International Journal of Database Theory and Application
Vol.7, No.3 (2014), pp.179-190
http://dx.doi.org/10.14257/ijdta.2014.7.3.16

Recursive Twig Pattern Query

Husheng Liao, Hongyu Gao and Zhaoning Guan

Beijing University of Technology, Beijing, China
{liaohs, hy_gao}@bjut.edu.cn, gznmind@gmail.com

Abstract

XQuery is a language for querying XML data which is widely used on the Internet. In XQuery, user can define recursive functions for querying and processing XML data. XML twig pattern query is considered as core operation for querying XML data which has been studied intensively in recent years. More powerful recursive queries can be achieved via combining user-defined recursive function and twig pattern query. This paper proposes recursive twig pattern query (RTPQ) to extend traditional twig pattern query with recursion and an effective holistic twig matching algorithm for RTPQ. With the help of RTPQ, recursive function call can access its result instead of evaluating local twig pattern queries in each calling. Results of experiments show that this approach can improve query efficiency and eliminate redundant intermediate results.

Keywords: *XML, XQuery, Twig Pattern, Recursive Function*

1. Introduction

XML has been widely used on the data storage and exchange on Internet. XML twig pattern query [1] is considered as a core operation for querying XML which has been studied intensively in recent years. It models a query request over a number of structural joins (ancestor-descendant relationship and parental-child relationship) between XML nodes as a pattern tree. This kind of query can be performed by twig pattern matching against a tree-structured data like XML data. In XQuery language, which is recommended by W3C for querying XML data, the query usually consists of one or more nested FLWOR expressions containing some XPath expressions to locate XML nodes. The core operation within them is twig pattern query. For example, XPath expressions *//person/name* and *//person/gender* can be merged into a twig pattern, which has a root node labeled with *person* and two leaf nodes labeled with *name* and *gender* respectively.

```
<! DOCTYPE family [
<! ELEMENT family (person) >
<! ELEMENT person (name, child, gender) >
<! ELEMENT name (#PCDATA) >
<! ELEMENT child (person*) >
<! ELEMENT gender (#PCDATA) > ]>
```

(a)

```
<! DOCTYPE book [
<! ELEMENT book (title, author+, section+) >
<! ELEMENT title (#PCDATA) >
<! ELEMENT author (#PCDATA) >
<! ELEMENT section (title, (p | figure | section)* ) >
<! ATTLIST section id ID #IMPLIED>
<! ELEMENT p (#PCDATA) >
<! ELEMENT figure (title, image) >
<! ELEMENT image EMPTY>
<! ATTLIST image source CDATA #REQUIRED > ]>
```

(b)

```
<! DOCTYPE inventory [
<! ELEMENT inventory (category+) >
<! ELEMENT category (category*, production*) >
<! ATTLIST category
        id ID #REQUIRED
        name CDATA #REQUIRED>
<! ELEMENT production (id, name, quantity)>
<! ELEMENT id (#PCDATA) >
<! ELEMENT name (#PCDATA) >
<! ELEMENT quantity (#PCDATA) > ]>
```

(c)

