AN EFFICIENT AND EXTEND B-TWIG PATTERN FOR XML QUERY PROCESSING

Santhosh Kumar KP[*]

T.Kumesh[**]

Abstract—

Twig pattern matching is a critical operation for XML query processing, and theholistic computing approach has shown superior performance over other methods. Since Bruno et al. introduced the first holistic twig join algorithm, TwigStack, numerous so-called holistic twig join algorithms have been proposed. Yet practical XML queries often require support for more general twig patterns, such as the ones that allow arbitrary occurrences of an arbitrary number of logical connectives (AND, OR, and NOT); such types oftwigs are referred to as B-twigs (i.e., Boolean-Twigs) or AND/OR/NOT-twigs. We have seen interesting work on generalizing theholistic twig join approach to AND/OR-twigs and AND/NOT-twigs, but have not seen any further effort addressing the problem of AND/OR/NOT-Twigs along with XOR twig at the full scale, which therefore forms the main theme of this paper. In this paper, we investigate novel mechanisms for efficient B-twig pattern matching. In particular, we introduce "B-twig normalization" as an important first-step in ourapproach toward eventually conquering the complexity of B-twigs, and then present BTwigMerge the first holistic twig join algorithm designed for B- twigs. Both analytical and experimental results show that BTwigMerge is optimal for B-twig patterns with AD (Ancestor-Descendant) edges and/or PC (Parent-Child) edges.

Index Terms: XOR twig, Query processing, database management, XML data querying, twig join, Boolean twig, logical predicate

[*] PG Student, Department of Computer Science and Engineering, PSN Engineering College, Tirunelveli-627152, India

[**] Assistant Professor of the Department, Department of Computer Science and Engineering, PSN Engineering College, Tirunelveli-627152, India

1.INTRODUCTION

AN XML database stores a collection of data trees. An XML query describes a tree-shaped search pattern, which is often referred to as a twig pattern [5], with additional query conditions (if any) described as predicates on the three nodes. XML queries thus are called tree queries or twig queries. Answering a twig query is essentially to find all matching instances from a database that match the twig pattern implied by the query and satisfy all additional predicates (if any) in the query. A naive way of finding the matches for a twig pattern is to scan the database (usually for many times). A better way uses structural joins [14], [4] in a bulk way to compute the matches for each individual edge, and then “stitch” the matches found for individual edges together to form the answers for the whole twig. This approach typically creates large sets of unused intermediateresults, even when the final result set is pretty small. Yet, a much more efficient approach, called holistic twig join, computes the matches for the whole twig in a holistic way so that irrelevant intermediate results (which need be output and input, and thus are most detrimental to queryperformance) can be avoided. The first holistic twig joinalgorithm, TwigStack, was proposed by Bruno et al. [5] in2002. Since then the “holistic join” approach has been donebroadly extended by numerous followers [6], [8], [7], [9],selects the authors who have papers either titled “TwigJoin” or published in SIGMOD 2006. This query contains bothOR and AND operations. The next query, finds papers that donot have references. This query contains a NOT operation. Atwig 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).

The importance of B-twigs for XML queries is obvious andwell recognized [7], [13]. So far, we only see that Jiang et al. [7]studied the holistic twig join issue for AND/OR-twigs (i.e.,twigs with only AND and OR predicates) and Yu et al. [13] tackled the problem for AND/NOT-twigs (i.e., twigs withonly AND and NOT predicates). There is no integral methodever reported facing the full challenge of holistic B-twigcomputing. The challenge with full B-twigs lies in thearbitrary occurrences of an arbitrary number of AND/OR/NOT predicates in a B-twig (we refer to this challenge as the“double arbitrariness” challenge of B-twigs). This challengemakes programmatic handling of B-twigs in the frameworkof holistic computing extremely hard (if not impossible). Wehave made numerous years of effort on conquering the fullchallenge of B-twigs. We could not easily sort out thecomplication caused by the “double arbitrariness” of Btwigs,and thus could not systematically andprogrammatically solve the problem of B-twigs with anice algorithm. However, our effort has helped us gain in-depthinsight into the challenge of holistic B-twig patterncomputing. We have made numerous years of effort onconquering the full challenge of B-twigs. We could noteasily sort out the complication caused by the “doublearbitrariness” of B-twigs, and thus could notsystematically and programmatically solve the problem ofB-twigs with a nice algorithm. The severity of thischallenge, we believe, there are so many ways holistic joinsolution for B-twigs that has not been developed nearly 9 years after Bruno et al. first proposed the promisingholistic join approach [5] that afterward quickly inspired thesolutions for AND/OR-twigs [7] and AND/NOT-twigs [13]separately proposed by different researchers. From AND/OR-twigs and AND/NOT-twigs to full B-twigs appears to bejust one step, however, as the complexity implied by thedouble arbitrariness blows up, a holistic join approach for fullB-twigs cannot be simply obtained from combining themethods separately designed for AND/OR-twigs andAND/NOT-twigs. Rather, a more creative strategy with more powerful supporting mechanisms must be invented forB-twigs. Solving the challenge of holistic B-twig computinghas both practical and academic significance. From thepractical perspective, this effort helps to mature the promisingholistic twig join approach and can immediately find usein real XML query applications; from the academic side, itsolves an important technical problem and the obtainedresult can be generalized to any data sources incarnating atree data model (while XML is just one use case of the generaltree data model).

