International Journal of Electronics Communication and Computer Engineering
Volume 3, Issue 5, ISSN (Online): 2249–071X, ISSN (Print): 2278–4209

XML Tree Pattern Matching Algorithms

V. S. N. Deepthi
(M.Tech.)
Pydah Engineering College, A.P., India.

A. Rajya Lakshmi
Asst. Professor, Dept. of CSE,
Pydah Engineering College, A.P., India.

P. Suresh Babu
Assoc. Professor, Dept. of CSE,
Kaushik Engineering College, A.P., India.

Abstract **— An XML database is a data persistence software system that allows data to be stored in XML format. This data can then be queried, exported and serialized into the desired format. XML databases are usually associated with document-oriented databases. The XML documents are usually modelled as trees and queries in XML query languages and are typically twig (or small tree) patterns with some nodes having value-based predicates. Therefore, finding all distinct matching's of a twig pattern becomes a core operation in an XML query evaluation. This paper presents two algorithms for XML tree pattern matching. The TwigStack algorithm is used to compute answers to a query twig pattern. This algorithm solves the problem of larger intermediate results with decomposition-matching-merging methods. When the patterns have only ancestor-descendant edges, the intermediate result of each path matching is guaranteed to be part of the final result. The TreeMatch algorithm defines an extended XML tree pattern (twig) means P-C, A-D, negation, wildcard and order restriction. This algorithm provides tree matching for class Q/, //, *, class Q/, //, *, <, *class* Q/, //, *, <, ¬.**

Keywords **— Query processing, Tree pattern matching, Twigs, XML.**

I. INTRODUCTION

An XML database [1] is a data persistence software system that allows data to be stored in XML format. This data can then be queried, exported and serialized into the desired format. XML databases are usually associated with document-oriented databases.

XML employs a tree structured data model. Each node of an XML tree has a label and a unique node identifier. Leaves may also carry some content, which for the purposes of our study can only be a string. The root of the tree is called the document node. AN XML tree is defined as follows:

An XML tree over a possibly infinite set of strings S is a tree t = (N, E, r), where:

- N is a finite set of nodes. Each node n ∈ N has a label from S, denoted with label(n), and a unique node identifier, denoted with id(n). Leaf nodes may also carry some content of string type.
- E ⊆N X N is a finite set of edges.
- r ∈ N is the root of the tree, which is labeled "root".

Given an XML tree t = (Nt,Et, rt), we say that a node u ∈Nt is a descendant of a node v ∈ Nt, if there is a path from v to u in t, that is if there is a sequence of edges of Et, (v, n1), (n1, n2),……, (nk, u), k >= 0. If k = 0, that is if there is an edge (v, u) ∈ Et, we say that u is a child of v in t. We say that an XML tree t0 = (Nt^1, Et^1, rt^1) is a subtree of t if $Nt^1 \subseteq$ Nt, Et^1 = (Nt^1 X Nt^1) ∩ Et, and Nt^1 contains all the nodes of Nt that are descendants of rt^1 . We denote the subtree of t rooted at some node n with (t)n.

A convenient way to query tree-structured databases, such as XML databases, is through tree-structured queries, called tree pattern queries[2]. A tree pattern query is defined as follows:

A tree pattern query (TP query or simply TP) p over a possibly infinite set of strings S is a tree p = (Np, Ep, rp, Rp, Cp), where:

- Np is a finite set of nodes. Each node n ∈ Np has a label from S, denoted with label(n), and a unique node identifier, denoted with id(n).
- Ep ⊆ Np X Np is a finite set of edges. Edges can be either of child type, denoted with solid lines, or of descendant type, denoted with dashed lines. Edges of child type are called c-edges, and edges of descendant type are called d-edges. If (u, v) is a c-edge, v is called a c-child of u. If (u, v) is a d-edge, v is called a d-child of u. C-children and d-children of a node are collectively called the node's children.
- rp ∈ Np is the root of p.
- Rp ∈ Np is the selection node or result node of p.
- Cp is a set of equality predicates of the form value(n) = c, where n ∈ Np is a leaf node, and c is a string constant.

