(root) for further processing. Although b_4 can satisfy the structural constraint of $A \Rightarrow B$ with top(S_A), i.e. $a_1 \Rightarrow b_4$, it is a useless element for Q_9 since no element with tag C can satisfy the structural constraint of $B \Rightarrow C$. We propose a notion, Useful Element, in the next section as a metric to tackle this problem.

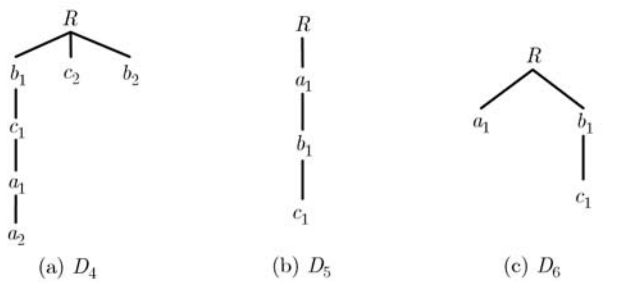


Figure 7 Three XML documents.

4.2.4 Stack organization. For a PSTP, a query node is the ancestor of another one in some derived TPs; in other derived TPs, however, it may be a descendant of that node. In our method, data elements in the same stack (from bottom to top) lie on a root-to-leaf path in an XML document, and data elements in different stacks are linked together through pointers from elements in the stack of descendant query node to elements in stack of ancestor query node. For example, consider Q_9 in Figure 4 and D_4 in Figure 7. a_1, b_1 and c_1 satisfy the structural constraint of $A \Rightarrow B \Rightarrow C$, so they should be pushed into S_A, S_B and S_C, respectively. Although a_1 is a descendant of b_1, the pointer between them starts with b_1 and ends at a_1.

5 Related notions

From the discussion in section 4 we know that the first problem of evaluating PSTPs is: in what condition a query node q should be returned by getNext(root)? For example, Q_S in Figure 5(b) is a PSTP, B and C have the Samepath relationship, so each of them can be a leaf node, which is denoted by (1) with arrows. Suppose B is returned by getNext(A) in our method. Then C_B appears before C_C and maybe they are not on the same path. We cannot say that, in this case, C_B is useless. So the objective of getNext(root) can be stated as not checking whether an element is useful, but whether it is useless. If it is useless, the element will be skipped directly, otherwise, it will be returned for further processing. In the following definition, hasSE(q) checks whether q has a Solution Extension.

Definition 4 (partial solution extension (PSE)). Let Q be a PSTP, we say that a query node q of Q has a PSE if and only if q satisfies any one of the following conditions:

1. isLeaf(q)$\wedge C_q \neq$NULL, or,
2. for each $q' \in$ children(q)

(1) $q//q' \wedge C_q//C_{q'} \wedge$hasSE($q'$), or,

(2) $q \Rightarrow q' \wedge$hasSE(q') $\wedge$ (pre(C_q) $<$ pre($C_{q'}$) $\vee$ $C_{q'} \Rightarrow C_q$).

Case 1 is straightforward. Case 2 consists of two independent conditions. (1) means that if q and q' have A-D relationship, the current elements of q and q', i.e. C_q and $C_{q'}$, should satisfy the structural constraint of $q//q'$, i.e. $C_q//C_{q'}$, at the same time, q' should have Solution Extension. For (2), consider D_5 in Figure 7(b) and Q_9 in Figure 4(b), and suppose C_A, C_B and C_C point to a_1, b_1 and c_1, respectively. As B and C have the Samepath relationship, C has a PSE and $b_1 \Rightarrow c_1$, which is equal to $q \Rightarrow q' \wedge$hasSE(q') $\wedge C_q \Rightarrow C_{q'}$, so we say that B has a PSE since b_1 may participate in final answers. Consider D_6 in Figure 7(c) and Q_9, and suppose C_A, C_B and C_C point to a_1, b_1 and c_1, respectively. In this case, A and B have the Samepath relationship, B has a PSE, and a_1 and b_1 are not on the same path, which is equal to $q \Rightarrow q' \wedge$hasSE(q') $\wedge$ pre(C_q) $<$ pre($C_{q'}$) $\wedge \neg(C_q \Rightarrow C_{q'})$. However, we cannot say that a_1 is useless just according to the structural relationship between a_1 and b_1, because A can be a leaf node in some derived TPs shown in Figure 4(b). In such a case, A has a PSE and the union of the two cases is equal to (2).

By the definition of PSE, we can easily check whether a query node q has PSE. However, C_q may not participate in final answers and we need to check whether C_q is useful, which forms the second problem, i.e. in what condition should C_q be pushed into stack S_q? From (2) with arrow in Figure 5(b) we know that this operation should include checking the satisfiability of all Samepath relationships that are directly related with B from both up and down directions. We propose a notion, Useful Element, to answer this problem, where top(S_q) returns the top element from stack S_q and isEmpty(S_p) checks whether S_p is empty. For two elements e_p and e_q, isHold($e_p, e_q, \langle p, q\rangle$) is used to check whether e_p and e_q satisfy the structural constraint between query nodes p and q.

Definition 5 (Useful element)**.** An element pointed by C_q (q is returned by getNext(root) in our method) is a useful element if and only if any one of the following conditions holds:

1. isRoot(q)$\wedge$hasSE(q), or,
2. isHold(top($S_{\text{parent}(q)}$)$, C_q, \langle\text{parent}(q), q\rangle$)$\wedge$
(1) hasSE(q), or,
(2) for each child q' of q, if isHold($C_q, C_{q'}, \langle q, q'\rangle$) = FALSE, then isHold($C_q$,top($S_{q'}$)$, \langle q, q'\rangle$) = TRUE

Intuitively, C_q is useful means that it can participate in final answers. The fact we should understand is that q has a PSE does not means that it must have a Solution Extension. Because the Samepath axis is bidirectional in essence, if q has not a Solution Extension, we need to check for each child node q' of q, whether there exists in $S_{q'}$ elements that can satisfy the structural constraint between q and q' with C_q.