We are thus motivated to sort out the complicationinvolved in holistic computing of B-twig pattern matches.In this paper, we present our complete approach, includingthe techniques we developed for systematically solvingholistic B-twig computing. The contributions of our workreported here can be summarized as follows:

- We propose a novel facility, i.e., B-twignormalization, which serves as the first milestone in our approach toward eventually solving the increased complexityof B-twigs.
- We expound a sound method for automatically performing B-twig normalization, which is an important prestep in our overall approach.
- We present BTwigMerge, the first holistic join algorithm ever designed for (normalized) B-twigs, including numerous original supporting mechanisms.

- BTwigMerge performs optimal matching [5] for both AD (Ancestor-Descendent) edges and PC (Parent-Child) edges, while prior algorithms claim optimality only for AD edges.

The remainder of this paper is organized as follows:Section 2 reviews related work. Section 3 sets forth thepreliminaries for the subsequentdiscussion, including datamodel, B-twig representation, and normalization. Section 4 presents our algorithm, BTwigMerge,including its various supporting functions (each implements an important supporting mechanism). Section 5 provides experimental results, demonstrating the superiority of our approach and algorithm.

2 RELATED WORK

Twig pattern matching is a core operation in the XML query processing. Naive navigation (or pointer-chasing), structural joins, and holistic twig joins have all been studied for twig pattern matching. In the following, we review representative works on structural joins and particularly on holistic twig joins.

The first structural join (called containment join) algorithmwas proposed by Zhang et al. [14], which extends thetraditional merge join to multipredicate merge join(MPMGJN). Al-Khalifa et al. [4] later proposed two familiesof structural join algorithms, i.e., tree-merge and stack-basedstructural joins, as primitives forXMLtwig query processing.In 2002, Bruno et al. [5] first proposed the holistic twig joinapproach for XML twig queries in order to overcome thedrawback of structural joins that usually generate large setsof unused intermediate results. Bruno et al. designed the firstholistic twig join algorithm, named TwigStack, which isoptimal for twigs with only AD edges (but not with PCedges). The work of Lu et al. [9] aimed at making up this flawand they presented a new holistic twig join algorithm,TwigStackList, in which a list structure is used to cachelimited elements in order to identify a larger optimal queryclass. Chen et al. [6] studied the relationship betweendifferent data partition strategies and the optimal queryclasses for holistic twig joins. Lu et al. [10] proposed a newlabeling scheme, called extended Dewey, and an interestingalgorithm, named *TJFast*, for efficient processing of XMLtwig patterns. Unlike all previous algorithms based on regionencoding, to answer a twig query, *TJFast* only needs toaccess the labels of the leaf query nodes. The result of Lu et al.
[10] Includes enhanced functionality (can process limitedwildcard), reduced disk access, and increased total queryperformance. The same group [11] also studied efficientprocessing for ordered XML twig patterns using their regionencoding scheme.

In an ordinary twig, the multiple sibling nodes under acommon parent node automatically signify the AND logicrelationship among them, and all previously proposedholistic twig join algorithms already support this impliedAND logic in their implementation schemes. Users wouldtake all the three commonly used logical predicates, AND, OR, and NOT, as granted facilities in formulating their XMLqueries and thus would expect full support from a queryengine for unlimited use of all these predicates in their XMLqueries. Jiang et al. [7] made the first effort towardincorporating support for OR predicates into the holistictwig join approach pioneered by Bruno et al. [5], and Yu et al.[13] made effort for supporting NOT predicates in XML twigqueries. Jiang et al. [7] presented an interesting frameworkfor holistic processing of AND/OR-twigs based on theconcept of OR-block. With resort to OR-blocks, an AND/OR-twig is transformed to an AND-only twig carryingspecial OR-blocks. This work is inspiring to us—we find it ispossible to substantially extend their framework so that theNOT logic can be seamlessly incorporated. Nevertheless,this work is not straightforward, but requires creativereinvention of the "wheels." In order to harness thecomplexity of B-twigs, we resort to B-twig normalization;then based on normalized B-twigs, we are able to extend andadapt the OR-block concept with new supporting mechanismsfor handling the NOT predicates involved in B-twigs.

The recent publication of Xu et al. [12] proposed anotherinteresting algorithm that claims to be able to efficientlycompute the answers to XML queries without holisticallycomputing the twig patterns—the answers obtained containindividual elements corresponding to designated output query nodes. So basically this work does not belong to thecategory of holistic twig join algorithms. But what isinteresting of their work [12] is the proposed path-partitionedelement encoding scheme, which bears efficiency potential andmay be considered in the future for further improving theperformance of holistic B-twig pattern matching.

