Faster Twig Pattern Matching Using Extended Dewey ID*

Chung Keung Poon and Leo Yuen

Department of Computer Science
City University of Hong Kong
{ckpoon, leo}@cs.cityu.edu.hk

Abstract. Finding all the occurrences of a twig pattern in an XML database is a core operation for efficient evaluation of XML queries. Recently, Lu *et al.* [7] proposed the TJFast algorithm that uses the *extended Dewey* labelling scheme and reported better performance compared with other *state-of-the-art* holistic twig join algorithms, both in terms of number of elements scanned and stored during the computation. In this paper, we designed an enhancement to further exploit the power of the *extended Dewey ID*. This reduces the CPU cost and also favors indexed inputs. Our algorithm can be shown analytically as efficient as TJFast in terms of worst case I/O, and experimentally performs significantly better.

1 Introduction

An XML document can be naturally modelled as a tree in which nodes represent XML elements while edges represent nesting between elements. A query in many important XML query languages such as XPath and XQuery can also be modelled as a tree in which nodes represent node tests in location steps while edges represent the structural relationships between nodes connected by the edges. Finding all the occurrences of such tree pattern (a.k.a twig pattern) in an XML document is a core operation for efficient evaluation of XML queries.

An early approach for the problem proceeds by breaking down the twig pattern into edges representing the required binary structural relationships (*e.g.*, parent-child, ancestor-descendant), then applying structural joins to match the binary relationships against the XML document and finally stitching together these basic matches. With clever labelling schemes (*e.g.* region or prefix coding) for the XML documents, the structural relationships between two nodes can be determined using only their labels without actually traversing the document tree. However, the problem of such approach is that the intermediate results can become very large, independent of the input and output sizes.

Other algorithms such as ViST [10] and PRIX [8] work by serializing the twig pattern and the XML document into strings and then finding common subsequences. However, time-consuming postprocessing is needed to remove false hits and false dismissals from the subsequence matchings.

* This research was fully supported by a grant from the Research Grants Council of the Hong Kong SAR, China [Project No. 9040906 (RGC Ref. No. CityU 1164/04E)].

S. Bressan, J. Küng, and R. Wagner (Eds.): DEXA 2006, LNCS 4080, pp. 297–306, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Another line of development started by Bruno *et al.* [1] is to recursively build up the matchings following the structure of the twig pattern. Their algorithm, TwigStack, is provably optimal for twig queries with only ancestor-descendant relationship under their specific I/O model. Jiang *et al.* [4] modified and enhanced TwigStack to TSGeneric$^+$ to favor indexed input data stream and to reduce the query time by avoiding certain useless recursive calls. Other algorithms along this line includes the GTwigMerge algorithm of Jiang *et al.* [3] that deals with twig pattern with OR-predicates and the TwigStackList algorithm of Lu *et al.* [6] which is a look-ahead approach to better handle query twigs with parent-child edge. It is proved in [2] that no algorithm for twig queries can read the input stream only once under the model of Bruno *et al.* when the twig contains parent-child edges.

All these stack-based algorithms rely on a labelling scheme called *region encoding* which allows for checking of ancestor-descendant relationship using the labels of two nodes only. Recently, Lu *et al.* [7] proposed the *extended Dewey* labelling scheme in which one can derive the names of all the elements on the path from the root to an element given the *extended Dewey ID* of this element. This unique feature is essential to their algorithm, TJFast, that requires scanning, by far, the least number of input elements and storing the smallest amount of intermediate results among all holistic twig join algorithms under the same I/O model.

In this paper, we observe that TJFast has not yet fully utilized the power of the *extended Dewey* labelling. By a small twist to TJFast, we obtain an improved algorithm which we called TJFaster. Compared with TJFast, our algorithm is more effective in skipping elements and hence avoids more useless recursive calls. It also replaces certain root-leaf path matchings by simpler label comparisons. Further, it fits well on indexed input. We perform a comprehensive experiment to demonstrate the benefits of our algorithm over the previous one.

The rest of the paper is organized as follow. After stating the problem, model and some notations in the next section, the *extended Dewey ID* is given in the section 3. This is followed by details of our enhancement and the experimental evaluation in section 4 and 5 respectively. The paper is then concluded in section 6.