Case 1 means that if q is the root node and q has Solution Extension, then C_q is a useful element. Case 2 means that if q is not the root node, C_q must satisfies the structural constraint between parent(q) and q with top($S_{\text{parent}(q)}$); moreover, q must have a Solution Extension (shown as "(1)"), or otherwise, for any child node q', if C_q and $C_{q'}$ do not satisfy the structural constraints between q and q', then C_q and the top element of $S_{p'}$ must satisfy the structural constraints between q and q'. For example, consider D_3 in Figure 6(a) and Q_9 in Figure 4(b), and suppose C_A, C_B and C_C point to a_2, b_4 and c_2, respectively. From Definition 4 we know that B has a PSE, and the top element a_1 in S_A satisfies the structural constraint of $A \Rightarrow B$, i.e. $a_1 \Rightarrow b_4$. However, we still cannot say that b_4 is a useful element. As b_4 and c_2 do not satisfy the structural constraint of $B \Rightarrow C$, B has not a Solution Extension. Further, b_4 does not satisfy the second condition "(2)" of "2", i.e. there is no element (in S_C) that can satisfy the structural constraint of $B \Rightarrow C$ with b_4. Therefore, b_4 is a useless element and can be safely discarded.

6 PSTP matching

Similarly to the TwigStack algorithm, in our method, each query node q in the given PSTP is associated with a stack S_q, a cursor C_q and a data stream T_q. S_q, C_q and T_q have the same meaning as that of TwigStack, and some functions used in our method are the same as that described in section 4.1.

6.1 Algorithm: pTwigStack

As shown in Algorithm 1, in the first phase (lines 1–8), as long as not all elements in element streams of leaf nodes are processed, getNext(root) is called repeatedly in line 2 to get a query node q with a PSE. If C_q is useful (determined by isUsefulEle(q) in line 3), it will be pushed into S_q in line 4, ModifyPointer(q) is used to modify related pointers in line 5. In line 6, elements that are not processed and have a smaller numerical id value than pre(C_q) are processed by calling ProcessOtherEle(q), i.e. all elements appear before C_q are processed all together with C_q if their corresponding query nodes are descendant of q in the given PSTP. In line 7, C_q is moved to the next element in T_q. When an element is popped from stack, all its related path solutions are produced by calling cleanStack(). In line 8, all remaining path solutions will be produced by calling outputPaths(). In the second phase, these path solutions are merge-joined to compute the final answers by calling MergeAllPathSolution() in line 9. Noting that path solutions should be outputted in root-to-leaf order so that they can be easily merged together to form the final answers, so we need to block some path solutions during output, just as showSolutionsWithBlocking[1] does.

```
Algorithm 1: pTwigStack(root)
1: while ¬end(root) do
2:    $q$ = getNext(root)
3:    if isUsefulEle($q$) then
4:       Push($C_q, S_q$, NULL)
5:       ModifyPointer($q$)
6:       ProcessOtherEle($q$)
7:    Advance($C_q$)
8: outputPaths()
9: MergeAllPathSolution()
Function: end($q$)
1: return $\forall q_i \in$ subtreeNodes($q$): isLeaf($q_i$)$\wedge$end($C_{q_i}$)
Procedure: ModifiyPointer($q$)
1: for $p \in$ relatedNodes($q$) do
2:    if $p//q \vee p \Rightarrow q$ then top($S_q$).$ptr$=top($S_p$)
3:    if $q \Rightarrow p$ then top($S_p$).$ptr$=top($S_q$)
Function: isUsefulEle($q$)
1: bUseful=FALSE; bFlag=TRUE
2: if isRoot($q$)$\wedge$hasSE($q$) then bUseful=TRUE
3: if ¬isRoot($q$)$\wedge$isHold(top($S_{\text{parent}(q)}$), $C_q$, $\langle$parent($q$), $q\rangle$)
4:    if hasSE($q$) then bUseful=TRUE
5:    else for each $q' \in$ children($q$)
6:       if ¬isHold($C_q, C_{q'}, \langle q, q'\rangle$)$\wedge$
         ¬isHold($C_q$,top($S_{q'}$), $\langle q, q'\rangle$) then bFlag = FALSE
7:       if bFlag=TRUE then bUseful=TRUE
8: return bUseful
Procedure: Push($C_q, S_q, ptr$)
1: push the pair ($C_q, ptr$) onto stack $S_q$
Procedure: cleanStack($S_p, C_q$)
1: Push all useful elements that are descendant of each
   popped element of $S_p$, which does not satisfy the
   structural relationship of $\langle p, q\rangle$ with $C_q$, then
   output related path solutions
Procedure: ProcessOtherEle($q$)
1: for $p \in$ children($q$) do
2:    while $q \Rightarrow p \wedge$pre($C_q$) $<$ $pre$($C_p$) do
3:       cleanStack($S_q, C_p$)
4:       if isUsefulEle($p$)=TRUE then
5:          Push($C_p, S_p, C_q$)
6:          ModifyPointer($p$)
7:          ProcessOtherEle($p$)
8:       Advance($C_p$)
```

getNext, as shown in Algorithm 2, is the core function called in pTwigStack, in which we need to consider A-D and the Samepath relationship simultaneously. getNext is used here to get a query node with a PSE, from which we can get an element that may participate in final answers. If q is a leaf node, it will be returned directly in line 1. If not, in lines 2–5, for each child p of q, if p' (returned by getNext(p)) is not equal to p, p' is returned in line 4; otherwise, if p' equals p and p has not a Solution Extension, p is directly returned in line 5. If all children of q have Solution Extension, we need to determine whether q has a PSE. In lines 6–7, we find $n_{\min}$ and $n_{\max}$ which have the minimal and maximal numerical id value from all children that have A-D but not the Samepath relationship with q. In lines 8–9, C_q is forwarded until C_q and $C_{n\max}$ are on the same path or $C_{n\max}$ is before C_q. If pre(C_q) $>$ pre($C_{n\min}$), $n_{\min}$ is returned in line 10. In lines 11–12, for each child of q that has the Samepath relationship with q, if C_q cannot cover or be covered by C_p and C_p appears before C_q, p is returned. Finally, if all children of q satisfy the structural constraint with q (hasSE(q)=TRUE) or C_q appears before C_p, i.e. $\neg(C_q \Rightarrow C_p) \wedge$pre($C_q$) $<$pre(C_p) (hasSE(q)=FALSE), q is returned in line 13.

```
Algorithm 2: getNext($q$)
1: if isLeaf($q$)=TRUE then return $q$
2: for $p \in$ children($q$) do
3:    $p'$ = getNext($p$)
4:    if $p' \neq p$ then return $p'$
5:    if ¬hasSE($p$) then return $p$
6: $n_{\min}$ =minarg$_p${pre($C_p$)|$q//p$}
7: $n_{\max}$ =maxarg$_p${pre($C_p$)|$q//p$}
8: while pre($C_q$) $<$pre($C_{n\max}$)$\wedge\neg$ ($C_q \Rightarrow C_{n\max}$) do
9:    Advance($C_q$)
10: if pre($C_q$) $>$pre($C_{n\min}$) then return $n_{\min}$
11: for $p \in$ children($q$) do
12:    if $q \Rightarrow p \wedge \neg(C_q \Rightarrow C_p)\wedge$pre($C_q$) $>$pre($C_p$) then
          return $p$
13: return $q$
```