Two major classes of XML database exist:

- *XML-enabled:* These map all XML to a traditional database (such as a relational database), accepting XML as input and rendering XML as output. This term implies that the database does the conversion itself (as opposed to relying on middleware).
- *Native XML (NXD):* the internal model of such databases depends on XML and uses XML documents as the fundamental unit of storage, which are, however, not necessarily stored in the form of text files.

XML data is more often generated by business and enterprises. So there is an increasing need for efficient processing of queries on XML data. Searching for the occurrences of a tree pattern query in an XML database is a core operation in XML query processing. This paper presents algorithms for XML tree pattern matching.

II. TWIG PATTERN MATCHING ALGORITHM

An XML database is a forest of rooted, ordered, labeled trees, each node corresponding to an element or a value, and the edges representing (direct) element-subelement or element-value relationships. Node labels consist of a set of (attribute, value) pairs, which suffices to model tags, IDs, IDREFs, etc. The ordering of sibling nodes implicitly defines a total order on the nodes in a tree, obtained by a

International Journal of Electronics Communication and Computer Engineering
Volume 3, Issue 5, ISSN (Online): 2249–071X, ISSN (Print): 2278–4209

preorder traversal of the tree nodes. Figure 1 shows the tree representation of a sample XML document.

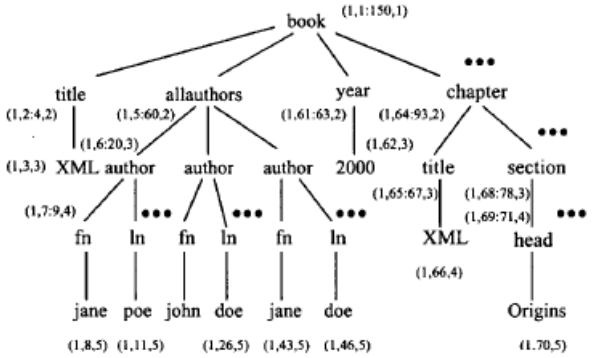

Fig.1. A sample XML tree representation

Queries in XML query languages like XQuery [3], Quilt [4] and XML-QL [5] make use of (node labeled) twig patterns [6] for matching relevant portions of data in the XML database. The twig pattern node labels include element tags, attribute- value comparisons, and string values, and the query twig pattern edges are either parent-child edges (depicted using a single line) or ancestor-descendant edges (depicted using a double line). For example, the XQuery expression:

Book [title = 'XML' AND year = '2000']

which matches book elements that (i) have a child title subelement with content XML, and (ii) have a child year subelement with content 2000, can be represented as the twig pattern in Figure 2(a). Only parent-child edges are used in this case. Similarly, the XQuery expression in the introduction can be represented as the twig pattern in Figure 2(b). Note that an ancestor-descendant edge is used between the book element and the author element. In general, at each node in the query twig pattern, there is a node predicate on the attributes (e.g., tag, content) of the node in question.

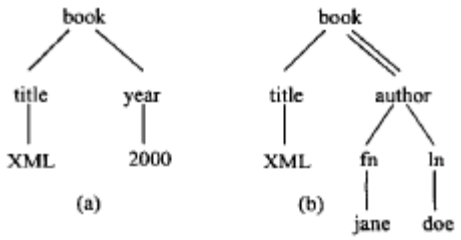

Fig.2. Query twig patterns

The following illustrates algorithm TwigStack which computes answers to a query twig pattern.

Algorithm TwigStack (q)

// Phase 1

1 while ¬end(q)

2 q_{act} = getNext(q)

3 if (¬isRoot(q_{act}))

4 c1eanStack(parent(q_{act}). nextL(q_{act}))

5 if (isRoot(q_{act}) V ¬empty(Sparent(q_{act})))

6 cleanStack(q_{act}, next(q_{act}))

7 moveStreamToStack(T q_{act} , S q_{act}, pointer to top(Sparent(q_{act})))

8 if (isLeaf(q_{act}))

9 showSoiutionswithBlocking (S q_{act}, 1)

10 pop(S q_{act})

