Asian Journal of Information Technology 15 (4): 756-764, 2016
ISSN: 1682-3915
© Medwell Journals, 2016

Intra and Inter XML Query Answering Using Holistic Boolean Twig Pattern Matching

J.C. Miraclin Joyce Pamila and Divya Rajagopal
Department of Computer Science and Engineering, Government College of Technology,
Tamil Nadu, India

Abstract: The XML is used as a standard for expressing semi structured data, a form of data that does not conform to the formal structure of relational data models but nonetheless contains tags to separate semantic elements and enforce hierarchies within the data. Semi structured data model allows information from several sources with related but different properties to be integrated, thereby enabling sharing of information. Answering the queries issued on such data is therefore, a great challenge. Twig pattern matching is a critical operation for XML query answering and holistic approaches have shown superior performance over other methods. The existing holistic approaches research on handling Boolean twigs, twigs which contain arbitrary occurrences of the logical connectives AND, OR and NOT. In this study, we extend the holistic Boolean twig pattern matching approach to support two different query formats, intra query and inter query which provide a different context to the query.

Key words: Ancestor-descendant, boolean, holistic, inter query, intra query, parent-child, XML

INTRODUCTION

An XML document consists of data enclosed within a set of user-defined tags. The tags should be properly nested, they should be paired and there should be one and only one root tag. The XML offers simplicity, flexibility, standardization and interoperability. Hence, XML is being widely used as a data representation format for representing nearly all kinds of data. However, XML documents are often very large and have a deeply nested structure. Further, the XML data can also be very complex. Hence pattern matching algorithms are needed to retrieve the data from such documents by answering the queries issued. A sample XML document is shown in algorithm 1.

A query on the XML document describes a tree-shaped or hierarchical search pattern which is often referred to as a twig pattern (Bruno *et al*., 2002). The XML queries are thus called tree queries or twigs and the relationships between the components of the twig are represented as edges. Single backslash (/) is used to represent a parent-child edge or PC edge. When/is used at the beginning of a query for example/book, it will define an absolute path to node "book" relative to the root. In this case, it will only find "book" nodes at the root of the XML tree. When/is used in the middle of a query for, e.g.,/book/author, it will define a path to node "author" that is a direct descendant (i.e., a child) of node "book". Double backslash (//) is used to represent an ancestor-descendant edge or AD edge. When//is used at the beginning of a query for example//book, it will define a path to node "book" anywhere within the XML document. In this case, it will find "book" nodes located at any depth within the XML tree. When//is used in the middle of a query for, e.g.,/book//author, it will define a path to node "author" that is any descendant of node "book".

The core operation of XML query answering is twig pattern matching: finding in an XML document tree 'D', all matches of a given tree-type query 'Q' called twig. A match is identified by a mapping from nodes in 'Q' to nodes in 'D' such that query node predicates are satisfied by the corresponding document tree nodes and also the structural relationships (AD or PC) between query nodes are satisfied by the corresponding document tree nodes.

The answer to query 'Q' with 'n' nodes can be represented as an n-ary relation where each tuple (d1,...,dn) consists of the document tree nodes that identify a distinct match of 'Q' in 'D'.

Holistic twig pattern matching approaches avoid large sets of irrelevant intermediate results by considering the structural inter-dependencies among the XML elements. Holistic approaches optimize pattern matching in two phases:

- Labeling: assigns to each node x in the data tree t, an integer label, label (x) that captures the structure of t
- Computing: exploits the labels to match a twig pattern p against t without traversing t again

Corresponding Author: J.C. Miraclin Joyce Pamila, Department of Computer Science and Engineering, Government College of Technology, Tamil Nadu India

A twig that contains a single path from root to leaf is called a plain twig. It does not contain any boolean predicates. A twig that may contain arbitrary combination of ANDs, ORs and NOTs is referred to as an AND/OR/NOT twig or Boolean-twig (or simply B-twig). Normalization is the important pre-processing module which is performed on the B-twig to effectively restrain the complexity in a B-twig. In this study, we extend the holistic Boolean twig pattern matching approach proposed in (Che *et al.*, 2012) in order to support two different query formats, intra query and inter query which provide a different context to the query.

When a logical Boolean predicate is used within a query as a condition such a query format is called intra query. The context represented by this query is as follows: 'AND' represents conjunction, e.g.,//book [[//author = Jack] AND [/editor = Jack]] finds the books of author Jack who is also the editor of the same book. The 'OR' represents disjunction (inclusive OR), e.g.,//book [[//author = Jack] OR [/editor = Jack]] finds the books either whose author is Jack or editor is Jack. 'NOT' represents negation, e.g.,//book [[//author = Jack] NOT [/editor = Jack]] finds the books of author Jack who is not that book's editor.

When a logical boolean predicate is used between two intra queries, such a query format is called inter query. The context of this query is: 'AND' represents and exists, e.g., {//book [//author = Jack]} AND {//book [/editor = Jack]} finds the books of author Jack who is also, the editor of any book. The 'OR' represents exclusive OR, e.g., {//book [//author = Jack]} OR {//book [/editor = Jack]} finds the books either whose author is Jack or editor is Jack but not both. 'NOT' represents not exits, e.g., {//book [//author = Jack]} NOT {//book [/editor = Jack]} finds the books of author Jack who is not the editor of any book.