Example 3. As shown in Figure 8, (a) is the given XML document D, (b) is a PSTP Q and (c)–(f) are four TPs of Q. Initially, C_A, C_B and C_C point to a_1, b_1 and c_1, respectively. The first call of getNext(A) returns C with a PSE since A and C have A-D relationship and pre(c_1) $<$pre(a_1) $\wedge$ $\neg(a_1//c_1)$. Because c_1 is useless, it is skipped directly and C_C is moved to c_2. The second call of getNext(A) returns A with a PSE, then a_1 is pushed into S_A and b_1 is also pushed into S_B since pre(b_1) $<$pre(a_1) and $B \in$children(A) and they are all useful elements. The statuses of S_A, S_B and S_C are shown in Figure 8(g). Though B is a leaf node, no path solution will be produced after b_1 is pushed into S_B since b_1 is ancestor of a_1. After that, C_A and C_B point to a_2 and b_2, respectively. Thus C_A, C_B and C_C point to a_2, b_2 and c_2. In the third call of getNext(A), a_2 is skipped directly and C_A is moved forwardly to a_3 since A and C have A-D relationship and pre(a_2) <pre(c_2) $\wedge \neg(a_2//c_2)$. Then C is returned with a PSE. Since c_2 is useful, it is pushed into stack S_C. The statuses of S_A, S_B and S_C are shown in Figure 8(h). The path solution (a_1, c_2) is produced since C is a leaf node and c_2 is a descendant of a_1. After that, c_2 is popped from S_C. The next call of getNext(A) returns B with a PSE, and b_2 will be pushed into S_B. The statuses of S_A, S_B and S_C are shown in Figure 8(i). The path solution (a_1, b_2) is produced and b_2 is popped from S_B. After all elements are processed, all related path solutions are produced at the end of the first phase. They are $(a_1, b_1), (a_1, b_2)$ and (a_1, c_2), respectively. In the second phase, all path solutions are merge-joined to get the final results, i.e. (a_1, b_1, c_2) and (a_1, b_2, c_2).

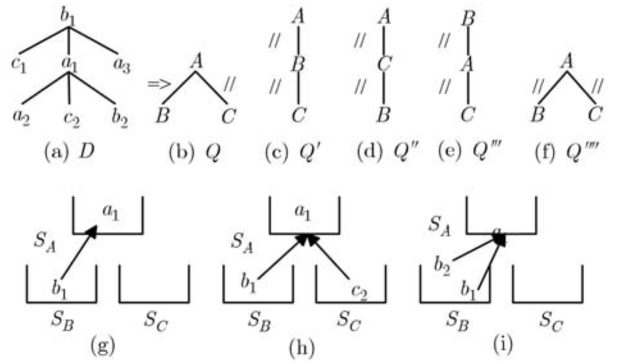


Figure 8 Document D, PSTP Q and its TPs ((a)–(f)), and running examples ((g)–(i)).

When P-C or PC-samepath edges appear in the given PSTP, we just need to take the level information of each element into account, the detailed description is omitted from Algorithm 1 for simplicity.

6.2 Analysis of pTwigStack

We first show the correctness of pTwigSatck and then analyze the complexity of pTwigStack.

Lemma 1. Let Q be a PSTP, and q be a query node of Q. If q=getNext(root), q has a PSE.

Lemma 2. Any useful element C_q will be pushed into stack S_q.

Obviously, Lemma 1 shows that from the query node q returned by getNext(root) we can get an element C_q that may be useful. Lemma 2 means that all useful elements are pushed into stacks. Let Q=A op B, where op can be A-D ("//"), P-C ("/"), PC-samepath("$\rightarrow$") or AD-samepath("$\Rightarrow$") relationship. Because P-C and PC-samepath relationships can be easily processed based on A-D and AD-samepath relationships, we just show the correctness about A-D and AD-samepath relationships. If op="//", $Q = A//B$ and the subsequent operation is the same as that in TwigStack. If op="$\Rightarrow$", $Q = A \Rightarrow B$. As shown in Figure 9, we need to consider four cases: (a) $C_A//C_B$, which is consistent with $A \Rightarrow B$ and A is returned first, the element processing order is C_A, C_B. (b) $C_B//C_A$, which is consistent with $A \Rightarrow B$ and A is returned first. Further, element C_B is processed simultaneously after A is returned, and the element processing order is C_A, C_B. This case denotes that all elements appearing before C_A are processed without another call of getNext(A). (c) pre(C_B) <pre(C_A) $\wedge \neg(C_A \Rightarrow C_B)$. B is returned first by getNext(A) since C_B appears before C_A. The element processing order is C_B, C_A. (d) pre(C_B) >pre(C_A) $\wedge \neg(C_A \Rightarrow C_B)$. In this case, A is returned first by getNext(A) since C_A appears before C_B, and A has a PSE, and the element processing order is C_A, C_B. In each case, if the processed element is useful, it will be pushed into S_q, otherwise, it will be discarded directly. Thus we have the following theorem.

Theorem 1. Let Q be a PSTP. If each edge in Q represents an A-D or AD-samepath relationship, then algorithm pTwigStack guarantees that only useful elements can be pushed into stack and each intermediate path solution can participate in final answers.