11 else advance(T q_{act})

// Phase 2

12 mergeAllPathSolutions ()

Function getNext (q)

1 if (isLeaf(q)) return q

2 for qi in children(q)

3 ni = getNext(qi)

4 if (ni ≠ qi) return ni

5 nmin = minargni nextL(Tni)

6 nmax = maxargni nextL(Tni)

7 while (nextR(Tq) <nextL(Tnmax))

8 advance (Tq)

9 if (nextL(Tq) <nextL(Tnmin) return q

10 else return nmin

Procedure cleanStack(S, actL)

1 while (¬empty(S) ∧ (topR(S) < actL))

2 pop(S)

Algorithm TwigStack operates in two phases. In the first phase (lines 1-11), some (but not all) solutions to individual query root-to-leaf paths are computed. In the second phase (line 12), these solutions are merge-joined to compute the answers to the query twig pattern.

III. TREEMATCH ALGORITHM

Twig pattern matching algorithm is an efficient technique to answer an XML tree pattern with parent-child (P-C) and ancestor-descendant (A-D) relationships, as it can effectively control the size of intermediate results during query processing. TreeMatch algorithm[7] include P-C, A-D relationships, negation functions, wildcards and order restriction as extended XML tree pattern. Figure 3, for example, shows four extended XML tree patterns. Query (a) includes a wildcard node "*", which can match any single node in an XML database. Query (b) includes a negative edge, denoted by '¬'. This query finds A that has a child B, but has no child C. Query (c) has the order restriction, which is equivalent to an XPath "//A/B[following-sibling::C]". The '<' in a box shows that all children under A are ordered. The semantics of order-base tree pattern is captured by a mapping from the pattern nodes to nodes in an XML database such that the structural and ordered relationships are satisfied. Finally, Query (d) is more complicated, which contains wildcards, negation function and order restriction.

International Journal of Electronics Communication and Computer Engineering
Volume 3, Issue 5, ISSN (Online): 2249–071X, ISSN (Print): 2278–4209

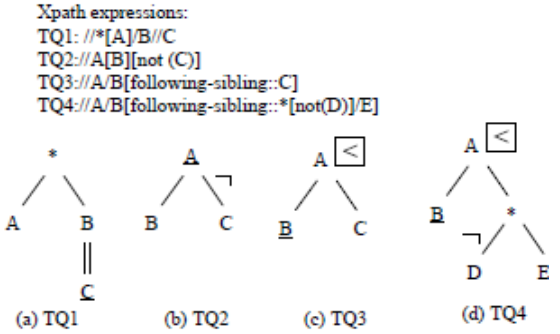

Fig.3. Example extended XML tree pattern queries

The following illustrates data structures and notations for query class Q / //, *: There is an input list Tq associated with each query node q, in which all the elements have the same tag name q. Thus, we use eq to refer to these elements. Cur(Tq) denotes the current element pointed by the cursor of Tq. The cursor can be advanced to the next element in Tq with the procedure advance(Tq). There is a set Sq associated with each branching query node q (not each query node). Each element eq in sets consists of a three-tuple (label, bitV ector, outputList). label is the extended Dewey label of eq. bitV ector is used to demonstrate whether the current element has the proper children or descendant elements in the document. Specifically, the length of bitV ector(eq) equals to the number of child nodes of q. Given a node qc2 children(q), we use bitV ector(eq)[qc] to denote the bit for qc. Specifically, bitV ector(eq)[qc]= ‘‘1’’ if and only if there is an element eqc in the document such that the eq and eqc satisfy the query relationship between q and qc. Finally, outputList contains elements that potentially contribute to final query answers.

In algorithm, we will frequently use the following two notations. 1) NAB(q) denotes the Nearest Ancestor Branching node of q in the query pattern Q. Formally, q^1 =NAB(q) if and only if q^1 is a branching node and q^1 is an ancestor of q and there is no other branching node q^{11} s.t. q^{11} is in the path from q^1 to q. If there is no such ancestor of q, then NAB(q) denotes the top branching node in query. 2) NDB(q) denotes the nearest descendants branching (or leaf) nodes of q. Formally, q^1=2 NDB(q) if and only if q^1 is a branching or leaf node and q^1 is a descendant of q and there is no other branching or leaf node q^{11} s.t. q^{11} is in the path from q^1 to q.