Figure 1. DTDs of Recursive XML Documents

ISSN: 2005-4270 IJDTA

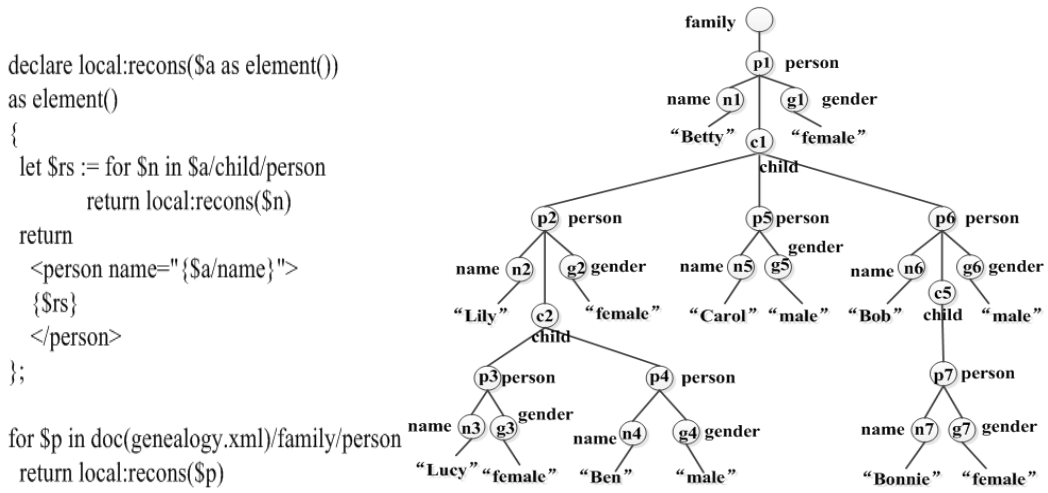

Figure 2. An Example of XQuery Recursive Query and an XML Document

In practice, XML can be used to represent data with recursive data structure. Figure 1 shows DTDs of three XML documents in which the recursively defined element are denoted by bold letters. For these documents, recursive query can be submitted by various query languages. XQuery can describe recursive processing of XML data in user-defined recursive function, and XSeq [2], which is a query language for XML streams, supports recursive query by some powerful extensions to XPath, such as Kleene closure.

Figure 2 shows an example of XQuery recursive query against the XML document defined by DTD shown in Figure 1(a). The query program uses a user-defined recursive function with element constructors to rebuild document by searching */child/person* nodes from the document repeatedly, and creating *person* nodes and using the content of each original *name* node as an attribute of the new *person* nodes. This searching can be also described as *person(/child/person)** in XSeq language.

However, efficiency of such query requests may be reduced by both recursive computation and complex data query. In recursive function calls in XQuery, if a function body includes some FLWOR or XPath expressions, evaluation of these expressions at each recursion call level may work on the result of the corresponding twig pattern matching. Similar queries are performed repeatedly and redundant intermediate results are generated at each recursion call level. Consequently, it increases the cost of data querying and causes declines in performance of the function call.

This paper studies on the processing of recursive query with twig pattern query. The main contribution is as follows:

a. Propose the concept of recursive twig pattern query denoted by *RTPQ* which extends the traditional twig pattern to describe recursive query request.

b. Propose a holistic twig matching algorithm and an enumeration algorithm for *RTPQ*. A number of twig patterns in recursive calls can be replaced by a single *RTPQ*, which avoids querying XML documents repeatedly and reduces the redundant intermediate data.

The remainder of this paper is organized as follows. Section 2 explains the motivation of the research. Section 3 defines recursive twig pattern. Section 4 describes RTPQ algorithm. Section 5 discusses the related work. Experimental results are presented in Section 6.

2. Motivation

Recursive structure often appears in XML documents. In Figure 2, the ancestor node named *Betty* is stored under the family node and its descendant recursively stored in the child node. For simplicity, *person* nodes are numbered as "*p1*" and so on. For XML query in XQuery, twig patterns can be identified from its FLOWR expression.

For example, the twig pattern shown in Figure 3(a) can be identified from the XQuery program in Figure 2, which will be applied on the function's parameter *$a*. Dotted edges in the tree pattern denotes the optionally relationships. The nodes in twig pattern is called query node. The results of twig query are bound to the variables on query nodes for subsequent processing. By the current holistic pattern matching algorithm [3], its query results can be gotten through a single matching against an XML document.

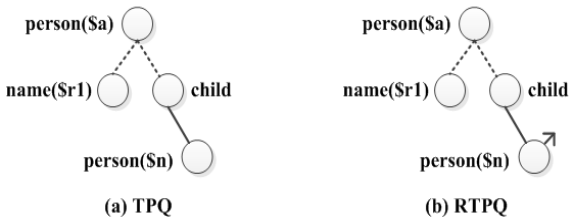

Figure 3. The Twig Pattern Extracted from Function Body

However, this twig pattern can only match data in a single function call. For example, when function *recons* is first called, the root node *person($a)* matches with *p1*, *person($n)* with matches *p2*, *p5* and *p6* from sub-trees of *p1*, and they are put into the next recursion call level one by one as its argument. The twig query needs to be executed against them and scan entire sub-tree rooted at these nodes passed by *$a*. Thus, some XML sub-tree may be traversed repeatedly. But the original intention of twig query is to scan an XML document only once.