MATERIALS AND METHODS

Existing systems: TwigStack is a holistic twig join algorithm that ensures that no large intermediate results are produced (Bruno *et al.*, 2002). When the query has only ancestor-descendant relationships between the elements, Twig Stack is I/O and CPU optimal for ancestor-descendant relationship but it is suboptimal when the query has parent-child relationship among the elements. It supports only plain twigs. The GT wig Merge (Jiang *et al.*, 2004), a basic framework for holistic processing of AND/OR-twigs researches correctly when AND/OR twig queries contain parent-child query nodes. However, the optimality in terms of worst-case I/O and CPU cost is no longer guaranteed. Twig Stack List (Lu *et al.*, 2004) is another holistic twig join algorithm which is I/O optimal for queries with only

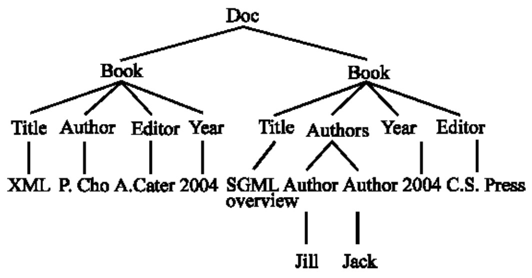


Fig. 1: Representation of XML tree

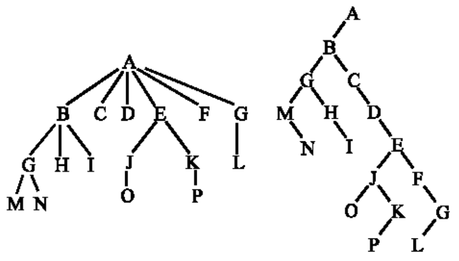


Fig. 2: Data structure for XML tree representation

ancestor-descendant relationships below branching nodes. The optimality cannot be proved for the case where parent-child relationships appear only in edges below non-branching nodes. TwigStack List¬ is an algorithm to match NOT-twig queries holistically (Yu *et al.*, 2006). In a NOT-twig, this algorithm can guarantee the I/O optimality only when all the positive edges below branching nodes are ancestor-descendant relationships. BT wig Merge (Che *et al.*, 2012) is a novel Holistic Twig Join algorithm which completely supports Boolean twigs. It performs optimal matching for B-twigs with AD and PC edges.

Tree representation of XML and twig: As the XML and the twig are hierarchical, they are represented using a tree data structure. Figure 1 shows the tree for the XML document referred in Fig. 1.

Each node in the tree corresponds to an XML element. The root node corresponds to the root element, the intermediate nodes to sub elements, the leaf node to values. Each edge corresponds to an element-sub element or element-value relationship. Each non-leaf node in the XML tree can have multiple, variable number of children. Hence, instead of a linked list implementation of the tree, a more optimized tree representation (Fig. 2) is used. In this representation, each non-leaf node has two pointers: a pointer to the first child and a pointer to the next sibling. The optimized XML tree representation of the tree used in Fig. 1 is demonstrated in Fig. 3. Because, each node has at most only two children, the new tree is a binary representation of the previous tree.

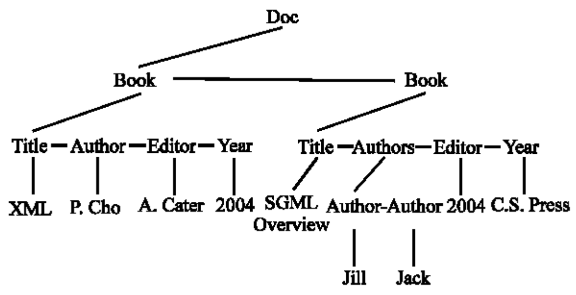


Fig. 3: Optimized representation of XML tree

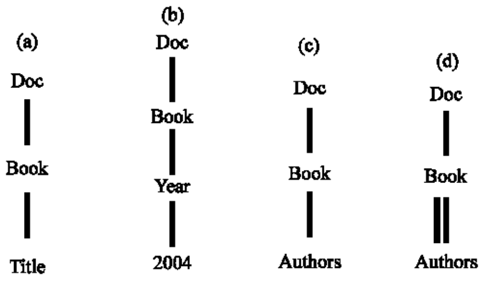


Fig. 4: Representation of XML plain query trees

Sample XML document:

```
< xml version = "1.0">
<doc>
<book>
        <title>XML </title>
        <author>P. Cho </author>
        <editor>A. Cater </editor>
        <year>2004</year>
<book>
<book>
        <title>SGML overview</<title>
<authors>
        <authors>Jill </author>
        <author>Jack </author>
<authors>
        <year>2004</year>
        <editor>C.S. Press </editor>
<book>
</doc>
```

Sample plain and Boolean query trees are shown in Fig. 4 and 5, respectively. A query tree can have four types of nodes: Q Node/query nodes, A node/AND nodes, O Node/OR nodes and N Node/NOT nodes.

XML tree labeling: The aim of data tree labeling schemes is to determine the relationship (i.e., Parent-child or ancestor-descendant) between two nodes of a tree from their labels alone. Each node in the XML tree is given a unique identity called label or region code. In this study, the triplet region encoding scheme (Bruno *et al.*, 2002) which is obtained through pre-order traversal of the document tree is used. Each label consists of three parts: start position end position, level. The encoded version of the XML tree shown in Fig. 1 and 6.