TreeMatch Algorithm:

Algorithm TreeMatch for class Q/, //, *.
1: locateMatchLabel(Q);
2: while (¬end(root)) do
3: f_{act} = getNext(topBranchingNode);
4: if (f_{act} is a return node)
5: addToOutputList(NAB(f_{act}), cur($T_{f_{act}}$));
6: advance($T_{f_{act}}$); // read the next element in $T_{f_{act}}$
7: updateSet(f_{act}); // update set encoding
8: locateMatchLabel(Q); // locate next element with matching path
9: emptyAllSets(root);

Now we go through Algorithm . Line 1 locates the first elements whose paths match the individual root-leaf path pattern. In each iteration, a leaf node f_{act} is selected by getNext function (line 3). The purpose of lines 4 and 5 is to insert the potential matching elements to outputlist. Line 6 advances the list $T_{f_{act}}$ and line 7 updates the set encoding. Line 8 locates the next matching element to the individual path. Finally, when all data have been processed, we need to empty all sets in Procedure EmptyAllSets (line 9) to guarantee the completeness of output solutions.

The procedures and algorithms in TreeMatch algorithm is as follows:

Procedure locateMatchLabel(Q)
1: for each leaf $q \in Q$, locate the extended Dewey label *eq* in list *Tq* such that *eq* matches the individual root-leaf path

Procedure addToOutputList(q, eqi)
1: for *each eq* ∈ *Sq* do
2: if (satisfyTreePattern(*eqi ,eq*))
3: *outputList*(*eq*). *add*(*eqi*);

Function satisfyTreePattern(eqi ,eq)
1: if (*bitV ector*(*eq, qi*) = ‘1’) return true;
2: else return false;

Procedure updateSet(q, e)
1: *cleanSet*(*q, e*);
2: add *e* to set *Sq*; //set the proper *bitV ector*(*e*)
3: if (¬*isRoot*(*q*) ∧ (*bitV ector*(*e*)=“1...1”))
updateAncestorSet(*q*);

Procedure cleanSet(q, e)
1: for each element *eq* ∈ *Sq* do
2: if (satisfyTreePattern(*eq,e*))
3: if (*q* is a return node)
4: *addToOutputList*(*NAB*(*q*), *e*);
5: if (*isTopBranching*(*q*))
6: if (there is only one element in *Sq*)
7: output all elements in *outputList*(*eq*);
8: else merge all elements in *outputList*(*eq*) to *outputList*(*ea*), where *ea*=NAB(*eq*);
9: delete *eq* from set *Sq*;

Procedure updateAncestorSet(q)
1: /*assume that $_{q^1}$ = NAB(*q*)*/
2: for each $e \in S_{q^1}$ do
3: if (*bitV ector*(*e*, *q*) = 0)
4: *bitV ector*(*e; q*) = 1;
5: if (¬*isRoot*(*q*) ∧ (*bitV ector*(*e*)=“1...1”))
6: *updateAncestorSet*($_{q^1}$);

Procedure emptyAllSets(q)
1: if (*q* is not a leaf node)
2: for each child *c* of *q* do EmptyAllSets(c);
3: for each element $e \in Sq$ do cleanSet(*q; e*);

Algorithm : getNext(n)
1: if (isLeaf(*n*)) then

2: return *n*
3: else
4: for each *ni* ∈ NDB(*n*) do
5: *fi* = *getNext*(*ni*)
6: if (isBranching(*ni*) ∧ ¬*empty*(*Sni*))
7: return *fi*
8: else *ei* = *ma{p|p* ∈ *MB*(*ni*, *n*)*g}*
9: end for
10: max = *maxargi{ei}*
11: for each *ni* ∈ NDB(*n*) do
12: if (∀ *e*∈ *MB*(*ni*, *n*) : *e*∉ *ancestors*(*emax*))
13: return *fi*;
14: endif
15: end for
16: min = *minargi{fi| fi is not a return node}*
17: for each *e* ∈ MB(*nmin*, *n*)
18: if (*e*∈ *ancestors*(*emax*)) updateSet(*Sn*, *e*)
19: end for
20: return *fmin*
21: end if