In order to optimize the recursive query in XQuery functions, this paper extends XML twig pattern with the ability to represent recursive query. As shown in Figure 3(b), *RTPQ* can describe the relationship between actual parameter and formal parameter in recursive function calling. The argument of recursive function call corresponds to a recursive query node, which is denoted as a circle with a slanted arrow, and it will be considered as the root query node at the next recursion call level. Benefitted from *RTPQ*, only one single pattern-matching is needed to obtain all data for every recursive function callings. It merges multiple twig matching into one single *RTPQ* matching that improves efficiency of recursive query in XQuery. For simplicity, there are two assumptions in this paper: (1) twig query in recursive functions is applied on only one single XML document; (2) XPath expressions in the function body are static.

3. Recursive Twig Pattern

Definition 1. Recursive twig pattern Q is defined as a non-empty query tree *T (N, E, bind, lab, root)*, where *N* is a finite set of query nodes, *E* is a finite set of edges, $root \in N$ is the root query node, $R \subset N\text{-}\{root\}$ is a set of recursive query nodes, $M \subset N$ is a set of return nodes,

function *bind: M→Name* gives bind variables of return nodes, function *lab: N→String* gives node labels. Edges in *E* are defined as *e=(q1, q2, struct, bindtype)*, where *q1, q2*∈*N*, *struct*∈*{PC,AD}*, *bindtype*∈*{mandatory, optional}*, where *PC* and *AD* denotes ancestor-descendant and parental-child relationships respectively, *mandatory* and *optional* denotes the relationships are mandatory and optional respectively.

Definition 2. Matching *M* between a twig pattern and a tuple of XML nodes should satisfy following properties: (1) In accordance with depth-first order, types of XML nodes in the tuple correspond to labels of query nodes. (2) Structural relationship between any two XML nodes in the tuple meets structure constraints described by the edge between corresponding query nodes.

In the recursive twig pattern, all labels of the query nodes are determined by corresponding location steps of an XPath expression. Especially, label of the root query node is determined by the argument of function call in main query body. The recursive query node not only describes the argument of inner function call at current recursion call level but also describes the parameter of the callee at next recursion call level. The root query node describes parameter of the function only at first recursion call level. Therefore, tuples in matching results can be divided into two types.

Definition 3. A basic matching result of recursive twig pattern is a tuple *D* which consists of XML nodes. *D* meets the matching *M* with recursive twig pattern. The element of *D*, which corresponds to the root query node, must be the XML node bound by the parameter at the first run of recursive function.

Definition 4. A recursive result of recursive twig pattern is a tuple *B* which consists of XML nodes. *B* meets the matching *M* with recursive twig pattern (makes recursive query node replace the root query node). The element of *B*, which corresponds to the root query node, must also correspond to the recursive query node in another matching result.

The result of *RTPQ* is composed of one basic matching result and several recursive results. A basic result is what recursive function needs at first recursion call level. The root query node only matches with XML nodes passed by the argument. A recursive result provides data to recursive function at a deeper recursion call level.

The recursive twig pattern shown in Figure 3(b) consists of root query node *person($a)*, two ordinary query nodes labeled with *name($r1)* and *child*, and a recursive query node labeled with *person($n)*. All edges are PC edge, where solid lines for mandatory relationships and dotted lines for optionally relationships. The root query node matches with the argument of function *recons* and other query nodes correspond to XPath location steps in the function body.

4. Recursive Twig Matching Algorithm

A number of matching algorithms for ordinary twig pattern query have been issued[3-8]. [3] proposes two holistic twig matching algorithms: *PathStack* and *TwigStack*, using twig pattern to match XML document as a whole firstly. [4] proposes *Twig2Stack* which optimizes *TwigStack* and supports *GTP* which an extended twig pattern, but its complexity of stack structure is not easy to use. [5] comes up with *TwigList* using a kind of list as the data structure which represents intermediate results clearly and is more practical. Recursive twig matching algorithm develops on the basis of *TwigList* algorithm.

4.1. TwigList

Definition 5. XML node *v* meets the sub-query rooted at query node q, if *v* and some of its descendants match with the sub-tree rooted at *q* in twig pattern under Definition 2, denoted by *meet(v,q)*.