2 Problem Statement, Model and Notations

In the rest of this paper, "node" refers to a tree node in the query twig pattern while "element" refers to an element in the XML document. For any node v in the query tree, we let P_v be the path from the root to v; and Q_v be the query subtree rooted at v *together* with P_v. The path P_e for element e in the document tree is similarly defined.

We say that Q_v has a *matching* if there is a mapping of Q_v to the document tree that preserves the node types and structural relations of edges in Q_v. A node v *matches* an element e if Q_v has a matching in which node v is mapped to e. Let v_{tb} be the *top branching node* in the query tree, *i.e.*, the branching node that is an ancestor of all branching nodes in the query tree. Thus, our problem is to find all the matchings of $Q_{v_{tb}}$.

A matching of Q with n nodes $(v_1, \cdots, v_n)$ can be represented as an n-tuple $(d_1, \cdots, d_n)$ where for $1 \leq i \leq n$, node v_i is mapped to element d_i in the XML document in the matching.

Following the model employed in Ref. [1,7], we assume that each leaf f in the query tree is associated with a stream T_f of all the elements that match the node type of f, sorted in ascending lexicographical order of their labels. A stream T supports the $\mathsf{get}(T)$ function that reads the "current" element of the stream, the $\mathsf{advance}(T)$ function that moves the stream cursor to the next element and the $\mathsf{eof}(T)$ function that tells if T reaches its end. Initially, the stream cursor points to the first element of T.

To describe our algorithms, we assume the following functions (whose implementations are straightforward): $\mathsf{isLeaf}(v)$ and $\mathsf{isBranching}(v)$ determine if a query node v is a leaf or a branching node respectively, $\mathsf{leafNodes}(v)$ returns the set of leaf nodes in the twig rooted at v; and $\mathsf{dbl}(v)$ (for *direct branching or leaf node*) returns the set of all branching nodes b and leaf nodes f in the twig rooted at v such that there is no branching nodes along the path from v to b or f, excluding v, b or f. We will also refer to the (conceptual) functions $anc(e)$ and $desc(e)$ which denote the set of all ancestors and descendants of an element e in the document respectively.

3 Extended Dewey ID and Some Intuition

Tatarinov *et al.* [9] proposed the *Dewey ID* labelling scheme to represent the position of an element occurrence in an XML document. By labelling the root as an empty string ε, each non-root element u is labelled as $label(s).x$ where u is the x-th child of s. From now on, writing "$a < b$" means that element a has a smaller label than that of b.

The *extended Dewey ID* [7] encodes not only the positions but also the element names along the path from the root to the element. To overcome the problem of large label size, Lu *et al.* made use of schema constraints such as DTD or XML schema. They embedded such constraints in a *finite state transducer* so that given the *extended Dewey ID* of an element, all the element names along the path from the root to that element can be decoded efficiently. Due to this *ancestor name vision* property, determining if there is a matching between a simple query path (*i.e.* without twig) and a document path is now straightforward (taking time linear to the sum of lengths of the two paths). The remaining problem is to determine which root-leaf path matchings can be combined to form a matching for the whole query tree.

To give some intuition of our algorithm, consider the following problem that pops up again and again, namely, that of finding a matching for Q_v where v is a branching node with dbls $v_1, \ldots, v_d$. Suppose for each i, some matchings of Q_{v_i} have been found and S_{v_i} stores a set of elements with which v_i matches. Furthermore, assume that all the elements in each S_{v_i} lie on the same path (in the document tree). Now we want to combine the matchings of the Q_{v_i}'s to form matchings of Q_v.

Let e_i be the maximum element in S_{v_i}, *i.e.*, the element with the largest depth since all elements in S_{v_i} lie on the same path. Define $\mathsf{MB}(v_i,v)$ as the set of all ancestors, a, of e_i such that a can match node v in the path solution of e_i to P_{v_i}. Observe that if there is an element a present in all the $\mathsf{MB}(v_i,v)$'s, then for each i, there is a matching of Q_{v_i} such that the path P_v is mapped in the same way for different i's (in particular, node v is mapped to element a) while the subtree rooted at v_i is mapped to the document subtree rooted at e_i. This forms a matching for Q_v. In fact, other matchings may be possible by mapping the subtree rooted at v_i to elements in S_{v_i} higher than e_i but below a.