The relative positional information obtained is as follows: let x and x' be two nodes labeled (S, E, L) and (S', E', L'), respectively. Then:

- 'x' is a descendant of x if and only if 'S'>S and E>E'. Thus, the edge between x and x' represents an ancestor-descendant edge
- x' is a child of x if and only if S'>S and E>E' and L' = L+1. Thus, the edge between x and x' represents a parent-child edge

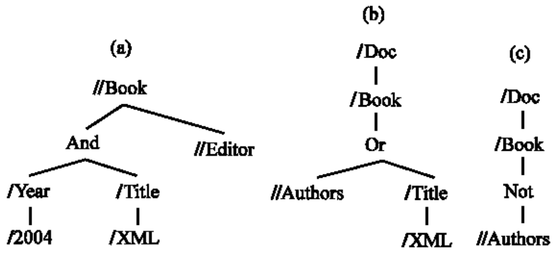


Fig. 5: Representation of XML Boolean query trees

Normalization: The following definition incorporates DNF (Disjunctive Normal Form) from Boolean logic into the context of a B-twig and forms the concept of normalized B-twigs: A normalized B-twig is a query tree that has only four types of nodes: QNodes, NQNodes, ONodes and ANodes that satisfy the following conditions:

- Each OR predicate branch can be mapped to a DNF
- Each NQNode, i.e., the combined form of NOT node with subsequent QNode child must be a leaf
- Each explicit ANode must appear within an OR predicate branch

The normalization procedure consists of repeated application of the following three consecutive steps of transformation:

- NOT-pushdown
- AND-pushdown
- Simplification

Each transformation step is implemented via a set of transformation rules listed below:

- A [NOT [B AND C]] = A [[NOT B] OR [NOT C]]
- A [NOT [B OR C]] = A [[NOT B] AND [NOT C]]
- A [NOT B [C]] = A [NOT B OR B [NOT C]]
- A [[B OR C] AND D] = A [[B AND D] OR [C AND D]]
- A [NOT NOT B] = A [B]

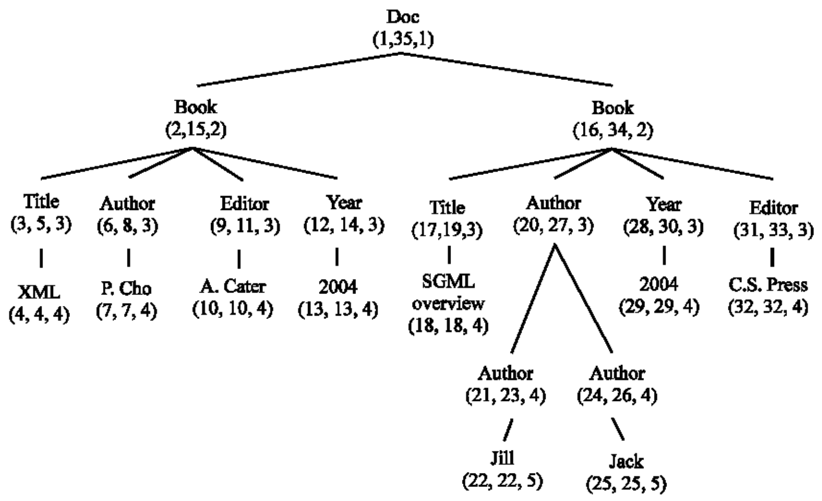


Fig. 6: Encoded XML tree representation

Holistic boolean twig pattern matching: The main algorithm "BTwigmerge" of the second phase called the computing phase of the Boolean twig pattern matching process. The BT wig merge uses the labels to compute the answers to the twig. All the functions are as described in (Bruno *et al*., 2002; Che *et al*., 2012).

BTwigMerge holistic Boolean twig pattern matching algorithm; algorithm BTwigMerge (root):

```
While not end (root) do
q = GetQNode (root)
if q = null then
continue
if not is Root (q) then
clean Stack (SQ parent (Q), Cq)
clean Stack (Sq, Cq);
if is Root (q) or (not empty (SQParent (q))) then
if not is out Leaf (q) then
push (Sq, Cq is Root (q)?-1
top (SQParent (q)))
else
output Path Solutions (Cq)
Cq→advance ()
end while
merge Path Solutions ()
```

GetQNode is an essential subroutine which is called by the main algorithm BT wig Merge to decide the next QNode for processing. The GetQNode guarantees that, the stream head element associated to the returned QNode is part of the final output.

Pseudocode for GetQNode function; function GetQNode (q):

```
if is Leaf(q) then
return q
for each qi ∈ Qchildren(q) do
q0 = GetQNode(qi)
if q0! = qi and is out Node(q0) then
return q0
end for
q max = Get Max Qchild(q)
while Cq→end <Cqmax→start do
Cq→advance()
end while
q min = argminqi {Cqi->start}, qi ∈ Qchildren (q)
while Cq→start < Cqmin→start do
if has Extension(q) and is OutNode(q) then
return q
else
Cq→advance()
end while
if has Extension(qmin) and is OutNode(qmin) then
return q min
else
Cqmin→advance()
if end(q) then
return null
else
return GetQNode(q)
```

Another key subroutine has extension helps in ensuring that only relevant contributing nodes are taken for processing with respect to edge-type (AD or PC) for leaves and for non-leaf nodes, performs testing of further extensions.