3 PRELIMINARIES

In this section, we first address the data model issue,B-twig representation and normalization, and then introducethe notations and operations needed in our subsequentdiscussion.

3.1 Data Model

We adopt the general perspective [5] that an XML databaseis a forest of rooted, ordered, and labeled trees, each nodecorresponds to a data element/value, and each edgerepresents an element-sub element or element-value relation.The order among sibling nodes implicitly defines atotal order on the tree nodes. Node labels are important forefficient processing of a twig pattern as properly designednode labels may leave out the necessity of accessing thenode contents during query evaluation. This is especiallytrue with twig pattern matching, which is at the core ofXML query processing. Node labels typically encode theregion information of data elements that reflects the relativepositional relationships among the elements in the sourcedata file. We assume a simple encoding scheme using atriplet region code—(start; end; level)—which is assigned toeach data element in a tree database as a label. Whenmultiple documents are present, the document-id is added tothe labels to differentiate the documents. Region code canbe conveniently obtained through preorder document-treetraversing.

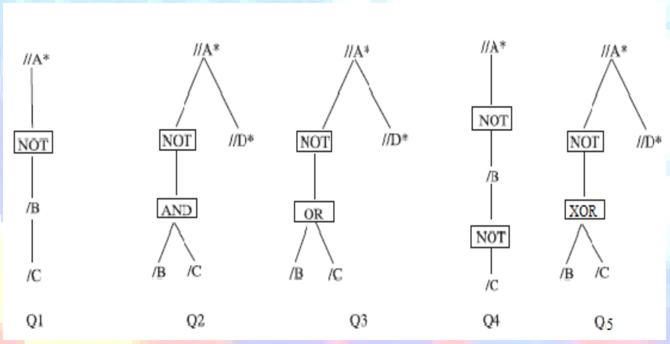
3.2 Tree Representation

Each XML query implies a twig pattern, small, or large. Thesmallest twig may contain just a single node, but a typicaltwig usually comprises a number of nodes. The target ofour investigation is the B-twigs that allow arbitrarycombination of AND, OR, and NOT predicates, of whicheach may have multiple occurrences. Each B-twig mayconsist of two general categories of nodes: ordinary querynodes standing for element types (or tags) and specialconnective nodes denoting logical predicates—AND, OR, andNOT. More specifically, we represent a B-twig using thefollowing specific types of nodes:

- *QNode*. An ordinary query node, associates to an element type (or tag name) in a tree database. For programmatic purposes (as in [7]), a *QNode* records its location step axis “//” or “/” for edge test, and a tag name for node test. Therefore, the content of a *QNode* takes the general format of “/tag” or “//tag.” A non-root *QNode* in a B-twig may be conveniently called a d-child or c-child (of its parent) depending on whether a “//” or a “/” symbol is recorded in the *QNode*’s content (Notice that in the sequel we may not always show the location step axes in ourillustrations when the emphasis is onsomething else).

- *ANode*. An AND predicate node, always takes the text “AND” as its content. It connects two or morechild sub trees through the AND logic.
- *ONode*. An OR predicate node, always takes the text“OR” as its content. It connects two or more childsub trees through the OR logic.
- *NNode*. The NOT predicate node, always takes thetext “NOT” as its content. Functionally, a *NNode*negates the predicate denoted by the sub tree immediatelyunderneath it. A *NNode* is commonlycombined with the node underneath it in the B-twig,forming a composite node. We have the followingthree kinds of composite nodes related to NOT
- *ZNode*. An XOR predicate node, always takes the text “XOR” as its content. It connects two or morechild sub trees through the XOR logic.

Fig. 1. Example twigs involving NOT



-*NQNode*: the combined form of a NOT nodewith a subsequent *QNode* child (such combinationjust causes the representation of a B-twigmore compact, and does not affect the semanticsor interpretation of the twig pattern). Forexample, in the query Q1 (shown in Fig. 1), theNOT and the subsequent child *QNode* “/B” canbe combined and replaced by a single *NQNode*with content “□/B.”

- *NANode*. The combined form of a NOT nodewith its sole *ANode* child.

- *NONode*. The combined form of a NOT nodewith its sole *ONode* child.

-*NZNode*. The combined form of a NOT node with its sole *ZNode* child.

We could also have the fourth type of composite nodethat represents a NOT node combined with another (child)NOT node (i.e., a double negation node that could benamed *NNNode*). As the net effect of double negations isthe same as no negation at all, double negation nodes are notactually used in our representation for B-twigs. From nowon, we generally refer to *QNodes*and *NQNodes* as querynodes, and other (plain or composite connectives) nodes in aB-twig as no query nodes.