To detect if such an element a exists, it suffices to check for the intersection of $\mathsf{MB}(v_{max}, v) \cap \mathsf{MB}(v_{min}, v)$ where v_{max} and v_{min} are the children with the maximum and minimum e_i respectively. Any element in the intersection must also be present in every other $\mathsf{MB}(v_i,v)$ and hence qualifies for such an a. Also, note that all the elements in $\mathsf{MB}(v_{max}, v) \cap \mathsf{MB}(v_{min}, v)$ must lie on the same path from the root to the lowest common ancestor of e_{max} and e_{min} in the document tree. Hence we can set S_v to be $\mathsf{MB}(v_{max}, v) \cap \mathsf{MB}(v_{min}, v)$.

Thus we can associate with each node b in the query twig a set S_b and build up the matchings from the leaves towards the top branching node v_{tb} using the above idea. This gives us a recursive approach for the problem.

4 Details of Our Design

Our algorithm has the same high level structure as TJFast (shown in Algorithm 1). Like other holistic twig join algorithms, it operates in two phases. In the first phase (line 1-7), some solutions to individual root-leaf path patterns are computed. In the second phase (line 8), these solutions are merged to form the answers to the query twig pattern. To allow for efficient merging of root-leaf path into output during the second phase, a *blocking* technique is used in outputSolutions(line 5) so that the path solutions are in sorted order, irrespective of the root-leaf path provided. It is commonly employed in previous works [1,2,3,4,6,7], and its details are omitted due to space limitation. From now on, we focus on the task of outputting all and only those individual root-leaf paths that can be merged in the second phase, without duplication.

Algorithm 1. TJFaster

```
1: for all f ∈leafNodes(root) do
2:    locateMatchedLabel(f)
3: while ∃f ∈ leafNodes(root) : ¬ eof(T_f) do
4:    f_act =getNext(v_tb)
5:    outputSolutions(f_act)
6:    advance(T_f_act)
7:    locateMatchedLabel(f_act)
8: mergeAllPathSolutions()
```

The procedure locateMatchedLabel(f) locates the first element e in T_f such that P_e matches P_f. This is done by repeatedly matching get(T_f) with P_f and calling advance(T_f) until a match is found. It is called in the initialization step (line 1-2) and whenever a stream has just been advanced (line 6-7).

Function getNext(v) is the core function in the algorithm. Our getNext function is very similar to that of TJFast (shown in Algorithm 2). Later, we will point out the inefficiency in the function and the changes we made. Taking a query node v as parameter, the function updates the set S_v and returns a query leaf node $f \in$ leafNodes(v) whose stream is "safe to advance". Roughly speaking, S_v will be updated to contain the set of elements to which v maps, in the same matching of Q_v when get(T_f) is matched to f unless there is no such matching. Note that this set of elements will lie on the same (document) path.

By using the top branching node v_{tb} as parameter, TJFaster calls getNext repeatedly to identify the next stream f_{act} to advance among all the input streams. Before advancing $T_{f_{act}}$, those path matching solutions are output (in outputSolutions) if all the elements along the path of get(T_f) that match a branching node b can be found in the corresponding set S_b.

The main weakness of TJFast is that the leaf streams are only advanced in line 6 and 7 after an expensive call to getNext(v_{tb}) in the main loop. We will show here

Algorithm 2. getNext(v)

```
 1: if isLeaf(v) then
 2:    return v
 3: else
 4:    skipElement(v) {newly introduced in TJFaster}
 5:    for all v_i ∈dbl(v) do
 6:       f_i = getNext(v_i)
 7:       if isBranching(v_i) ∧ empty(S_{v_i}) then
 8:          return f_i
 9:       e_i = max{p|p ∈MB(v_i, v)}
10:    min = minarg_i{e_i}
11:    max = maxarg_i{e_i}
12:    for all v_i ∈dbl(v) do
13:       if ∀e ∈MB(v_i,v):e ∉ anc(e_max) then
14:          return f_i
15:    for all e ∈MB(v_min, v) do
16:       if e ∈ anc(e_max) then
17:          S_v = ((desc(e) ∪ anc(e)) ∩ S_v) ∪ {e}
18:    return f_min
```