Function MB(n, b)

1: if (isBranching(*n*)) then
2: Let *e* be the maximal element in set *Sn*
3: else
4: Let *e* = cur(*Tn*)
5: end if
6: Return a set of element *a* that is an ancestor of *e* such that *a* can match node *b* in the path solution of *e* to path pattern pn

The following algorithm describes the extended TreeMatch algorithm for answering ordered tree queries. The purpose of the extension is to maintain and check the order relationship among the matching elements of query sibling nodes. In line 2 of Procedure updateSet, we need to set the proper *minChild* and *maxChild* according to the current elements. In Function satisfyTreePattern, we also need to check the order restriction according to *minChild* and *maxChild*.

*Algorithm TreeMatch for class Q/, //, *, <*

Procedure updateSet(q; e)

1: *cleanSet*(*q*, *e*);
2: add *e* to set *Sq*; //set the proper *bitV ector*, *minChild* and *maxChild*
3: if (¬ *isRoot*(*q*) ∧ (*bitV ector*(*e*)="1...1"))
updateAncestorSet(*q*);

Function satisfyTreePattern(qi,eq)

1: assume that child nodes of *q* in Q are *q*1, ..., *qn* (in order)
2: if (*eqi* < *minChild*(*eq*, *qi*-1)) return false;
3: else if (*eqi* > *maxChild*(*eq*, *qi*+1)) return false;
4: else if (*bitV ector*(*eq*, *qi*) = '1') return true;
5: else return false;

The following algorithm extend TreeMatch to support negative edges.

*Algorithm TreeMatch for class Q/, //, *, <, ¬*

Procedure updateSet(q, e)

1: *cleanSet*(*q*, *e*);
2: add *e* to set *Sq*; //set the proper *bitV ector*, negBitV ector, minChild and maxChild
3: if (¬ *isRoot*(*q*) ∧ (*bitV ector*(*e*)="1...1"))
updateAncestorSet(*q*);

Function satisfyTreePattern(qi, eq)

1: if (*eqi* < *minChild*(*eq*, *qi*-1)) return false;
2: else if (*eqi* > *maxChild*(*eq*, *qi*+1)) return false;
3: else if ((*bitV ector*(*eq*)[*qi*]) = '1') and (*negBitV ector*(*eq*)[*qi*]) = '0'))
4: return true;
5: else return false;

IV. CONCLUSION

In this paper we present an overview of XML query processing. XML tree pattern query (TPQ) processing is a research stream within XML data management that focuses on efficient TPQ answering. With the increasing popularity of XML for data representation, there is a lot of interest in query processing over data that conforms to a tree-structured data model. Queries on XML data are commonly expressed in the form of tree patterns (or twig patterns), which represent a very useful subset of XPath and XQuery. TwigStack algorithm is used to compute answers to a query twig pattern. TreeMatch algorithm defines an extended XML tree pattern (twig) means P-C, A-D, negation, wildcard and order restriction.

REFERENCES

[1] H. V. Jagadish and S. AL-Khalifa. Timber: A native XML database. Technical report, University of Michigan, 2002.
[2] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. of ICDE Conference*, pages 141–152, 2002.
[3] Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu XQuery 1.0: An XML Query Language. W3C Working Draft. Available from http://www.w3.org/TR/xquery, Dec. 2001.
[4] D. D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In WebDB (Informal Proceedings), 2000.
[5] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. Available from http://www.w3.org/TR/NOTE-xml-ql., 1998.
[6] T. Chen, J. Lu, and T. W. Ling. On boosting holism in xml twig pattern matching using structural indexing techniques. In *SIGMOD*, pages 455– 466, 2005.
[7] J. Lu, T. W. Ling, Z. Bao, and C. Wang. Extended xml tree pattern matching: theories and algorithms. In *Technical Report*, 2010.