TwigList algorithm crieates a list L_i for every query nodes q_i. For every XML nodes *v* and query node *q* in a twig pattern, *v* will be added to the end of list in turn according to *lab(q)* if *meet(v, q)* is true. The algorithm makes preorder traversal of an XML document and process XML nodes in postorder traversal. A stack is used to converse the processing order. Each XML node matched with q_i has a set of range pointers point to each L_c (q_c is sub nodes of q_i) to indicate an interval, denoted by $interval_c$, which is composed of a *start* pointer and *end* pointer. *Start* pointers of its $interval_c$ are set to the end of L_c when an XML node matching with q_i is pushed into the stack, and their *end* pointers are set to the end of L_c when the XML node is popped up. Therefore, the XML nodes which are appended to every L_c during the period between the two operations must be in the interval. If every $interval_c$ is not empty, *meet(v, q_i)* holds for each XML node *v* matching with q_i and *v* should be appended to the

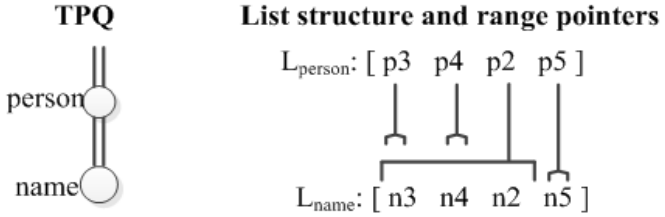

Figure 4. The List Structure of TwigList

corresponding list. In a twig pattern, PC edges are handled as same as AD edges, but a brother chain is attached to the lists to connect the XML nodes that meet PC relationship with its parent node. After traverse entire XML document, starting from the list of root query node, user can enumerate query results via XML nodes and their range pointers. Figure 4 shows the list structure for query *person//name* against the document shown in Figure 2(b). Here, XML node *p2* has an interval pointer to list L_{name} and it indicates that node *n3*, *n4* and *n2* are its children. It should be noted that the nodes in these list are arrayed in the XML document order.

4.2. Recursive Twig Matching

Root query node and recursive node are special treated in recursive twig matching algorithm in order to match all query results that entire recursion period needs and make matching results to represent the relations between different recursion call levels. The algorithm is divided into two phases: matching and enumeration. The matching algorithm is used to construct list similar to *TwigList* and the enumeration algorithm provides a unified interface to obtain sequence of XML nodes from the list according to the query node of twig pattern.

As shown in Figure 3(b), the matching algorithm should be able to match *p1* with *person($a)*, other *person* node with *person($n)*, *name* node with *name($r1)* and *child* node with query node *child*. Recursive hierarchical relationships between them should be recorded too. Enumeration algorithm should be able to enumerate data depending on recursion call level. At first level, its basic matching result should be found and recursive matching results

should be enumerated at other call levels. For example, at first recursion call level it should be able to enumerate sequence of nodes *p2*, *p5* and *p6* by the query node *person($n)* and they are enumerated by iteration variable *$n* one by one as the argument and is passed into the next recursion call level.

TwigR-Enumeration (Q, $\{Lv_1,...,Lv_n\}$, s, V_q, b)

Input: twig pattern $Q=(N,E,bind,lab,root)$,
lists Lv_i ($1 \le i \le n$), parameter s, given query node V_q, boolean b.
Output: sequence *seq* of XML nodes from Lv_q.

```
1.  Begin
2.  seq ← ∅;
3.  while Lv1≠∅ do
4.    v = next(Lv1);
5.    if v = next(s) then
6.      path ← path from V1 to Vq in Q;
7.      while path≠∅ do
8.        V ← next(path);
9.        if b = true then
10.         EnumNodes(v.start, v.end, V, Vq, seq)
11.       else
12.         EnumNodes(v.rstart, v.rend, V, Vq, seq)
13.     end of while;
14. end of while;
15. End.
Procedure EnumNodes(start, end, V, Vq, seq)
16. for v∈ [start, end] do
17.   if (V, Vq, struct, type)∈ E then
18.     addToSeq(seq, v, Lvq, v.startvq, v.endvq);
19.   else
20.     Vc ← advance(path);
21.     EnumNodes(v.startvc, v.endvc, Vc, Vq, seq);
22. end of for.
Procedure addToSeq (seq, v, L, start, end)
23. for each i ∈ [start, end] do
24.   node ← L[i];
25.   if isAD(VC) or ischild(node, v) then
26.     if node.traversed = false then
27.       addinto(seq, node);
28.       node.traversed ← true;
29. end of for.
```

Figure 7. Enumeration Algorithm

The matching algorithm **TwigR-Construction** is outlined in Figure 5. Its inputs are the recursive twig pattern Q with n query nodes, XML nodes sequence s that is value of the argument of a function call in main query body. Each input XML nodes is accompanied by the encoding which represents structural relationship with others. Output is list Lv_i built for each query node V_i. In the algorithm, v represents XML node, V represents its corresponding

query node, where V_1 is root query node. Each node v maintains pairs of pointers $v.start_{vp}$ and $v.end_{vp}$ to specify the interval for its V_p-type descendants in lists Lv_p, where V_p are child nodes of V. If v is a recursive query node, there are another pairs of pointers $v.rstart_{vk}$ and $v.rend_{vk}$ to specify the interval for its V_k-type descendants in lists Lv_k, where V_k represents each child node of V in the twig pattern.