Function MB(v, b)

```
1: if isBranching(v) then
2:    Let e = max{p|p ∈ S_v}
3: else
4:    Let e =get(T_v)
5: return the set of ancestors, a, of e such that b can be mapped to a in some mapping
   of P_v to P_e.
```

that there are other elements that will surely not contribute to any matchings and that we can identify them while performing a call to getNext. The immediate benefit is the saving of more useless recursive getNext calls. Moreover, we can filter elements by checking only their lexicographic order. This way, path pattern matchings need not be done on all input elements. Furthermore, in addition to the existing advance method, if the input stream T is indexed and supports an efficient method to forward the cursor to the first element such that get(T)$> e$ for any given element e, the I/O access can also be saved.

To achieve the above improvement, all we need is to add the function call SkipElement(v) (see Algorithm 3) in line 4 of getNext(v) (Algorithm 2).

Algorithm 3. skipElement(v)

```
1: Let f_max be leaf f ∈LeafNodes(v) with the maximum get(T_f)
2: a = the highest ancestor of get(T_{f_max}) such that P_v matchable to P_a
3: for all f in leafNodes(v) do
4:   if empty(S_v) or ∀e ∈ S_v : get(T_f) ∉ desc(e) then
5:     while get(T_f) < a do
6:       advance(T_f)
7:     locateMatchedLabel(f)
```

Given a query node v as a parameter, SkipElement skips those elements in the leaf streams in the twig rooted at v that cannot contribute to any output solution. Clearly, Q_v has a matching only if an element in each stream in leafNodes(v) is found to share at least one common ancestor a such that v matches a. In particular, P_v has to match P_a. The following lemma is also obvious:

Lemma 1. *Consider two elements, a and b in the document tree. If $a < b$, then either a precedes b (i.e., b follows a), or a is an ancestor of b (i.e., b is a descendant of a).*

We now argue that SkipElement(v) will only advance a stream T_f if get(T_f) cannot participate in any matching of Q_v. Note that we need only consider the possibility of get(T_f) forming a matching of Q_v with other get($T_{f'}$) or elements following them for all $f' \in$leafNodes(v). The other matchings involving elements of a stream $T_{f'}$ preceding get($T_{f'}$), if any, should have been processed already by the construction of getNext(v).

We first consider the case when S_v is empty. So there has not been any matching of Q_v. Consider two arbitrary leaves f and f' in leafNodes(v) and let a be the highest ancestor of get(T_f) such that P_v matches P_a. Suppose get($T_{f'}$) has a smaller label than that of a. Then get($T_{f'}$) precedes a (by Lemma 1) and by definition of a, get($T_{f'}$) cannot participate in any matching of Q_v involving get(T_f). Furthermore, get($T_{f'}$) precedes get(T_f) as well (since $a <$get(T_f)). Hence, get($T_{f'}$) cannot participate in any matching involving elements of T_f following get(T_f). In other words, get($T_{f'}$) is not useful in generating any new matching of Q_v. Hence it is safe to advance $T_{f'}$.

Note that the above argument holds for any f. Picking the stream f with the largest a will result in skipping the largest number of elements. In SkipElement(v), we therefore choose $f = f_{max}$ that maximizes get(T_f) (Line 1-2). This will give the largest corresponding ancestor a among all streams $f \in$leafNodes(v).

For the case when S_v is not empty, S_w is also nonempty for any descendant w of v. Moreover, some of the elements in the S_w's may form a matching of Q_v, possibly with some get(T_f) for some f in leafNodes(v). If a get(T_f) is involved, it can be proved that get(T_f) must be a descendant of some element in S_v. Hence we have the following lemma which is important in proving our algorithm correctness.