Theorem 1 means that each intermediate path solution produced by pTwigStack is useful when considering only A-D and AD-samepath relationship. The proof is simple. From Algorithm 1 we know that if an element is useless (line 3 in Algorithm 1), the cursor pointing to it will be forwarded to the next element (line 7 in Algorithm 1), otherwise, it will be pushed into stack according to Lemma 2. Further, if an element is useful, it must

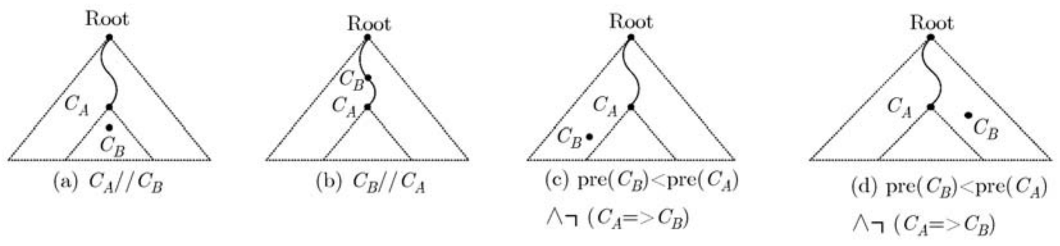

Root
C_A
C_B
(a) $C_A//C_B$
Root
C_B
C_A
(b) $C_B//C_A$
Root
C_A
C_B
(c) pre(C_B)<pre(C_A) $\wedge\neg$ (C_A=>C_B)
Root
C_A
C_B
(d) pre(C_B)<pre(C_A) $\wedge\neg$ (C_A=>C_B)

Figure 9 Cases for pTwigStack.

satisfy the structural constraint of the given query with other elements in the running stacks; thus each intermediate path solution consisting of only useful elements will definitely participate in final answers, i.e. it is useful.

Since all useful elements are pushed into stacks according to Lemma 2, in the procedure cleanStack, path solutions are produced when elements are popped from stacks. Finally, in the second phase of pTwigStack, all path solutions are merge-joined to compute the final answers. So we have the following theorem.

Theorem 2. Given a PSTP Q and an XML database D, algorithm pTwigStack correctly returns all answers for Q on D.

While the correctness holds for any given PSTP Q, the I/O optimality holds only for the case where no P-C and PC-samepath edges exist in Q as only useful elements are pushed into stacks. Therefore, we have the following result.

Theorem 3. Consider an XML database D and a PSTP Q that has n nodes and just A-D and AD-samepath edges. Algorithm pTwigStack has worst case I/O and CPU time complexities linear in the sum of sizes of the n input lists and the output list. Further, the worst case space complexity of Algorithm pTwigStack is the minimum of (i) the sum of sizes of the n input lists, and (ii) n times the maximum length of a root-to-leaf path in D.

Theorem 3 holds only for PSTP with A-D and AD-samepath edges, in the case where the PSTP contains P-C or PC-samepath edges. Algorithm pTwigStack is no longer guaranteed to be I/O and CPU time optimal. In particular, the algorithm might produce a solution for one root-to-leaf path that does not match any solution in another root-to-leaf path.

6.3 Optimization

If schema is not considered for query processing, obviously, pTwigStack definitely outperform TwigStack since a PSTP may correspond to multiple TPs. In this paper, we represent the underneath schema S as a directed graph, where each node corresponds to a tag name, and each edge from node A to node B means that in the document complying with S, elements with tag B can be children of elements with tag A. We say there exists a cycle between A and B in S if there exists at least a path from A to B and vice versa, which means that elements with tag A can be ancestor or descendant of elements with tag B. However, if no cycle exists between two query nodes in S, the Samepath relationship between the two nodes in a PSTP can be replaced by just one P-C or A-D relationship. If we can make use of such structural information, then query performance can be improved significantly.

Let a PSTP $Q = \{V, E, T\}$, where V is the set of nodes in Q, $E \subseteq V \times V$ is the set of edges in Q, and $T : E \to R$ is a type function that maps each edge to a value in the relationship set R={'/','//','$\rightarrow$','$\Rightarrow$'}. We propose Algorithm 3 to optimize a PSTP using the schema S.

In Algorithm 3, hasCycle(A, B, S) is used to check whether there exists a cycle between A and B in schema S. For each edge $\langle A, B\rangle \in Q.E$, if A and B are connected by a PC-samepath edge and there is no cycle between A and B (line 3), the relationship between A and B can be replaced by P-C edge (lines 4–5). Similarly, if A and B are connected by an AD-samepath edge and there is no cycle between A and B (line 6), the relationship between A and B can be replaced by A-D edge (lines 7–8). In line 9, the optimized PSTP Q is returned. Compared with the PSTP input into

Algorithm 3, the number of derived TP is reduced significantly, as shown in Figure 10. For simplicity, we call Algorithm 3 the 1st optimization technique.

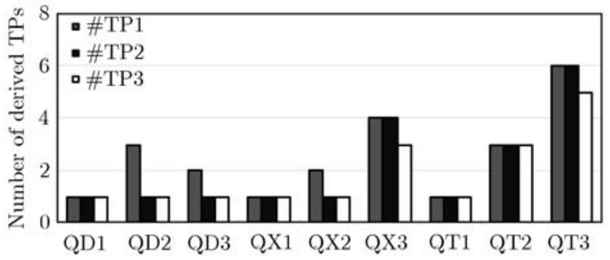


Figure 10 Number of derived TPs for each query. #TP1, Number of derived TPs without optimization; #TP2, number of derived TPs after using the 1st optimization technique; #TP3, number of remained TPs after using the 2nd optimization technique.

```
Algorithm 3: Rewrite(Q, S) /*Q is a PSTP, S is the schema graph*/
1: for $\forall A, B \in Q.V$ do
2:   if $\langle A, B\rangle \in Q.E$ then
3:     if $T(\langle A, B\rangle) =$ '$\rightarrow$' $\wedge\neg$hasCycle$(A, B, S)$ then
4:       $T(\langle A, B\rangle) \leftarrow$'/'
5:       '$A \rightarrow B$' is replaced by '$A/B$' or '$B/A$'
6:     if $T(\langle A, B\rangle)$='$\Rightarrow$' $\wedge\neg$hasCycle$(A, B, S)$ then
7:       $T(\langle A, B\rangle) \leftarrow$ '//'
8:       '$A \Rightarrow B$' is replaced by '$A//B$' or '$B//A$'
9: return $Q$
```

Example 4. Assume that the DTD schema of the given XML document consists of two rules, namely $\langle$!ELEMENT A $(A, B, C)\rangle$ and $\langle$!ELEMENT B $(A*, C)\rangle$. Obviously, there exists a cycle between A and B, not B and C. If the given PSTP is Q_9 in Figure 4 (b), by Algorithm 3, it can be rewritten as $Q_{9'}$: $A{\Rightarrow}B//C$, through which we can get four TPs, they are $A//B//C, B//A//C, B//C[.//A]$ and $B[.//A]//C$. Note that after optimization using Algorithm 3, the cost of evaluating Q_9 is greatly reduced since many derived TPs of Q_9 are no longer existent.