Pseudocode for has extension function; function has extension(q):

```
For each qi ∈ children(q) do
if Onode(qi) then
return OnodeTest (Cq, qi)
else if NQNode(qi) then
return NedgeTest(Cq, qi)
else
if is Leaf(qi) then
return edgeTest(Cq, qi)
else
return (edgeTest(Cq, qi) and has Extension(qi))
end for
```

This has extension function in turn calls three other supporting functions, Onode test, n EdgeTest and edgeTest respectively. Onode test function evaluates the OR predicates using OR-blocks mechanism where in a logical formula P(n) recorded in the root structure of the OR-block provides the needed information.

Pseudocode for Onode test function; function Onode test (e, n):

```
For each ni in P(n) do
if is Leaf (ni) and is Qnode (ni)
replace ni by edgeTest (e, ni)
else if is Leaf (ni) and is NQNode (ni)
replace ni by nEdge Test(e, ni)
else
replace ni by (edgeTest (e, ni) and
has Extention (ni))
break
end for
evaluate P(n) and return the result
```

Function n-EdgeTest which is designed for dealing with NQNodes relies on repeated calls to function edgeTest.

Pseudocode for nEdgeTest function
Function nEdgeTest (e, q):

```
while not end (Cq) do
if edgeTest (e, q) == true then
return false
else if Cq→end <e.end then
Cq→advance ()
else
break
end while
return false
```

Function edgeTest checks the region coverage relationship (AD or PC) between two candidate elements. The while loop brings an important optimization-fast skipping non contributing elements in stream.

Pseudocode for Edge test function
Function Edge test (e, q):

```
While not end(Cq) do
if e.start <Cq→start and e.end >Cq→end then
if q.axis == '//' then
return true
else if e.level == Cq→level-1 then
return true
if Cq→end <e.end then
Cq→advance()
else
break
end while
return false
```

The largest threshold value, introduced in (Jiang *et al*., 2004) is computed by a special supporting function called OR block max. This algorithm traverses the structure of an OR block and computes the maximum threshold value to help effectively skip disqualified elements in the parent stream.

Pseudocode for OR block max function
Function OR block max(n):

```
q0 = 0
if is NQNode(n) then
return 0
else if is QNode(n) and is Leaf(n) then
return n
else
if is QNode(n) then
q0 = n
for each ni ∈ children(n) do
qi = ORBlockMax(ni)
end for
if is ONode(n) then
return argminqi {ei.start} for qi
initialized at line 10
else
return argmaxqi {ei.start} for qi initialized at line 10 and line 1
```

Complexity analaysis: Given a twig query Q, the parameters used are:

- |Input| stands for the total size of all the input streams relevant to query Q
- |Output| stands for the total count of the data elements included in all output twig instances produced for query

The I/O cost of BTwigMerge consists of three parts: the I/O cost for accessing all the relevant input stream elements and the I/O cost for dealing with the intermediate path solutions plus the I/O cost for outputting the final twig solutions. Since, in BTwigMerge, the stream cursors are always advanced and never backtracked, the first part of the I/O cost is the total size of all relevant input streams. For the second part, since BTwigMerge is optimal with both AD and PC edges, i.e., it never produces useless intermediate path solutions, the I/O cost of this part is two times (for first output and then input) of the total final output size, i.e., 2* |Output|. And, the third part (for outputting the final results) of course is |Output|. The total I/O cost for BTwigMerge is the sum of the above three parts = |Input|+3* |Output|.

The CPU cost analysis for BtwigMerge is analogous. The CPU cost also consists of three parts. The first part is the time spent on computing the path solutions, the second part is the time spent on dealing with the obtained intermediate path solutions (output, input and merging) and the third part is on outputting the final twig solutions. The main structure of BTwigMerge is a loop that repeats no more than |Input| times which is the total number of elements in all the input streams because noncontributing elements are skipped at line 10, 17 and 22 of GetQNode or by the optimization rendered by the primitive function edgeTest. So, the first part of the CPU cost is linear to the input size. The second part depends on how many intermediate path solutions are produced and how many of them are going to be merged to form the final output twig solutions. As, BTwigMerge does not produce any unused intermediate path solutions (it actually does not push any noncontributing elements onto any stack), the second part of the cost is linear to and solely decided by the output size |Output|. And, the third part of course is also linear to the output size. Added together for the overall CPU cost of BTwigMerge, exactly the same result as that derived for the I/O cost is obtained (cost equations omitted).

The above cost analysis results shows that BTwigMerge has both optimal I/O cost and optimal CPU cost for twigs with both AD and PC edges.

RESULTS AND DISCUSSION

In this study, the answers to various Boolean twigs are shown. The platform of the experiments contains an Intel i5 Core at 2.27 GHz running Windows XP system with 4 GB memory and a 75 GB hard disk. Java SE is the software platform on which these algorithms are implemented and tested. JDK 1.6 is used to access the XML data elements in the data sets and SAX is the adopted XML parser for this study. Eclipse IDE has been used to develop and build the application.

To avoid potential bias of using a single data set, three XML data sets downloaded from the University of Washington XML repository are used for this study. The first data set consisting of 66729 data nodes and maximum depth of 6 contains the course details of a university. The second data set consisting of 150001 data nodes and maximum depth of 3 contains the order details of an online shopping site. The third data set consisting of 476646 data nodes and maximum depth of 8 contains various astronomical data of NASA. Preprocessing is performed beforehand to obtain the region code label for each data element and the input streams needed by all algorithms.