With the above mechanisms introduced, our representationscheme for B-twigs is apparently a superset of what canbe represented by the scheme adopted by Jiang et al. [7] for thesimpler AND/OR-twigs. ConsideringNOTas a new elementadded to B-twigs (comparing to the AND/OR-twigs studiedin [7]), we next illustrate the four typical cases that the NOTpredicates may appear in an XML twig query. The followingfour queries exemplify these four representative cases:

Q1. A[NOT/B/C]

Q2. A[NOT (/B AND/C)]//D

Q3. A[NOT (/B OR/C)]//D

Q4. A[NOT/B[NOT/C]]

Q5. A[NOT[/B XOR/C]]

3.3 Edge test Algorithm

The main structure of function *edgeTest* (andfunction *nEdgeTest*as well) is a while loop, which at the first glimpse appears unnecessary, but (at lines 7 and 8, see Fig.2) brings an important optimization—fast skipping noncontributingelements in stream Tq until the cursor moves over therange of the parent element e. (This “fast skipping” has theeffect of instant performance optimization, otherwise thosenoncontributing elements will stay in their streams causingextra iterations and consuming extra CPU time.). The location step axis is obtained from the content of the childquery node. The implementation of function *nEdgeTest* relies on repeatedcalls to function *edgeTest* (see Fig.3). The implementation offunction ONodeTest (see Fig. 4) almost straightforwardlyfollows Definition. It is based on *edgeTest* ,*nEdgeTest*,and yet another function, *hasExtension*, that realizes.

FUNCTION edgeTest(e,q)

```
1:  while not $end$(Cq) do
2:    if $e.start$<Cq →start and $e.end$>Cq→ $end$
then
3:    if $q.axis$==’//’ then
4:    return true
5:    else $if\ e.level == Cq \rightarrow level\ \text{-}1$ then
6:      return true
7:    if $Cq\rightarrow end<e.end$ then
8:    $Cq\rightarrow advance()$
9:    else
10:    break
11:  end while
12:  return false
```

Fig.2. Function *edgeTest*.

Holistic twig joins typically disallow backtracking ofstream cursors to guarantee linear time complexity. Noticethat the evaluation of a NOT predicate involved in a B-twigrequires disproving all elements in the negated stream(associated to the query node negated by the NOTpredicate). This seems to imply that we need to scan tothe very end of the negated stream to disprove all elements,and then backtrack the stream cursor to get ready forevaluating the subsequent parent elements. In fact, suchbacktracking can be avoided.

FUNCTION nEdgeTest(e,q)

```
1:   while not end(Cq) do
2:   if $edgeTest(e,q)$ = = true then
3:             return false
4:   else if $Cq\rightarrow end<e.end$ then
5:   $Cq\rightarrow advance()$
6:     else
7:       break
8:  end while
9:  return false
```

Fig.3. Function *nEdgeTest*

Let's exemplify this: assumingwe are processing the B-twig sub expression X NOT Y, andelements x_i and y_j are currently under the cursor of theirrespective streams, T_X and T_Y, there are two cases toconsider now: 1) y_j happens to fall within the region of xi,this immediately disproves x_i and the evaluation immediatelyreturns; 2) y_j is not in the range of xi, this leads to two possible subcases: a) y_j is ahead of x_i—then we just advancecursor C_Y and start the next iteration to evaluate with thenext element following y_j; b) y_j falls behind x_i, this isenough to qualify x_i, since all subsequent elements after y_j(if any) can only be farther away from the coverage of $x_i$due to the sortends of the elements in stream T_Y. In allcases, advancing of the stream cursor $T_Y$never goes beyondthe range covered by x_i, and backtracking is never needed.The code in Fig.4 embodies the idea discussed above.

FUNCTION ONodeTest(e,n)

```
1: for each $n_i$ in P(n) do
2: if *isLeaf($n_i$)* and *isQNode($n_i$)*
3:          replace ni by *edgeTest(e, $n_i$)*
4: else if *isLeaf($n_i$)* and *isNQNode($n_i$)*
5:          replace
   $n_i$ by(*edgeTest(e, $n_i$)* and*hasExtension($n_i$)*)
6:  else /* $n_i$ is a non leaf*QNode*
7:          replace ni by *edgeTest(e, $n_i$)*
8: end for
9:  evaluate P (n) and return the result
```

Fig.4. Function *ONodeTest*

4 AHOLISTIC B-TWIG JOIN ALGORITHM

With the supporting mechanisms set forth in the precedingsections, we now present our novel holistic B-twig joinalgorithm, *BTwigMerge*, in this section.