Further, schema information can also be used to check whether a PSTP or derived TP is satisfied, i.e. the answer set is not empty. Thus we can further reduce query processing cost by checking whether the given query is consistent with the schema information (i.e. Definition 6). We implemented such an operation in Algorithm 4, which we call the 2nd optimization technique.

Definition 6. Letting a PSTP $Q = \{V, E, T\}$, we say that Q is consistent with schema S if for each edge $\langle A, B\rangle \in Q.E$, we can find two nodes A', B' that satisfy Lable(A) =Lable(A') and Lable(B) =Lable(B') in S, such that the relationship of A' and B' satisfies the structural constraint of $T(\langle A, B\rangle)$ according to S, which corresponds to four cases: 1) if $T(\langle A, B\rangle)$ ='/', there exists an edge from A' to B'; 2) if $T(\langle A, B\rangle)$='//', there exists a path from A' to B'; 3) if $T(\langle A, B\rangle)$='$\rightarrow$', there exists an edge from A' to B' or vice versa; and 4) if $T(\langle A, B\rangle)$ ='$\Rightarrow$', there exists a path from A' to B' or vice versa.

```
Algorithm 4: isConsistent(Q, S)
1: for $\forall A, B \in Q.V$ do
2:   if $\langle A, B\rangle \in Q.E$ then
3:     find two nodes $A'$, $B'$ corresponding to $A$ and
       $B$ in $S$
4:     if $T(\langle A, B\rangle)$ ='/' $\wedge\neg$hasEdge$(A', B', S)$ then
5:       return FALSE
6:     if $T(\langle A, B\rangle)$ ='//' $\wedge\neg$hasEdge$(A', B', S)$ then
7:       return FALSE
8:     if $T(\langle A, B\rangle)$ ='$\rightarrow$' $\wedge$
         $\neg$(hasEdge$(A', B', S)\vee$hasEdge$(B', A', S)$) then
9:       return FALSE
10:    if $T(\langle A, B\rangle)$ ='$\Rightarrow$' $\wedge$
         $\neg$(hasPath$(A', B', S)\vee$hasPath$(B', A', S)$) then
11:      return FALSE
12: return TRUE
```

Algorithm 4 is used to determine whether the given query is consistent with the underneath schema, where hasEdge(A, B, S) is used to check whether there is an edge from A to B in S. And hasPath(A, B, S) is used to check whether there is a path from A to B. For each edge $\langle A, B\rangle$ in $Q.E$, we find two nodes A', B' that correspond to A and B in S. In lines 4–11, we check whether the relationship between A' and B' in S is satisfied with the structural constraint between A and B in Q, which corresponds to the four cases in Definition 6. If any one of the four cases does not hold, Algorithm 4 is terminated with FALSE returned, meaning that Q is not consistent with S; otherwise, TRUE is returned in line 12, which means that Q is consistent with S.

Assume that the DTD schema is the same to that of Example 4. Consider the four TPs derived from $Q_{9'}$, i.e. $A//B//C, B//A//C, B//C[.//A]$

and $B[.//A]//C$. Since no elements can appear below elements with tag C according to the DTD schema, we can safely discard $B//C[.//A]$ as it is not consistent with the DTD schema. Thus only three TPs are left for further processing.

7 Experimental evaluation

7.1 Experimental setup

Our experiments were implemented on a PC with Pentium4 2.8 GHz CPU, 512 MB memory, 160 GB IDE disk, and Windows XP professional as the operation system.

We used TwigStack as the basis when implementing the naive method proposed in ref. [12], which we call nTwigStack (nTS). In addition to nTS and pTwigStack (pTS), we also implemented five other algorithms using three optimization techniques including B+ tree index, the 1st and 2nd optimization techniques (i.e. Algorithm 3 and Algorithm 4). B+ tree is used to index element labels so that we can skip useless element labels in label stream like TSGeneric+[2]. The 1st optimization is used to rewrite the given PSTP and the 2nd optimization is used to check whether a given query is consistent with the underneath schema. All these algorithms are listed in Table 1, where nTS-O is an improved version of nTS using the 1st optimization technique, nTS-OB is the combination of nTS, the 1st optimization technique and B+ tree index, and nTS-OBO is the combination of nTS, B+ tree index, the 1st and the 2nd optimization techniques. Similarly, pTS-O is an improved version of pTS using the 1st optimization technique, pTS-OB is the combination of pTS, the 1st optimization technique and B+ tree index. All algorithms were implemented using Visual C++ 6.0.

7.2 Datasets and queries

We used XMark (http://monetdb.cwi.nl/xml), DBLP (http://www.cs.washington.edu/research/xmldatasets/www/repository.html) and TreeBank for our experiments. The main characteristics of the three datasets can be found in Table 2. Although PSTP can be used to extract useful information from multiple XML documents with structural heterogeneity, we use one data set with recursive structure to simulate multiple data sources with different structures.

In our experiment, each element is labeled with a triple (start, end, level) and then stored into two separate files, one is sequential file, and the other is random file (disk-based B+ tree index). All labels corresponding to same tag are stored together in a label stream in an ascending order according to start value of each label. Sequential file is used in nTS, nTS-O, pTS and pTS-O algorithms without B+ tree index, and random file is used in nTS-OB, nTS-OBO and pTS-OB algorithms with B+ tree index. Each query node with a distinct tag name corresponds to a separate label stream.