The first data set used is the course data set. Figure 7 shows the result of a Boolean intra query with NOT predicate. It displays the instructors of the courses with La803 as section where a building has not yet been allocated for that particular section. 636 solution paths are computed as output.

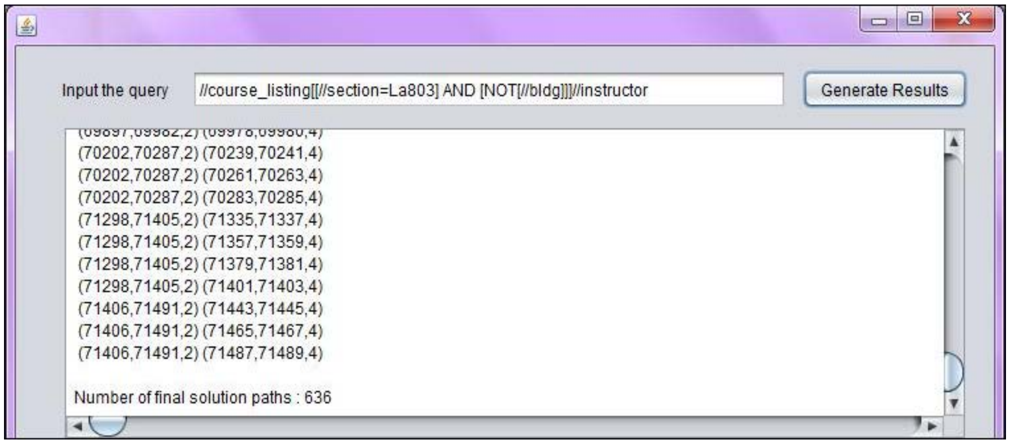
Figure 8 shows the result of a Boolean inter query with NOT predicate. It displays the instructors of the courses with La803 as section where a building has not yet been allocated for any section of that course. The 1130 solution paths are computed as output.

Figure 9 shows the result of a Boolean intra query with AND predicate. It displays the end times of the courses with La801 as section and 12:30 pm as start time for that particular section. The 333 solution paths are computed as output. Figure 10 shows the result of a Boolean inter query with AND predicate. It displays the end times of the courses with La801 as section and 12:30 pm as start time for any section of that course. 1484 solution paths are computed as output.

The second data set used is the order data set. Figure 11 shows the result of a Boolean intra query with OR predicate. It displays the order keys of either urgent priority orders or pending status orders or both. The 3319 solution paths are computed as output.

Figure 12 shows the result of a Boolean inter query with OR predicate. It displays the order keys of either urgent priority orders or pending status orders but not both. 3255 solution paths are computed as output.

The third data set used is the NASA data set. Figure 13 shows the result of a Boolean query with a combination of AND-OR-NOT predicates. It displays the titles which are either published by author with Jackson

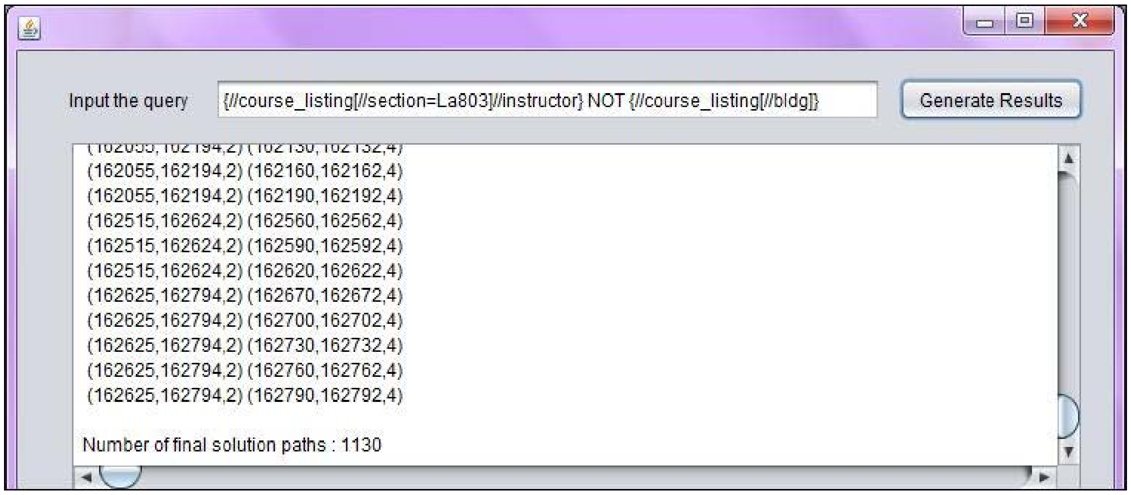


```
//course_listing[[//section=La803] AND [NOT[//bldg]]]//instructor

(69897,69982,2) (69978,69980,4)
(70202,70287,2) (70239,70241,4)
(70202,70287,2) (70261,70263,4)
(70202,70287,2) (70283,70285,4)
(71298,71405,2) (71335,71337,4)
(71298,71405,2) (71357,71359,4)
(71298,71405,2) (71379,71381,4)
(71298,71405,2) (71401,71403,4)
(71406,71491,2) (71443,71445,4)
(71406,71491,2) (71465,71467,4)
(71406,71491,2) (71487,71489,4)

Number of final solution paths : 636
```