Lemma 2. *Let v, a and f be as defined in Algorithm 3. If S_v is empty, or* get(T_f) *is not a descendant of some element in S_v, then T_f can be advanced until* get(T_f) $> a$ *without missing any matching of Q_v.*

Theorem 1. *Given a twig query Q and an XML database D, algorithm* TJFaster *correctly returns all the answers for Q on D.*

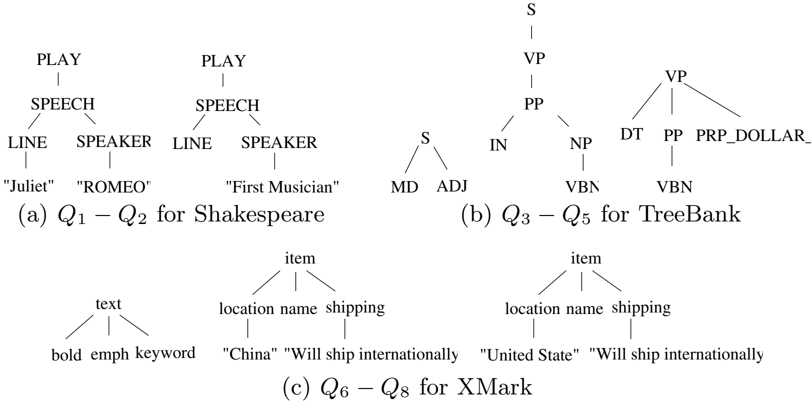
Now we analyse the complexities of our algorithm. Note that our algorithm uses no more space than that of TJFast. As for the query time, note that the stream cursors never go back during the execution of the algorithm. Therefore, the worst-case I/O time is $O(n)$ for any index method used, where n is the total number of input elements. Morever, any element scanned by TJFaster is also scanned by TJFast (which reads every input element in each stream once). Therefore, our algorithm is asymptotically no worse than TJFast in terms of the I/O cost. The next section will show that in practice, our algorithm is much faster than TJFast.

5 Experimental Evaluation

In this section, we present our experimental evaluation on the effectiveness of our enhancement. All experiments were conducted on a 2.6GHz Pentium 4 processor with 1GB main memory running Windows XP. We implemented TJFast and TJFaster in JDK 1.4 using PostgreSQL. TJFaster uses the identical program code with TJFast except for the additonal SkipElement function. We tested the algorithms with the following well-known datasets:

XMark : A synthetic benchmark dataset generated by the XML Generator, containing information about auctions.
Shakespeare plays : Shakespeare's plays in XML format.
TreeBank : A file with deep recursive structure.

We are interested in the following three performance measures: (1) the number of elements skipped, (2) the number of getNext calls, and (3) the elapsed time (which counts only the CPU cost). We measure the elapsed time by fetching the whole input streams into main memory before executing the algorithms. The number of elements skipped will indicate the effectiveness of the SkipElement



(a) $Q_1 - Q_2$ for Shakespeare (b) $Q_3 - Q_5$ for TreeBank

(c) $Q_6 - Q_8$ for XMark

Fig. 1. Twig Queries Tested

function and hence the reduction in the CPU cost of algorithm. If the input streams are indexed so that the cursors can be advanced directly to the desired element (without going through the uncessary elements), we expect that the I/O cost will also be proportional to the CPU cost.

We tested two twig queries in Shakespeare play "The Tragedy of Romeo and Juliet", three twig queries in TreeBank and three twig queries in XMark (Figure 1). We included some nodes that select text value (indicated by the double quotation marks) to control the selectivity. These queries represent some useful queries with semantic. Queries without text value are adopted from experiments in other earlier papers. All edges in the experimental queries are ancestor-descendant edges because TJFast deals with parent-child edges much in the same way as ancestor-descendant edges, though without guaranteeing optimality.

The major results are shown in Table 1 and Figure 2. The input size is counted as the total number of elements in all the input streams while the output size is the number of output elements in the leaf streams before the merging in phase 2.