The queries used in our experiment are listed in Table 3. Since the 2nd optimization technique is just used for the derived TPs in our experiment, we show in Table 3 only the changes caused by the 1st optimization technique. The benefits of the 2nd optimization is shown in Figure 10. All these queries can be classified into three categories: 1) PSTPs without the Samepath edges ("$\Rightarrow$" and "$\rightarrow$"), e.g. QD1, QX1 and QT1, which we call 1st group PSTPs. 2) PSTPs can be transformed to TPs based on DTD schema using the 1st optimization, e.g. QD2, QD3 and QX2, which we call the 2nd group. 3) PSTPs cannot be transformed to TPs, which means that some Samepath edges cannot be replaced by P-C or A-D edges since there exist cycles in the schema between the nodes connected by these Samepath edges, e.g. QX3, QT2 and QT3, which we call the 3rd group.

Table 1 Algorithms and optimization techniques used in our experiment

	nTS	nTS-O	nTS-OB	nTS-OBO	pTS	pTS-O	pTS-OB
The 1st optimization		√	√	√		√	√
B^+ tree index			√	√			√
The 2nd optimization				√			

Table 2 Statistics of XML data sets

Dataset	Size (M)	Nodes (Million)	Max depth	Average depth
DBLP	127	3.3	6	2.9
XMark	113	1.7	12	5.5
TreeBank	82	2.4	36	7.8

Table 3 Queries used in our experiment[a)]

	Dataset	Status	Queries	Group
QD1	DBLP	BO	//book/author	1
		AO	-no change-	
QD2	DBLP	BO	//www[.⇒editor]/url	2
		AO	//www[.//editor]/url	
QD3	DBLP	BO	//dblp⇒article/year	2
		AO	//dblp//article/year	
QX1	XMark	BO	//site/people/person/name	1
		AO	-no change-	
QX2	XMark	BO	/site⇒closed_auctions/emph	2
		AO	/site//closed_auctions/emph	
QX3	XMark	BO	//listitem[.//bold]/text[.//emph]⇒keyword	3
		AO	-no change-	
QT1	TreeBank	BO	//S/VP/VBD	1
		AO	-no change-	
QT2	TreeBank	BO	//S[.⇒JJ]/NP	3
		AO	-no change-	
QT3	TreeBank	BO	//S⇒VP/PP[NP/VBN]/IN	3
		AO	-no change-	

a) BO, Before the 1st optimization; AO, after the 1st optimization.

7.3 Performance comparison and analysis

We consider the following performance metrics to compare the performance of different algorithms: 1) number of derived TPs, which reflects the effect of using the 1st and 2nd optimization techniques; 2) number of scanned elements; 3) running time; and 4) scalability.

Consider the first metric, i.e. number of derived TPs, as shown in Figure 10, for the 1st group of PSTPs, optimization equals no optimization since no Samepath edge is contained in the three PSTPs. For the 2nd group of PSTPs, the 1st optimization will cause the Samepath edge replaced by A-D edge, the 2nd optimization technique will not bring us any benefits since each PSTP is consistent with the schema. For PSTPs in the 3rd group, the 1st optimization technique does not work in such a case since the three PSTPs are consistent with the schema and the Samepath edge in each PSTP cannot be replaced by A-D or P-C edge, but the 2nd optimization technique does work since some derived TPs of QX3 and QT3, except QT2, are not consistent with the schema; thus they can be safely discarded without further processing.

Consider the 1st group of PSTPs, as shown in Figure 11(a), the number of scanned elements of nTS, nTS-O, pTS and pTS-O are same as each other, because neither of the two optimization techniques works for the 1st group of PSTPs, and for a given query, each algorithm uses the same set of label streams (sequential files); thus they will read the same amount of element labels. Figure 11(b) shows that our method, pTS and pTS-O, need to afford additional CPU cost since our method has more judging operations. This conclusion also holds for the comparison of nTS-OB, nTS-OBO and pTS-OB. However, as shown in the following, this performance degradation can be safely ignored when compared with the significant performance improvement achieved from the 3rd group of PSTPs.

For the 2nd group of PSTPs, as shown in Figure 12, the number of scanned elements of nTS is more than that of other algorithms since nTS will process each derived TP of the given PSTP without any optimization; as a result, running time of nTS is also very large compared with other algorithms. For the remainder algorithms, we can see from Figure 12 that algorithms with same basic configuration (label streams and optimization techniques) have similar performance because after the 1st optimization, each one of the 2nd group of PSTPs will be transformed to a TP, and the 2nd optimization doesn't work for each PSTP after transformation. Note using B+ tree index to skip useless elements will greatly improve query performance for QD2 rather than QD3. For QD2, only few elements are useful, so the running time of nTS-OB, nTS-OBO and pTS-OB is less than that of nTS-O, pTS and pTS-O. For QD3, although the number of scanned elements is reduced after using B+ tree index, repeatedly searching from root to leaf node makes the I/O cost of nTS-OB, nTS-OBO and pTS-OB larger than that of nTS-O, pTS and pTS-O, which will further result in poor query performance.

For the 3rd group of PSTPs, as shown in Figure 13, for any metrics in this figure, our methods, i.e. pTS, pTS-O and pTS-OB, outperform nTS, nTS-O, nTS-OB and nTS-OBO significantly, because the 1st optimization technique is useless for the 3rd group of PSTPs and each PSTP corresponds

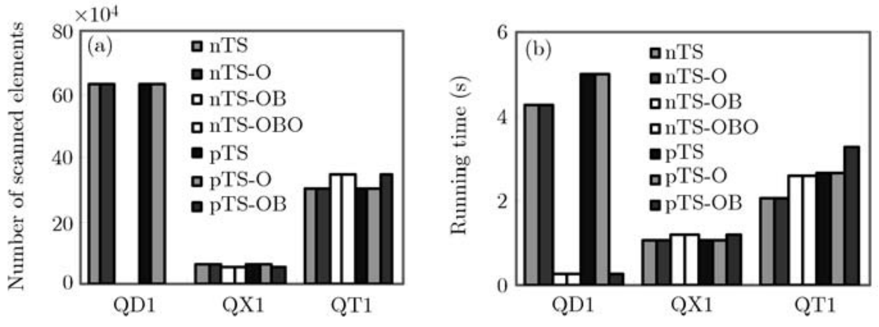


Figure 11 Performance comparison against the 1st group of PSTPs. (a) Number of scanned elements; (b) runnin time.