Initialization phase of the algorithm is line 2 and 3. It builds empty stack and lists and assigns all input XML nodes in preorder to different label streams X_i according to the type of query nodes. The stream typed by root query node only keeps XML nodes bound by the parameter in function call expression, only these XML nodes can be put into the list of root query node eventually, while the rest of XML nodes will not be placed in the list even if they have the same type as root query node. Starting from line 4, current node v is picked out from X_q which the top element is the first following preorder traversal among all top elements in all X_i. Before v is pushed into the stack S, *top(S)* is popped up if v is not its ancestor and procedure *node2list* is called to check whether it matches with a sub-tree of the current XML document. For node V_m popped up from stack, it will be appended to the tail of Lv_m if $vm.start_{vn} \leq vm.end_{vn}$ holds for each its mandatory child V_n. Both end pointers *v.end* and *v.rend* for every child are set in *node2list*. Both start pointers $v.start_{vk}$ and $v.rstart_{vk}$ for its every child is set to $length(L_{vk})+1$ before it is pushed into stack. After traversing all XML nodes, remaining nodes in the stack S will be processed in turn.

Suppose a twig pattern shown in Figure 3(b) matches with the XML document shown in Figure 2(b). Figure 6 shows all lists L_{vi} and stack S, when it has traversed to *p5*. List $L_{'person}$ here corresponds recursive query node *person($n)*. Before *p5* is pushed into the stack, nodes *p4*, *c2* and *p2* are popped-up since they are not ancestors of *p5*. Then, they are checked to ensure that $meet(p4,q_{person})$, $meet(c2,p_{child})$ and $meet(p2,q_{person})$ hold, so they are put into corresponding lists. Double dashed lines indicate range pointers. Here the lines from recursive query node indicate range pointers *rstart* and *rend*. Obviously, *c2*, *p3* and *p4* can be accessed by range pointers from *p2*. Finally, after all XML nodes have been traversed, *p1* will be placed in the list of root query node and the matching is completed.

4.3. Enumeration

As mentioned above, the recursive query node and its range pointer *rstart* and *rend* acts as a link between different recursion call levels. For different call level, it is necessary to enumerates different group of XML nodes from the matching results in lists. Based on the XML nodes bound by the function's parameter, enumeration algorithm gets basic query results via range pointers *start* and *end* if it is at first recursion call level, otherwise gets recursive query results via range pointers *rstart* and *rend*.

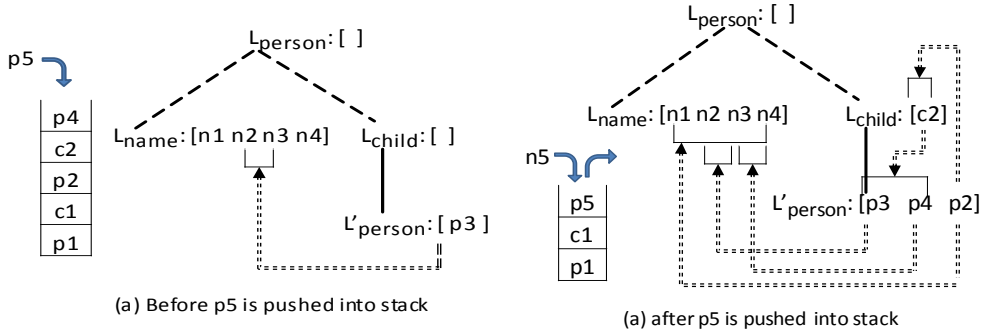

Figure 6. The Stack and List Structures of Recursive Twig Matching Algorithm

TwigR-Construction(*Q*, *s*)