4.1 BTwigMerge: The Main Algorithm

The structure of our main algorithm, *BTwigMerge*, asshown in Fig.5, is not much different from most otherholistic twig join algorithms [5], [6], [8], [7], [9], [10], [11],[13]. It is a merge-based, two-phase algorithm. However, aswe confront a different, more complex problem of B-twigsthat was not considered by previous holistic algorithms, theprocessing strategy of our *BTwigMerge* must be accordingly different. The difference mainly lies in the key supporting function, *GetQNode*(detailed in the next
Section). In addition to feeding the main algorithm the nextquery node to be processed, our *GetQNode* functionthoroughly investigates the candidacy of the elements inthe input streams and guarantees that for the next *QNode*returned to the main algorithm the current element in theassociated stream is fully qualified, i.e., the element satisfiesall relevant criteria (including all predicates and edge tests).Functionally, algorithm *GTwigMerge* [7] is the closest toour *BTwigMerge*, but at the main algorithm level,*BTwigMerge* is more concise: with each valid query node qreturned by *GetQNode* (the validness is checked at line 3, seeFig.5), *BTwigMerge* cleans up relevant stacks (lines 6 and7), moves the element associated to q from stream to stack if itis not an output leaf (at lines 8 to 10), otherwise (q is an outputleaf) outputs the path solutions currently on the stacks(lines 9 and 10). It is worth to point out that *BTwigMerge* doesnot explicitly process OR and any other predicates at themain algorithm level (differing from *GTwigMerge* [7]).Instead, all critical processing logics are encapsulated inthe key supporting function, *GetQNode*, and other lowerlevel supporting functions. *GetQNode* also checks whethereach involved PC or AD edge is, respectively, satisfied by thestream head element associated to q that is to be returned tothe main algorithm as the next *QNode* for processing.Therefore, our *BTwigMerge* achieves matching optimalitynot only with AD edges but also with PC edges (this is instrong contrast to all previous holistic algorithms including*TwigStack* [5] and *GTwigMerge* [7], etc.).

```
FUNCTION ORBlockMax(n)
    1: $q_0$=0 /* refers to QNode */
    2: if  isNQNode(n) then
```

```
3   return 0
4: else if isNQNode(n) and isLeaf(n) then
5:  return n
6: else
7:      if isQNode(n) then
8:              $q_0$= n
9:       for each $n_i \in$ children (n) do
10:             $q_i$ = ORBlockMax($n_i$)
11:      end for
12:     if isONode(n) then
13:             return $argmin_{qi}\{_{ei},start\}$
14:     else
15:             return $argmax_{qi}\{_{ei},start\}$
```

Fig.6. Function *ORBlockMax.*

Some important features of *BTwigMerge* are highlightedas follows:

1. *BTwigMerge* receives (from *GetQNode*) either avalid output *QNode* q or an invalid *QNode*, denotedby null. An invalid *QNode*is typically generated by*GetQNode* when a non-top level recursive call intothis function fails to find a *QNode* associated with afully qualified element. But since noncontributingelements encountered during this process have beenskipped, the main algorithm quick jumps to its nextiteration (at line 4) to start a new call to *GetQNode*for getting the next valid *QNode*.
2. No stacks are allocated for no output*QNodes*, norfor any output leaves (*QNodes*) since the contributingelements corresponding to an output leaf can bedirectly grabbed from the associated stream foroutput.
3. *GetQNode* performs specific edge test (PC or AD),which renders both I/O and CPU optimality for bothAD and PC edges involved in a B-twig.
4. "Stack cleaning" is needed in *BTwigMerge* solelybecause each time after outputting path solutions,some elements on the stacks may become irrelevantfor future path solutions and must be cleaned out.(In most prior algorithms such as *TwigStack*, stackcleaning is required

to get rid of those noncontributingelements that may have been tentatively addedto the stacks but are actually noncontributing.)

5. *BTwigMerge* does not explicitly (at the mainalgorithm level) deal with any AND/OR/NOT predicates,nor with any no output*QNodes*.

4.2 GetQNode: The Key Supporting Algorithm

GetQNode is an essential subroutine which is called by themain algorithm BTwigMerge to decide the next *QNode* forprocessing. It is *GetQNode* that guarantees that the streamhead element associated to the returned *QNode* is part ofthe final output since all the relevant predicates (if any) arethoroughly checked by *GetQNode* or its lower levelprimitive subroutines such as *edgeTest*, *nEdgeTest*,*ONodeTest*, and *hasExtension*, etc.

While feeding BTwigMerge with the next *QNode* to beprocessed, some elements on the stream under considerationmay be found noncontributing to the final answer andthus should be skipped right away. The term, largestthreshold value, introduced by Jiang et al. [7] refers to thestart label of a sub element e_{max} of another element, say, esuch that e_{max} maximizes the start label among all theoffspring elements of e. Such a threshold value can be usedto skip e and all its successors if their end label are smallerthan this threshold value. It still makes sense to carry outthis type of optimization for B-twig join, but we need toredefine the mechanism to fit the particular need of B-twigs.

```
FUNCTION GetQNode(q)
  1:  if $isLeaf(q)$ then
  2:    return q
  3:  for each $q_i \in children(q)$ do
  4:    $q_0 = GetQNode(q_i)$
  5:        if $q_0 \neq q_i$ and $isOutNode(q_0)$ then
  6:        return $q_0$
  7:  end for
  8:   $q_{max} = getMaxQChild(q)$
  9:   while $Cq \rightarrow start < Cqmax \rightarrow start$ do
  10:  $Cq \rightarrow advance()$
  11:  end while
  12: $q_{min} = argmin_{qi}\{ Cq \rightarrow start\}$, $q_i \in$ children (q)
  13:  while $Cq \rightarrow start < Cqmin \rightarrow start$ do
```

```
14:  if hasExtension(q) and isOutNode(q) then
15:          return q
16:  else
17:          Cq→advance()
18:  end while
19:  if hasExtension(qmin) and isOutNode(qmin)
20:  return qmin
21:  else
22:  Cqmin→advance()
23:  if end(q) then
24:  return null
25:  else
26:  return GetQNode(q)
```