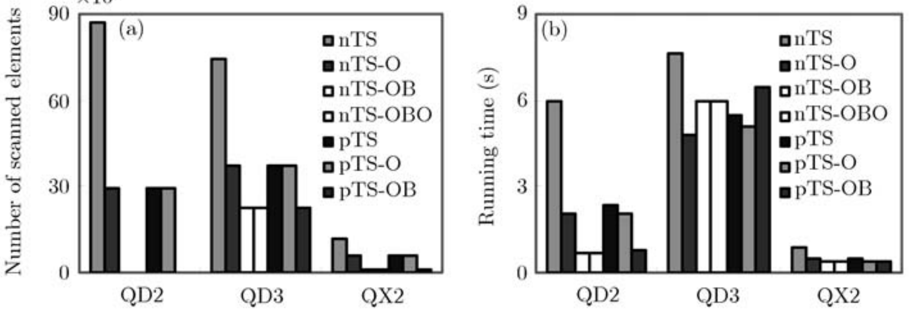


Figure 12 Performance comparison against the 2nd group of PSTPs. (a) Number of scanned elements; (b) running time.

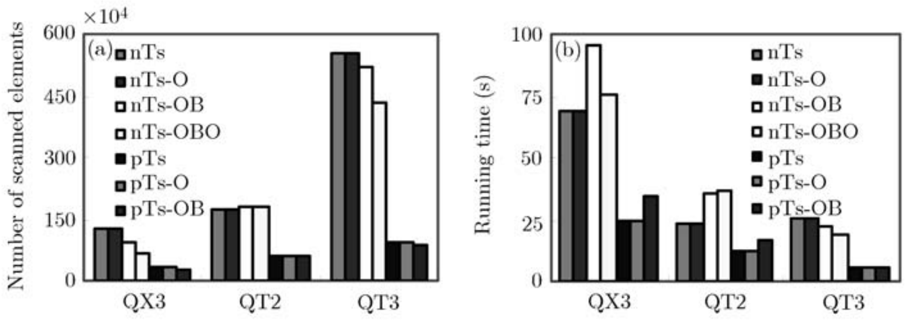


Figure 13 Performance comparison against the 3rd group of PSTPs. (a) Number of scanned elements; (b) running time.

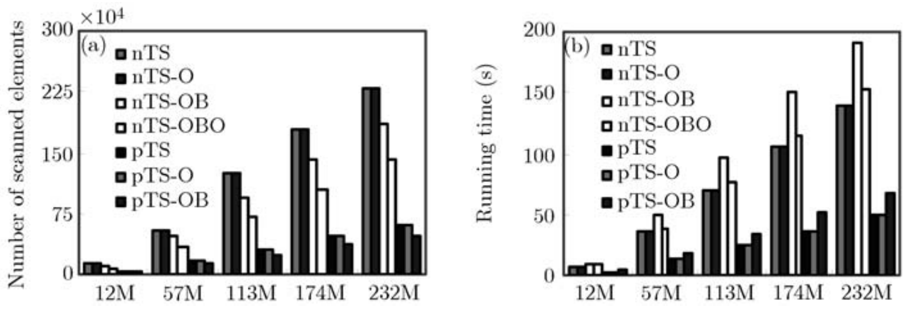


Figure 14 Performance comparison of seven algorithms over XMark of different sizes using QX3. (a) Number of scanned elements; (b) running time.

to multiple TPs (Figure 10), among which only limited TPs can be safely discarded using the 2nd optimization technique, thus all remainder TPs after the 2nd optimization will be processed one by one using nTS, nTS-O, nTS-OB or nTS-OBO algorithm, as a result, large amount of elements need to be scanned multiple times, which will result in high CPU cost. Although the 2nd optimization technique is useful in such a case, the effect is limited for the 3rd group of PSTPs. Obviously, naive method, nTS, and its improved algorithms, nTS-O, nTS-OB and nTS-OBO, cannot work efficiently for the 3rd group of PSTPs.

Further, we present the performance results about scalability of QX3 in Figure 14, from which we know that our methods can work more efficiently than naive methods when processing PSTP over XML document with different size, because our method processes each element only once, while the performance of naive methods is determined by the number of derived TPs.

From the above experimental results and our analysis we know that when processing PSTPs with Samepath edges, especially the effect of optimization is not remarkable, e.g. QX3, QT2 and QT3, our methods, e.g. pTS, pTS-O and pTS-OB, can work much more efficiently than naive methods, i.e. nTS, and its optimization, i.e. nTS-O, nTS-OB and nTS-OBO. The reason lies in two aspects: 1) our methods guarantee that each element is scanned only once; and 2) our methods guarantee that no useless intermediate paths will be produced when considering only A-D and AD-samepath relationships (Theorem 1). Even if no Samepath edge appears in the query expression, e.g. QD1, QX1 and QT1, or otherwise, the Samepath edges can be replaced by P-C or A-D edges after optimization, e.g. QD2, QD3 and QX2, our method can achieve similar performance to that of the existing methods.

8 Related work

TPs can be used to match data fragments in an XML document. MPMGJN[13] was proposed for efficient structural join, Stack-Tree-Desc/Anc[15] improves the query performance of MPMGJN by using stack-based binary structural join algorithm.

Wu et al.[16] studied the problem of binary join order selection for complex queries based on a cost model. All these structure join methods suffer from the large number of intermediate results. To process a TP holistically, many methods[1−4] were proposed to avoid producing large size of useless intermediate results. Among them, TwigStack[1] was proposed to process a TP in a holistic way. When considering only A-D edges, TwigStack guarantees that the CPU time and I/O complexity is independent of the size of partial matches to any root-to-leaf path. Other methods[2−4] made improvements against TwigStack from different aspects. TSGeneric+[2] focused on holistic twig joins on all/partly indexed XML documents to skip some useless elements. Chen et al.[3] proposed iTwigJoin that exploits different data partition strategies to further boost the holism. TJEssential[4] uses a hybrid strategy to avoid redundant operations compared with the methods of refs. [1–3]. All these methods can only be used to a single TP, for a PSTP, however, they cannot work efficiently since a PSTP may correspond to several TPs.