Input: recursive twig patten Q with query nodes $\{V_1 \ldots V_n\}$, XML node sequence s
Output: Lv_i for V_i, $1 \le i \le n$.

```
1.  Begin
2.    initialize label streams Xi for Vi, 1≤i≤n, and put s into X1 and divide all Vi-typed nodes
      in its descendants into Xi;
3.    initialize S and Lvi(1≤i≤n) to ∅;
4.    while ∃ Xi ≠ ∅(1≤i≤n) do
5.       v ← removetop(Xq), where v is the first following the preorder traversal among
         all top elements in all Xi, and Vq is the corresponding query node.
6.       while S ≠ ∅ and noancestor(top(S), v) do
7.          node2list(pop(S));
8.       for each child Vp of Vq in Q do
9.          v.startvp ← length(Lvp) + 1;
10.      if Vq is the recursive node then
11.         for each child Vk of V1 do
12.            v.rstartvk ← length(Lvk) + 1;
13.      push(S, v);
14.   while S ≠ ∅ do
15.      node2list(pop(S));
16.   End

Procedure: node2list(Vm)
1.    for each child Vn of Vm do
2.       v.endvn ← length(Lvn);
3.    if Vm is the recursive node then
4.       for each child Vk of V1 do
5.          v.rendvk ← length(Lvk);
6.    for each mandatory child Vn of Vm do
7.       if vm.startvn > vm.endvn return;
8.    append(Vm, Lvm);
```

Figure 5. The Recursive Twig Matching Algorithm

The enumerate algorithm **TwigR-Enumeration** is outlined in Figure 7. It is used to get matching results of a given query node. Inputs are recursive query pattern Q, all lists Lv_i, real parameter s of a recursive function which is node sequence, given query node V_q and a boolean value b indicates that basic query result or recursive query result is looking forward. Its output is XML nodes in $List_{vq}$ which can be reached by the range pointers from XML nodes in s. When a recursive function is first called, b is set as true and all XML nodes in $List_{v1}$ are used as input s. Subsequently, whenever the recursive function is called, XML nodes bound by its parameter are used as s and a recursive query results can be gotten by their *rstart* and *rend* pointers. Sub procedure *addToSeq* is in charge of putting XML nodes to the result sequence. To cope with PC edges in twig pattern, it only adds descendent nodes that meet PC relationship to result sequence by comparing encoding.

4.4. Analysis

Given a twig pattern Q with n query nodes and an XML document t. The time and space complexity of **TwigR-Contruct** is $O(d \cdot |X| + r \cdot |Y|)$ in the worst case, where $|X|$ is the total number of V_i-typed XML nodes $v_i (1 \le i \le n)$, d is the max degree of a query node, $|Y|$ is the total

number of V_r-typed XML nodes that V_r is recursive query node, and r is the degree of root query node. It is linear w.r.t $|X|$. The algorithm just need performs once during entire recursion period, saving the cost of creating context for matching algorithm and maintaining intermediate result. Therefore, it should have better performance than ordinary twig matching algorithm which needs perform repeatedly at each recursion call level.

4.5. Implementation

We have implemented the *RTPQ* matching algorithm in an XQuery engine. In this engine, the XQuery query request is translated to a query plan, and then it is transformed into one with *RTPQ* by a twig pattern extracting algorithm. During the query plan is executed, *TwigR-Contruct* is used to get intermediate result before a recursive function is called at the first time. In the evaluation of the body expression of the function, **TwigR-Enumeration** is used to obtain the query result for the current recursion call level.

5. Related Works

There are some optimization methods for processing recursive query in XQuery. [9] optimizes a kind of structurally recursive queries by function inlining. During the function inlining, it embeds as much type information as possible into iteration parameters to prune the target data gradually, and is followed by algebraic simplification. It not only reduces the overhead of function calls but also avoids a large number of useless searches for XML document. [10] introduces a controlled form of recursion in XQuery, an inflationary fixed point operator based on the algebra XAT, to optimize recursive function in XQuery and transitive closure in some particular cases. The execution of recursive function can be performed by evaluation of the operator which iteratively computes a data collection. The iterative evaluation can be optimized according to the distributivity of XQuery expressions. The XQuery processors can benefit substantially from the mode of evaluation.

In works above, twig queries in XQuery functions have not be taken into account. It can be processed by existing twig query algorithms individually, but multiple twig queries in recursive function callings can be combined into a single *RTPQ* query by our approach in this paper.