Fig.7. Function *GetQNode*

The largest threshold value is computed by a specialsupporting function, called *ORBlockMax*(n), in [7]. Weextend this function for our purpose as shown in Fig.6,which conforms to our revised notion for OR-blocks.

Understanding the structural features of OR-blocks innormalized B-twigs is the key to understanding how our*ORBlockMax* function works. This algorithm traverses thestructure of an OR-block and computes the maximumthreshold value to help effectively skip disqualified elementsin the parent stream. Line 1 initializes the variable q0to a special (imaginary) query node, denoted by 0, which isalways associated to a special (imaginary) element identifiedby the region code (0; 0; 0). When the input node is an*NQNode*, line 3 returns this special query node 0(associated to the imaginary element (0; 0; 0). Variable q0is reinitialized at line 8 to n, and is used at line 16 whenchoosing the qmax from all the *QNodes* qi under considerationsuch that qmax gives the maximal start value. At line 13,function argmin$_{qi}$(e$_{i.}$start) selects q$_{min}$ from all the*QNodes* qi under consideration such that q$_{min}$ has theminimal start value. Notice that at this point (line 13), theimaginary element with region code (0, 0, 0) is excluded because all *NQNodes* are irrelevant to the purpose offunction *ORBlockMax*—i.e., to help skip disqualifiedelements in the parent stream.The implementation of function *GetQNode* is shown inFig.7. The *QNode*qx returned by *GetQNode*(q) can be oneof the following two cases:

FUNCTION getMaxQChild(q)

```
1:  for each $n_i \in$ children (q) do
2:     if isQNode($n_i$)
3:    $q_i = n_i$
4:     else if isNQNode(ni) then
5:    $q_i = 0$
6:     else
7:    $q_i$ = ORBlockMax($n_i$)
8:  end for
9:  return argmax$_{qi}$\{ $Cq \rightarrow start$\}, for $q_i$
```

Fig.8. Function *getMaxQChild*.

1) q_x= null (here null denotes aninvalid query node), signifying to the main algorithm toimmediately start another call to *GetQNode* for quicklygetting the next valid *QNode* if the streams are notexhausted yet; 2) q_x is a valid output *QNode*—this is thedominating case, similarly handled as in all other holistictwig join algorithms. Comparing with *GTwigMerge* [7], themost related holistic join algorithm to BTwigMerge, thestructure of our main algorithm is more succinct:we pushed all important tests—including AD and PC edgetests, and tests on any AND/OR/NOT predicate—all downto the core subroutine, *GetQNode*, or its lower levelprimitive supporting functions. The advantage is early skipping of disqualified elements in streams, leading toimproved algorithm performance.

In subroutine *GetQNode(q)*, the information provided bygetMaxQChild(q) (line 8 in Fig.7) is used to skipdisqualified elements in stream Tq. Unlike its counterpartin *GTwigMerge* [7], our *getMaxQChild*(q) (see Fig.8)considers *NQNodes* in addition which do not exist in thesimpler AND/OR-twigs that *GTwigMerge* was designed for.

4.3 Cost Analysis of BTwigMerge

We now analyze the I/O and CPU cost of our algorithmBTwigMerge. For ease of presentation, given B-twig query Q,we first introduce the following parameters:

- [*QNodes*] is the total number of *QNodes* in Q.
- [*NQNodes*] is the total number of *NQNodes* in Q.
- Query size Qj = [*QNodes*] + [*NQNodes*]. Noticehere we do not count other logical predicate nodestoward the query size.

- [Input] stands for the total size of all the inputstreams relevant to query Q.
- [List] stands for the average stream length.
- [Output] stands for the total count of the dataelements included in all output B-twig instancesproduced for query Q.

In terms of the set of twig patterns that can be processed,*BTwigMerge* is a "superset" of *GTwigMerge* and*GTwigMerge* is a "superset" of TwigStack. At the main algorithm level, the three algorithms share great similarity.The cost analysis methods are also similar. So, in thefollowing, we only provide a compact analysis for the I/Oand CPU cost of *BTwigMerge*.