Keyword based methods[8,9] can provide us with the most flexibility. MLCAS was introduced in ref. [9] to reduce meaningless results. XSEarch[8] returns semantically related document fragments that satisfy the user's query. However, we can not specify that several nodes are on the same path without specifying the concrete precedence relationship. Query relaxation based methods[10,11] will produce a large number of relaxed query expressions, thus resulting in too many approximate answers.

In the area of integrating tree-structured data, the Xyleme system[17] exploits XML views to cope with the problem. The Agora system[18] translates query expressions to SQL queries on each local data source. In ref. [19], queries are processed using mapping rules between a global schema and many local data sources, and then evaluated in each data source.

Although the notion of PSTP has been proposed in ref. [12] to provide the users with a more flexible way to express semantic constraints, no existing work has focused on holistic query evaluation for PSTPs. In ref. [20], the authors proposed a method for evaluation of partial path queries, not a query expression with branch node, thus we do not compare them with this method since it is based on different query syntaxes.

9 Conclusions

In this paper, we firstly extended XPath language with the new axis, Samepath axis, which allows users to express partial semantic constraints of being on the same path in a concise but effective way; then we proposed a new holistic query processing method, pTwigStack, for semantically querying tree-structured data sources using PSTPs. Our experimental results show that our method can work more efficiently than the existing methods when processing PSTPs. As our method cannot process a PSTP that is not specified with a root node, we will focus on this problem in the near future.

1 Bruno N, Koudas N, Srivastava D. Holistic twig joins: optimal XML pattern matching. In: Michael J F, Bongki M, Anastassia A, eds. Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data. Madison: ACM, 2002. 310–321

2 Jiang H, Wang W, Lu H, et al. Holistic twig joins on indexed XML documents. In: Freytag J C, Lockemann P C, Abiteboul S, et al., eds. Proceedings of 29th International Conference on Very Large Data Bases. Berlin: Morgan Kaufmann, 2003. 273–284

3 Chen T, Lu J, Ling T W. On boosting holism in XML twig pattern matching using structural indexing techniques. In: Fatma Ö, ed. Proceedings of the ACM SIGMOD International Conference on Management of Data. Baltimore: ACM, 2005. 455–466

4 Li G, Feng J, Zhang Y, et al. Efficient holistic twig joins in Leaf-to-Root combining with Root-to-Leaf way. In: Ramamohanarao K, Krishna P R, Mohania M K, et al., eds. Proceedings of 12th International Conference on Database Systems for Advanced Applications. Bangkok: Springer, 2007. 834–849

5 Olteanu D. Forward node-selecting queries over trees. ACM Trans Database Syst, 2007, 32(1): 75–111

6 Olteanu D, Meuss H, Furche T, et al. XPath: looking forward. In: Chaudhri A B, Unland R, Djeraba C, et al., eds. EDBT 2002 Workshops XMLDM, MDDE, and YRWS. Prague: Springer, 2002. 109–127

7 Gottlob G, Koch C, Pichler R. The complexity of XPath query evaluation. In: Alin D, ed. Proceedings of the Twenty-third

ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. San Diego: ACM, 2003. 179–190

8 Cohen S, Mamou J, Kanza Y, et al. XSEarch: a semantic search engine for XML. In: Freytag J C, Lockemann P C, Abiteboul S, et al., eds. Proceedings of 29th International Conference on Very Large Data Bases. Berlin: Morgan Kaufmann, 2003. 45–56

9 Li Y, Yu C, Jagadish H V. Schema-Free XQuery. In: Nascimento M A, Özsu M T, Kossmann D, et al., eds. Proceedings of the 30th International Conference on Very Large Data Bases. Toronto: Morgan Kaufmann, 2004. 72–83

10 Sihem A Y, Koudas N, Marian A, et al. Structure and content scoring for XML. In: Böhm K, Jensen C S, Haas L M, et al., eds. Proceedings of the 31st International Conference on Very Large Data Bases. Trondheim: ACM, 2005. 361–372

11 Sihem A Y, Cho S R, Srivastava D. Tree pattern relaxation. In: Jensen C S, Jeffery K G, Pokornÿ J, et al., eds. Proceedings of 8th International Conference on Extending Database Technology. Prague: Springer, 2002. 496–513

12 Theodoratos D, Souldatos S, Dalamagas T, et al. Heuristic containment check of partial tree-pattern queries in the presence of index graphs. In: Yu P S, Tsotras V J, Fox E A, et al., eds. Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management. Virginia: ACM, 2006. 445–454

13 Zhang C, Naughton J F, DeWitt D J, et al. On supporting containment queries in relational database management systems. In: Walid G A, ed. Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data. Barbara: ACM, 2001. 425–436

14 Tatarinov I, Viglas S, Beyer K S, et al. Storing and querying ordered XML using a relational database system. In: Franklin M J, Moon B, Ailamaki A, eds. Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data. Madison: ACM, 2002. 204–215

15 Shurug A K, Jagadish H V, Jignesh M P, et al. Structural joins: a primitive for efficient XML query pattern matching. In: Umeshwar D, ed. Proceedings of the 18th International Conference on Data Engineering. San Jose: IEEE Computer Society, 2002. 141–152

16 Wu Y, Jignesh M P, Jagadish H V. Structural join order selection for XML query optimization. In: Dayal U, Ramamritham K, Vijayaraman T M, eds. Proceedings of the 19th International Conference on Data Engineering. Bangalore: IEEE Computer Society, 2003. 443–454

17 Cluet S, Veltri P, Vodislav D. Views in a large scale XML repository. In: Apers P M G, Atzeni P, Ceri S, et al., eds. Proceedings of 27th International Conference on Very Large Data Bases. Roma: Morgan Kaufmann, 2001. 271–280

18 Manolescu I, Florescu D, Kossmann D. Answering XML queries on heterogeneous data sources. In: Apers P M G, Atzeni P, Ceri S, et al., eds. Proceedings of 27th International Conference on Very Large Data Bases. Roma: Morgan Kaufmann, 2001. 241–250

19 Christophides V, Cluet S, Siméon S. On wrapping query languages and efficient XML integration. In: Chen W, Naughton J F, Bernstein P A, eds. Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. Texas: ACM, 2000. 141–152

20 Souldatos S, Wu X, Theodoratos D, et al. Evaluation of partial path queries on XML data. In: Silva M J, Laender A H F, Baeza-Yates R A, et al., eds. Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management. Lisbon: ACM, 2007. 21–30