[2] proposes a XML query language XSeq, which extends XPath expression with Kleene closure. XQuery recursive functions can express the same semantics as the extended XPath expression. Therefore, our algorithm for *RTPQ* can be also used to implement the new feature of the query languages effectively. Moreover, *RTPQ* is more powerful than Kleene closure in XPath expression since it can contain multiple recursive nodes.

6. Experiments

We test several recursive queries on above XQuery engine, in which the queries are implemented by three approaches respectively, including RTPQ, TwigList and traditional navigating way (NAV). All experiments are performed on a Core i3-2120 3.30GHz processor PC with 4GB RAM running on Windows7 system. Because existing benchmarks have few XML documents with deep recursive structure, the documents used in experiments are constructed in random content according to DTDs shown in Figure 1. Specifically, a group of them has the same structure and about 5.7 average depths, other groups have different average depths.

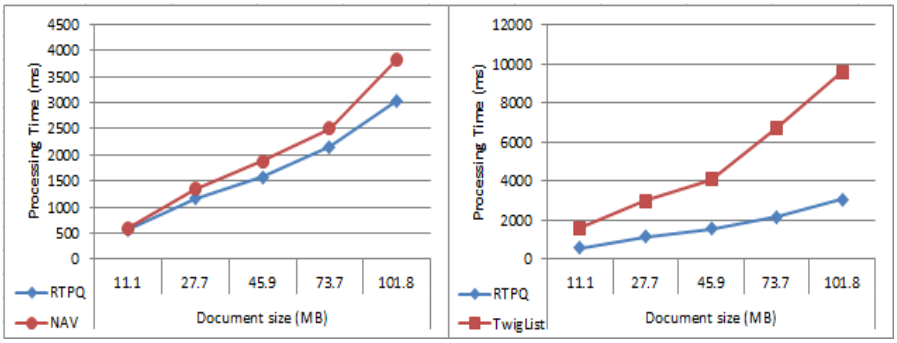

Figure 8. Test Result with Different Document Size

The traditional twig pattern cannot match data for whole recursion, its matching procedure and enumeration procedure should be invoked over and over again for each recursive function call. As show in Figure 8, traditional *TwigList* is at least twice the time cost of *RTPQ*, and the rate grows to 4 times as the document size grows to 101MB. *RTPQ* is a little faster than *NAV*, but the difference increases with the raising of document.

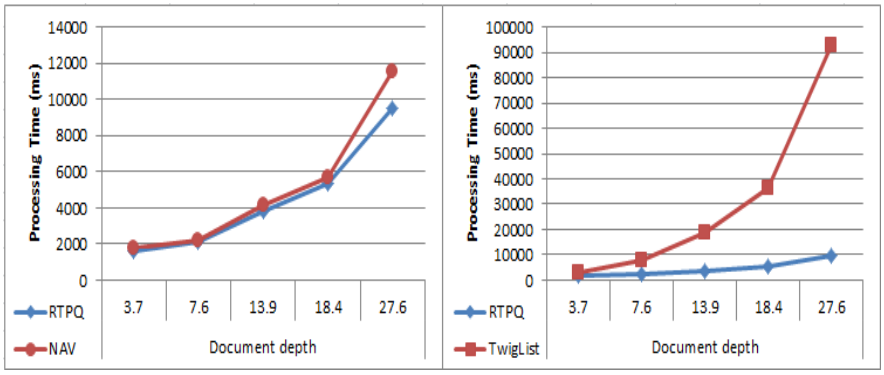

Figure 9. Test Result by Simple Twig Patterns with Different Document Depth

The complexity of recursive twig pattern is expressed by the number of branch. For the recursive twig pattern with a single branch, the result used in Figure 9 shows that the time cost for *NAV* grows exponentially with recursion depth, but for *RTPQ*, the cost grows slowly. The gap between them is growing up to 8 times when the document average depth is 27.6. Contrasting *RTPQ* and *NAV*, the efficiency difference of them is small since the twig pattern is simple. Test results for some complicated twig pattern are shown in Figure 10. Here, time cost of *NAV* is more than 2 times that of *RTPQ*, and the gap grows rapidly with the depth of document. It means that *RTPQ* outperform *NAV* and *TwigList* for recursive query with complicate twig pattern.