The I/O cost of *BTwigMerge* consists of three parts: theI/O cost for accessing all the relevant input stream elementsand the I/O cost for dealing with the intermediate pathsolutions plus the I/O cost for outputting the final twigsolutions. Since in *BTwigMerge*, we always advance thestream cursors and never backtrack, the first part of the I/Ocost is the total size of all relevant input streams. For thesecond part, since *BTwigMerge* is optimal with both ADand PC edges—i.e., it never produces useless intermediatepath solutions, the I/O cost of this part is two times (for firstoutput and then input) of the total final output size, i.e.,2· [Output]. And the third part (for outputting the finalresults), of course, is [Output]. All together, the total I/Ocost for *BTwigMerge* is the sum of the above three parts.We therefore have the following equations regarding the I/O cost of *BTwigMerge*:

I/Ocost=([QNodes]+[NQNodes]+[ZQNodes])·

[List]+3· [Output]

= [Q] · [List]+3 · [Output]

= [Input] +3 · [Output]:

The CPU cost analysis for *BTwigMerge* is analogous. TheCPU cost also consists of three parts. The first part is the timespent on computing the path solutions, the second part is thetime spent on dealing with the obtained intermediate pathsolutions (output, input, and merging), and the third part ison 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 streamsbecause noncontributing elements are skipped at line 10, 17,and 22 of *GetQNode* (see Fig. 13) or by the optimizationrendered by the two primitive functions, *edgeTest* and*nEdgeTest* (see Figs. 8 and 9, respectively). So the first part ofthe CPU cost is linear to the input size. The second partdepends on how many intermediate path solutions areproduced and how many of them are going to be merged toform the final output twig solutions. As BTwigMerge doesnot produce any unused intermediate path solutions (itactually does not push any noncontributing elements ontoany stack), the second part of the cost is linear to and solelydecided by the output size [Output]. And the third part ofcourse is also linear to the output size. Added together, forthe overall CPU cost of BTwigMerge, we have exactly thesame result as we have for the I/O cost (cost equationsomitted). It is worth to point out that query size [Q] in CPUcost is counted slightly differently from that in I/O cost: forthe former, [Q] counts the duplicated query nodes caused bynormalization, but for I/O cost, it does not becauseduplicatequery nodes do not cause extra physical I/O.The above cost analysis results shows that ourBTwigMerge has both optimal I/O cost and optimal CPUcost for normalized B-twigs with both AD and PC edges.Our experimental study provides empirical evidences tofurther support this conclusion.

5 EXPERIMENTS

In this section, we present the experiment results. As ourBTwigMerge is the only algorithm of its kind—designed forholistic B-twig pattern matching, there does not exist a realcompetitor to compare with. In this case, one plausiblebaseline to compare with is a decomposition-basedapproach. A decomposition-based approach first splits an input B-twig at everypredicate node into a series of subtwigs, then separatelycomputes the partial solutions to the subtwigs, and finallycombines the obtained partial solutions to form the wholesolutions to the original B-twig. Such a decompositionbasedapproach suffers severe performance disadvantagethat Jiang et al. [7] had empirically proven with a subclassof B-twigs years ago. For more general B-twigs, the problemof decomposition-based approach can only become worse.So we have no intention to empirically reprove theconclusion of Jiang et al. [7] at the scale of full B-twigs,instead, we comparatively study the performance of ouralgorithm and other related algorithms on various commonsubclasses of B-twigs.

As the first holistic twig join algorithm, *TwigStack* [5] isdesigned for simple AND-only twigs. In terms of thecategories of twigs being processed, *GTwigMerge* [7] generalizes*TwigStack* and is a superset of TwigStack—capablefor AND/OR-twigs; *TwigStackList*: [13] also generalizes*TwigStack* but from a different aspect and thus is a superset of*TwigStack* [5] as well—capable for AND/NOT-twigs; our*BTwigMerge* significantly extends the approach embodied in*GTwigMerge* and becomes a superset of both *GTwigMerge*and *TwigStackList*:—capable for full B-twigs, i.e., AND/OR/NOT-twigs. The theme of our experimental study thus isset on comparing BTwigMerge, respectively, with thesepredecessor algorithms with regard to a common subset oftwig queries that they are all (or both) capable of dealing with.

5.1 Experimental Setup

Before proceeding to the details of our experiment study,we first address a few related issues about this study.Platform setup. The platform of our experimentscontains an Intel Core 2 DUO 2.2 GHz running WindowsXP System with 4 GB memory and a 75 GB hard disk. JavaSE is the software platform on which these algorithms areimplemented and tested. The various data sets used for thisstudy are kept as external files on the hard disk.Convenientplatform, JUnit 1.4 was used for concise timingof these algorithms on test queries.

Data preparation. To avoid potential bias of using asingle data set, we choose three popular XML data sets forthis study. The first data set is an XMark data set [3] storedin a single XML file. This data set takes roughly 100 MB,containing about 100 thousands elements (or nodes). Thesecond data set is a generated one by Stylus XML Generator[1] using a given XML Schema. Stylus XML Generatorallows users to specify the expected structure and size ofasa XML data via separate XML Schema files. For thispurpose, we carefully designed an XML schema with variedtree structures to avoid biased results.