Fig. 7: Twig results of Boolean intra query with NOT predicate

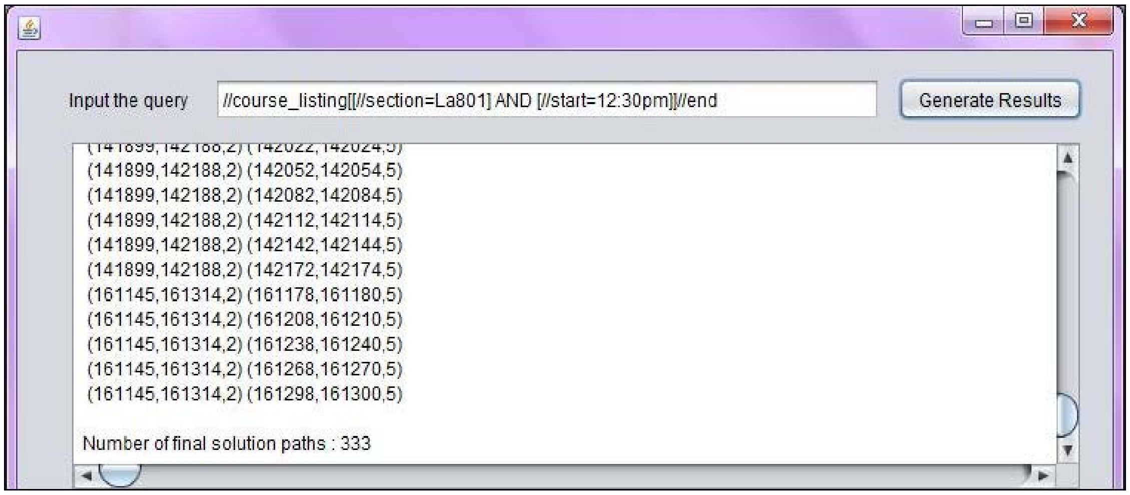


```
{//course_listing[//section=La803]//instructor} NOT {//course_listing[//bldg]}
```

```
(162055,162194,2) (162130,162132,4)
(162055,162194,2) (162160,162162,4)
(162055,162194,2) (162190,162192,4)
(162515,162624,2) (162560,162562,4)
(162515,162624,2) (162590,162592,4)
(162515,162624,2) (162620,162622,4)
(162625,162794,2) (162670,162672,4)
(162625,162794,2) (162700,162702,4)
(162625,162794,2) (162730,162732,4)
(162625,162794,2) (162760,162762,4)
(162625,162794,2) (162790,162792,4)

Number of final solution paths : 1130
```

Fig. 8: Twig results of Boolean inter query with NOT predicate

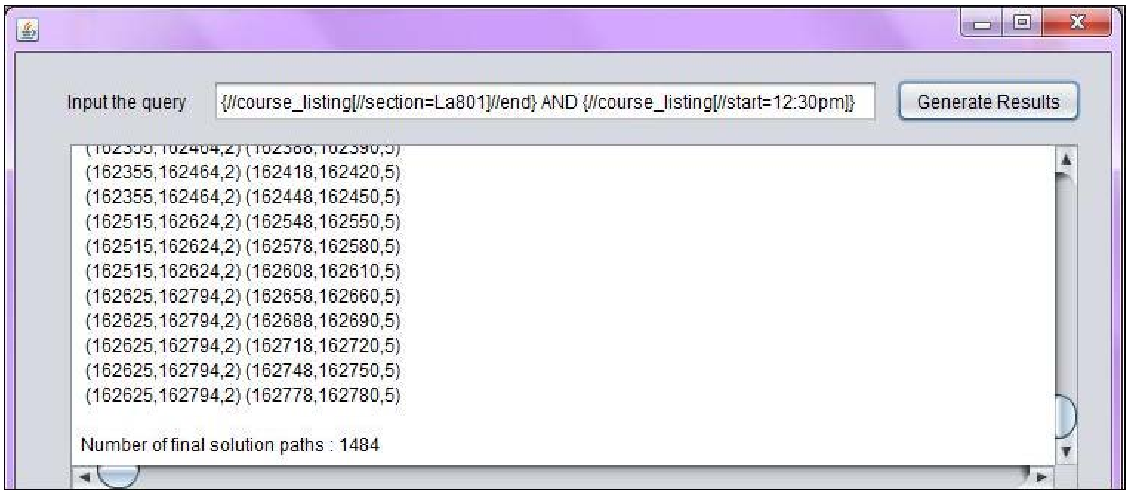


```
//course_listing[[//section=La801] AND [//start=12:30pm]]//end
```

```
(141899,142188,2) (142022,142024,5)
(141899,142188,2) (142052,142054,5)
(141899,142188,2) (142082,142084,5)
(141899,142188,2) (142112,142114,5)
(141899,142188,2) (142142,142144,5)
(141899,142188,2) (142172,142174,5)
(161145,161314,2) (161178,161180,5)
(161145,161314,2) (161208,161210,5)
(161145,161314,2) (161238,161240,5)
(161145,161314,2) (161268,161270,5)
(161145,161314,2) (161298,161300,5)

Number of final solution paths : 333
```