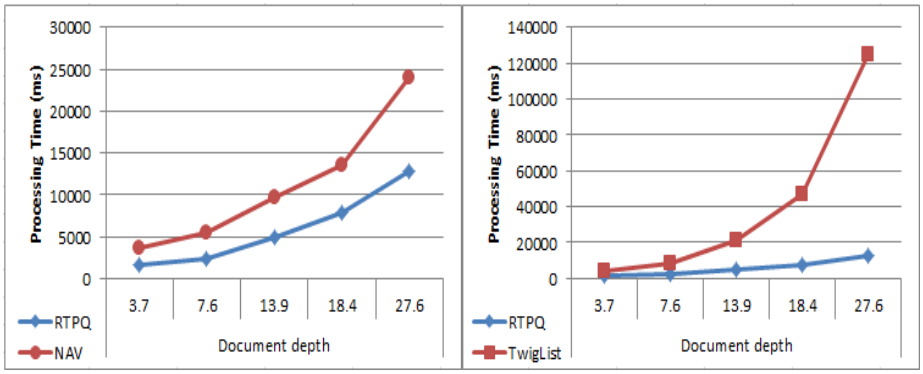

Figure 10. Test Result by Complex Twig Pattern with Different Document Depth

7. Conclusion

This paper proposes a kind of twig query called recursive twig pattern query (RTPQ) and a holistic twig matching algorithm for the query pattern. RTPQ can describe the recursive query request in both XQuery user-defined functions and extended XPath with Kleen closure. The algorithm can be used to obtain all the data which is needed during whole recursion period. These data can be enumerated by the enumeration algorithm at every recursive function call level. RTPQ is evaluated only once during whole recursion period, its efficiency is much better than performing traditional twig pattern matching.

Acknowledgements

This work was supported in part by the Beijing Nature Science Foundation under Grant 4122011 and the National Science Foundation for Young Scientists of China under Grant 61202074.

References

[1] H. Marouane and D. Jérôme, "A Survey of XML Tree Patterns", IEEE Trans. Knowledge and Data Engineering, vol. 1, no. 25, **(2013)**.
[2] M. Barzan, Z. Kai and Z. Carlo, "High-performance complex event processing over XML streams", Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, Scottsdale, Arizona, USA, **(2012)** May 20-24.
[3] B. Nicolas, K. Nick and S. Divesh, "Holistic twig joins: optimal XML pattern matching", Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, **(2002)** June 3-6.
[4] C. Songting, L. Hua-Gang, T. Junichi, H. Wang-Pin, A. Divyakant and C. K. Selçuk, "Twig2Stack: bottom-up processing of generalized-tree-pattern queries over XML documents", Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, **(2006)** September 12-15.
[5] Q. Lu, X. Y. Jeffrey and D. Bolin, "TwigList: make twig pattern matching fast", Proceedings of the 12th International Conference on Database Systems for Advanced Applications, Bangkok, Thailand, **(2007)** April 9-12.
[6] J. Lu, T. Chen and T. W. Ling, "TJFast: efficient processing of XML twig pattern matching", Proceedings of the 14th International World Wide Web Conference, Chiba, Japan, **(2005)** May 10-14.
[7] F. Mandreoli, R. Martogia and P. Zezula, "Principles of Holism for sequential twig pattern matching", The VLDB Journal, vol. 18, no. 6, **(2009)**.
[8] J. Lu, T. W. Ling, Z. F. Bao and C. Wang, "Extended XML tree pattern matching: theory and algorithms", IEEE Transaction on Knowledge and Data Engineering, vol. 23, no. 3, **(2011)**.

[9] P. Chang-Won, M. Jun-Ki and C. Chin-Wan, "Structural function in lining technique for structurally recursive XML queries", Proceedings of the 28th International Conference on Very Large Data Bases, Hong Kong, China, **(2002)** August 20-23.
[10] A. Loredana, G. Torsten, M. Maarten, R. Jan and T. Jens, Recursion in XQuery: put your distributivity safety belt on. Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, **(2009)** March 23-26, Saint-Petersburg, Russia.

Authors

Husheng Liao, received his M.Sc degree from TsingHua University in 1981. He is currently a professor of Beijing University of Technology. His current research interests include compiler technology, programming language and XML database.

Hongyu Gao, received his M.Sc. degree in Beijing University of Technology in 1995. He is currently an associate professor of Beijing University of Technology. His researches interests include compiler technology and XML database.

Zhaoning Guan, received his M.Sc. degree in Beijing University of Technology in 2013. His researches interests include XML database and query languages.