6. SUMMARY

Holistic twig joins are critical operations for XML queries.The three basic logical predicates, AND, OR, and NOT, arenatural expression mechanisms that people would desire toapply to general XML queries. However, all previouslyproposed holistic twig join algorithms failed to provide anintegral solution for efficient and uniform processing of Btwigqueries (with arbitrary combination of these logicalpredicates) in a single algorithmic framework. Consequently,given a general B-twig query, all prior holisticalgorithms become inapplicable and useless. In this paper,we presented a novel approach for holistic computing of Btwigpatterns and described an original algorithm, called*BTwigMerge*, which is the first of its kind—holisticcomputing of a more general class of twig patternsrepresented by B-twigs. The second distinctive feature of*BTwigMerge*is that it gracefully extends the I/O and CPUoptimality to twigs with PC edges as well.

In order to reduce the intrinsic complexity in arbitrary Btwigs,we proposed B-twig normalization that successfullysorts out the arbitrary combination of the logical predicatesin B-twigs. We designed a valid procedure to automaticallytransform input B-twigs into normalized forms. Thenormalized B-twigs are then sent to BTwigMergethatembodies our holistic twig join strategy and containsnumerous novel supporting mechanisms.

We have done analytical and experimental study withregard to the validity and performance of our approach andits accompanying algorithms, and concluded that ourBTwigMergeis so far the most powerful and most efficientholistic twig join algorithm—the sole one designed for Btwigs,with optimal I/O and optimal CPU on twigs witharbitrary AD and/or PC edges. As future work, thefollowing is on our agenda: Implementation of XOR andefficiency increase by Indexing method.

REFERENCES

[1] Stylus Studio XML Generator, http://www.stylusstudio.com/xml_generator.html,2012.

[2] Univ. of Washington XML Repository, http://www.cs.washington.edu/research/xmldatasets/, 2012.

[3] XMark? An XML Benchmark Project, http://www.xmlbenchmark.org/,2012.

[4] S. Al-Khalifa et al., "Structural Joins: A Primitivefor Efficient XMLQuery Pattern Matching," Proc.18th Int‟ l Conf. Data Eng. Conf.(ICDE‟ 02), pp. 141-152, 2002.

[5] N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Joins:Optimal XML Pattern Matching," Proc. ACM SIGMOD Int‟ l Conf.Management of Data (SIGMOD‟ 02), pp. 310-321,June 2002.

[6] T. Chen, J. Lu, and T.W. Ling, "On Boosting Holism in XML TwigPattern Matching Using Structural Indexing Techniques," Proc.ACM SIGMOD Int‟ l Conf. Management of Data (SIGMOD‟ 05),pp. 455-466, June 2005.

[7] H. Jiang, H. Lu, and W. Wang, "Efficient Processing of TwigQueries with OR-Predicates," Proc. ACM SIGMOD Int‟ l Conf.Management of Data (SIGMOD‟ 04), pp. 59-70, 2004.

[8] H. Jiang, W. Wang, H. Lu, and J.X. Yu, "Holistic Twig Joins onIndexed XML Documents," Proc. 29th Int‟ l Conf. Very Large DataBases (VLDB‟ 03), pp. 273-284, Sept. 2003.

[9] J. Lu, T. Chen, and T.W. Ling, "Efficient Processing of XML TwigPatterns with Parent Child Edges: A Look-ahead Approach," Proc.13th ACM Int‟ l Conf. Information and Knowledge Management(CIKM‟ 04), pp. 533-542, Nov. 2004.

[10] J. Lu, T.W. Ling, C.-Y. Chan, and T. Chen, "From Region Encodingto Extended Dewey: On Efficient Processing of XML Twig PatternMatching," Proc. 31st Int‟ l Conf. Very Large Data Bases (VLDB‟ 05),pp. 193-204, Aug. 2005.

[11] J. Lu et al., "Efficient Processing of Ordered XML Twig Pattern,"Proc. 16th Int‟ l Conf. Database and Expert Systems Applications(DEXA‟ 05), pp. 300-309, 2005.

[12] X. Xu, Y. Feng, and F. Wang, "Efficient Processing of XML TwigQueries with All Predicates," Proc. IEEE/ACIS Int‟ l Conf. Computerand Information Science (ICIS ‟ 09), pp. 457-462, June 2009.

[13] T. Yu, T.W. Ling, and J. Lu, "twigstacklist: A Holistic Twig JoinAlgorithm for Twig Query with Not-Predicates on XML Data,"Proc. 11th Int‟ l Conf. Database Systems for Advanced Applications
(DASFAA‟ 06), pp. 249-263, 200

[14]XML Repositoryhttp://ghr.nlm.nih.gov/search