Table 1. Major results

Query	**Input Size**	**Output Size**	**# of elts. skipped in TJFaster**	**# of getNext calls**		**Elapsed Time (in ms)**		**% of elts. skipped**	**Useless Recursion Avoided**
				TJFast	**TJFaster**	**TJFast**	**TJFaster**		
Q_1	167	0	149	495	48	47	1	89.2%	90.3%
Q_2	3102	18	2641	9306	1383	313	78	85.1%	85.1%
Q_3	13670	18	12434	40710	3654	1562	172	91.0%	91.0%
Q_4	168536	15732	126000	298890	47520	15235	4860	74.8%	84.1%
Q_5	155886	3349	128693	375664	30456	16375	2703	82.6%	91.9%
Q_6	21156	10276	9403	84624	47012	3047	2219	44.4%	44.4%
Q_7	5360	0	4400	10824	36	375	31	95.1%	99.7%
Q_8	7919	1173	2944	17264	5488	578	297	37.2%	68.2%

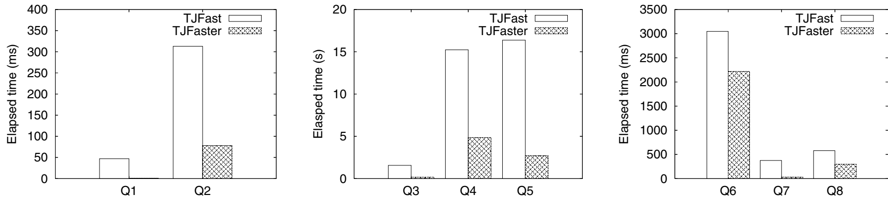


Fig. 2. Experimental results on computational time

The results show that the elapsed time is highly correlated to the number of getNext calls and the number of elements skipped. Moreover, the enhanced performance can now reflect the output size better rather than depending heavily on the input size as TJFast does. (For example, compare the elapsed time of Q_4 and Q_5 for the two algorithms.) We conclude that this optimization improves the previous design significantly, especially if the query is selective (i.e., produce small outputs).

6 Conclusions and Future Work

XML twig pattern matching is a key issue for XML query processing. In this paper, we have proposed TJFaster as an efficient algorithm to address this problem using some unexplored feature of *extended Dewey ID*. Our design improves the performance of the TJFast significantly when the selectivity is high. If an index of the input stream can efficiently support the finding of the successor of a given label, the new algorithm can avoid accessing elements that do not contribute to final results to save I/O access. Even without index support, the optimization saves the processing time in doing useless recursive calls and root-leaf pattern matchings.

For future work, we would like to apply the principle to *region encoding*, making use of RI-Tree[5], a relational data structure for selecting all intervals enclosing a given query point. Yuen and Poon [11] showed that an RI-tree can be used to support the ancestor axis efficiently in XPath when we regard each element as an interval, using *region encoding*. This may be used to substitute the *ancestor name vision property* when *extended Dewey ID* cannot be used.

References

1. Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the 2002ACM SIGMOD Conference on the Management of Data*, pages 310–321, 2002.
2. Byron Choi, Malika Mahoui, and Derick Wood. On the optimality of holistic algorithms for twig queries. In *DEXA*, pages 28–37, 2003.

3. Haifeng Jiang, Hongjun Lu, and Wei Wang. Efficient processing of XML twig queries with or-predicates. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 59–70, New York, NY, USA, 2004. ACM Press.
4. Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins in indexed XML documents. In *Proceedings of the 30th International Conference on Very Large Data Bases*, 2003.
5. Hans-Peter Kriegel, Marco Potke, and Thomas Seidl. Managing intervals efficiently in object-relational databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 407–418, 2000.
6. Jiaheng Lu, Ting Chen, and Tok Wang Ling. Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In *CIKM '04: Proceedings of the thirteenth ACM conference on Information and knowledge management*, pages 533–542, New York, NY, USA, 2004. ACM Press.
7. Jiaheng Lu, Tok Wang Ling, Chee-Yong Chan, and Ting Chen. From region encoding to extended dewey: on efficient processing of XML twig pattern matching. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 193–204. VLDB Endowment, 2005.
8. Praveen Rao and Bongki Moon. PRIX: indexing and query XML using Prüfer sequences. In *20th International Conference on Data Engineering*, pages 288–300, 2004.
9. Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of the 2002ACM SIGMOD Conference on the Management of Data*, pages 204–215, 2002.
10. H. Wang, S. Park, W. Fan, and P. Yu. Vist: A dynamic index method for querying XML data by tree structures, 2003.
11. Leo Yuen and Chung Keung Poon. Relational index support for XPath axes. In *XSym*, pages 84–98, 2005.