Fig. 9: Twig results of Boolean intra query with AND predicate

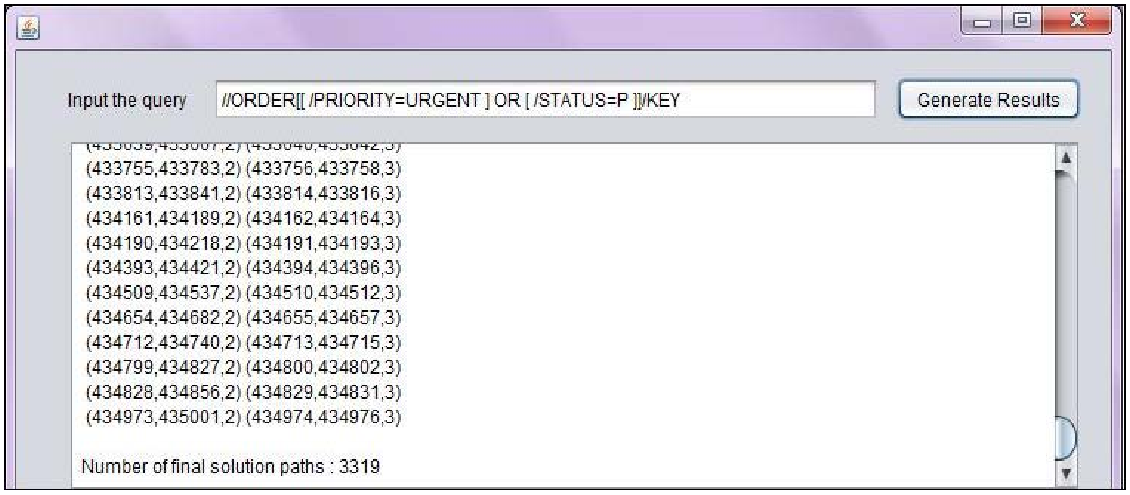


```
{//course_listing[//section=La801]//end} AND {//course_listing[//start=12:30pm]}
```

```
(162355,162464,2) (162388,162390,5)
(162355,162464,2) (162418,162420,5)
(162355,162464,2) (162448,162450,5)
(162515,162624,2) (162548,162550,5)
(162515,162624,2) (162578,162580,5)
(162515,162624,2) (162608,162610,5)
(162625,162794,2) (162658,162660,5)
(162625,162794,2) (162688,162690,5)
(162625,162794,2) (162718,162720,5)
(162625,162794,2) (162748,162750,5)
(162625,162794,2) (162778,162780,5)

Number of final solution paths : 1484
```

Fig. 10: Twig results of Boolean inter query with AND predicate

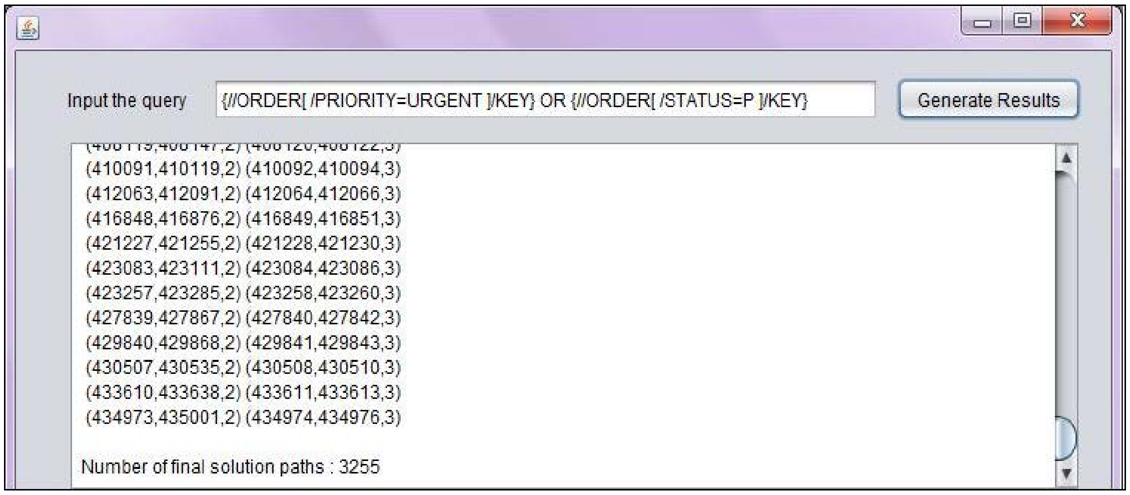


```
//ORDER[[ /PRIORITY=URGENT ] OR [ /STATUS=P ]]/KEY

(433755,433783,2) (433756,433758,3)
(433813,433841,2) (433814,433816,3)
(434161,434189,2) (434162,434164,3)
(434190,434218,2) (434191,434193,3)
(434393,434421,2) (434394,434396,3)
(434509,434537,2) (434510,434512,3)
(434654,434682,2) (434655,434657,3)
(434712,434740,2) (434713,434715,3)
(434799,434827,2) (434800,434802,3)
(434828,434856,2) (434829,434831,3)
(434973,435001,2) (434974,434976,3)

Number of final solution paths : 3319
```

Fig. 11: Twig results of Boolean inter query with OR predicate

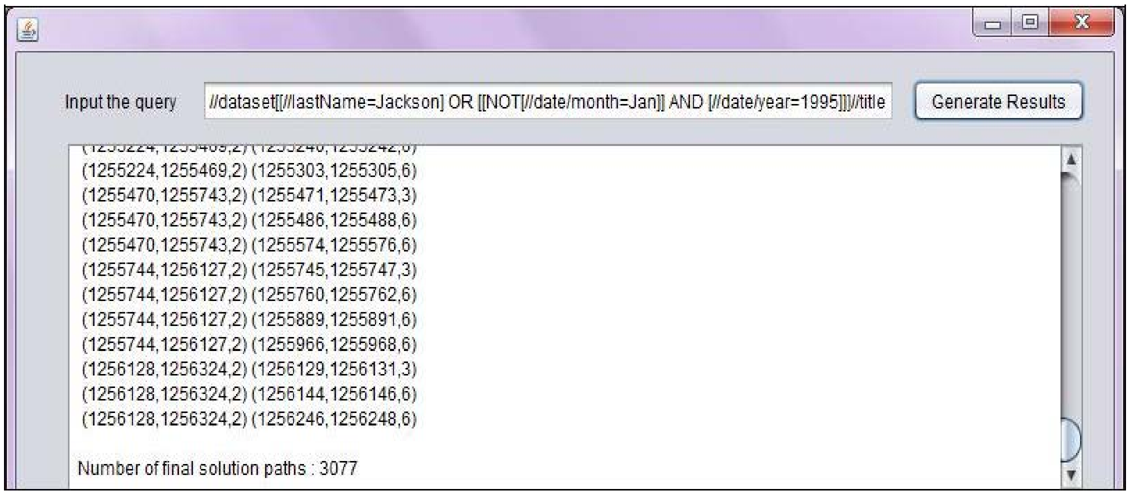


```
{//ORDER[ /PRIORITY=URGENT ]/KEY} OR {//ORDER[ /STATUS=P ]/KEY}

(410091,410119,2) (410092,410094,3)
(412063,412091,2) (412064,412066,3)
(416848,416876,2) (416849,416851,3)
(421227,421255,2) (421228,421230,3)
(423083,423111,2) (423084,423086,3)
(423257,423285,2) (423258,423260,3)
(427839,427867,2) (427840,427842,3)
(429840,429868,2) (429841,429843,3)
(430507,430535,2) (430508,430510,3)
(433610,433638,2) (433611,433613,3)
(434973,435001,2) (434974,434976,3)

Number of final solution paths : 3255
```

Fig. 12: Twig results of Boolean inter query with OR predicate



```
//dataset[[//lastName=Jackson] OR [[NOT[//date/month=Jan]] AND [//date/year=1995]]]//title

(1255224,1255469,2) (1255303,1255305,6)
(1255470,1255743,2) (1255471,1255473,3)
(1255470,1255743,2) (1255486,1255488,6)
(1255470,1255743,2) (1255574,1255576,6)
(1255744,1256127,2) (1255745,1255747,3)
(1255744,1256127,2) (1255760,1255762,6)
(1255744,1256127,2) (1255889,1255891,6)
(1255744,1256127,2) (1255966,1255968,6)
(1256128,1256324,2) (1256129,1256131,3)
(1256128,1256324,2) (1256144,1256146,6)
(1256128,1256324,2) (1256246,1256248,6)

Number of final solution paths : 3077
```

Fig. 13: Twig results of Boolean query with AND-OR-NOT predicates

as last name or published in the year 1995 except the month of January. The 3077 solution paths are computed as output.

CONCLUSION

Holistic twig joins are critical operations for XML queries. The three basic logical predicates and OR and NOT are natural expression mechanisms that people would desire to apply to general XML queries. BTwigMerge, a Boolean holistic twig join algorithm provides an integral solution for efficient and uniform processing of AND-OR-NOT queries in a single algorithmic framework. In this study, two Boolean query formats, intra-query and inter-query, extend the use of B-twigs by giving a different context to the query. The approach supports Boolean twigs or B-twigs, i.e., twigs which support any arbitrary combination of AND/OR/NOT Boolean predicates. In order to reduce the intrinsic complexity in arbitrary B-twigs, B-twig normalization that successfully sorts out the arbitrary combination of the logical predicates in B-twigs has been suggested. A valid procedure to automatically transform input B-twigs into normalized forms has been designed. The normalized B-twigs are then sent to BT-wigMerge that embodies the Boolean holistic twig join strategy. Thus, the proposed approach has been presented and the results have been recorded.

REFERENCES

Bruno, N., D. Srivastava and N. Koudas, 2002. Holistic twig joins: Optimal XML pattern matching. Proceedings of the International Conference on Management of Data, June 3-6, 2002, Madison, Wisconsin, pp: 310-321.

Che, D., T.W. Ling and W.C. Hou, 2012. Holistic boolean-twig pattern matching for efficient XML query processing. IEEE Trans. Knowledge Data Eng., 24: 2008-2024.

Jiang, H., H. Lu and W. Wang, 2004. Efficient processing of twig queries with OR-predicates. Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, June 13-18, 2004, Paris, France, pp: 59-70.

Lu, J., T. Chen and T.W. Ling, 2004. Efficient processing of XML twig patterns with parent child edges: A look-ahead approach. Proceedings of the 13th ACM International Conference on Information and Knowledge Management, November 8-13, 2004, Washington, DC., USA., pp: 533-542.

Yu, T., T.W. Ling and J. Lu, 2006. TwigStackList: A holistic twig join algorithm for twig query with not-predicates on XML data. Proceedings of the 11th International conference on Database Systems for Advanced Applications, April 12-15, 2006, Springer-Verlag, London, pp: 249-263.