Holistic Twig Joins on Indexed XML Documents*

Haifeng Jiang Wei Wang Hongjun Lu
Dept. of Computer Science
The Hong Kong Univ. of Science and Technology
Hong Kong, China
Email: {jianghf, fervvac, luhj}@cs.ust.hk

Jeffrey Xu Yu
Dept. of Systems Engineering
and Engineering Management
The Chinese Univ. of Hong Kong
Hong Kong, China
Email: yu@se.cuhk.edu.hk

Abstract

Finding all the occurrences of a twig pattern specified by a selection predicate on multiple elements in an XML document is a core operation for efficient evaluation of XML queries. Holistic twig join algorithms were proposed recently as an optimal solution when the twig pattern only involves ancestor-descendant relationships. In this paper, we address the problem of efficient processing of holistic twig joins on all/partly indexed XML documents. In particular, we propose an algorithm that utilizes available indices on element sets. While it can be shown analytically that the proposed algorithm is as efficient as the existing state-of-the-art algorithms in terms of worst case I/O and CPU cost, experimental results on various datasets indicate that the proposed index-based algorithm performs significantly better than the existing ones, especially when binary structural joins in the twig pattern have varying join selectivities.

1 Introduction

XML is emerging as a *de facto* standard for information exchange over the Internet. Although XML documents could have rather complex internal structures, they can generally be modelled as *ordered trees*. Queries in XML query languages (see, e.g., [1, 2]) typically specify patterns of selection predicates on multiple elements which have some specified structural relationships. For example, to retrieve all paragraphs that are nested inside sections and have at least one figure and one table can be expressed as

```
//section//paragraph[figure AND table]
```

Such a query can be represented as a node-labelled twig pattern (or a small tree) with elements and string values as node labels [3].

Finding all occurrences of a twig pattern is a core operation in XML query processing [6, 13, 10, 15]. A typical approach is to first decompose the pattern into a set of binary structural relationships (parent-child or ancestor-descendant) between pairs of nodes, then match each of the binary structural relationships against the XML database and finally stitch together the results from those basic matches [20, 9, 14, 4, 8, 19]. For example, to answer the twig pattern above, we first retrieve all the `section`, `paragraph`, `figure` and `table` element sets, possibly through a *tag index*. A possible evaluation strategy works as follows: (1) finding all (paragraph, figure) and (paragraph, table) pairs with two separate structural joins; (2) merging these results to obtain the paragraphs with a figure and a table; and (3) joining these paragraphs with all sections through another structural join.

The main disadvantage of such a decomposition based approach is that intermediate result sizes can get very large, even when the input and the final result sizes are much more manageable. To address the problem, Bruno *et al* proposed a *holistic twig join* approach for matching XML query twig patterns [3]. With a chain of linked stacks to compactly represent partial results of individual query root-to-leaf paths, their approach merges the sorted lists of participating element sets altogether, without creating large intermediate results.

*The project was partly supported by the Research Grant Council of the Hong Kong SAR, China (Grants AoE/E-01/99 and HKUST6060/00E) and National 973 Fundamental Research Program of China (G1998030414).

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

The work reported in this paper is motivated by the following observations: Although the proposed holistic twig join algorithm has been proved to be I/O and CPU optimal in terms of input and output sizes for twigs with only ancestor-descendant edges (and yet still efficient for patterns with parent-child edges), the potential benefit of skipping elements that do not participate in a final twig match by using available indices is not fully explored. In our previous study on structural joins using indices [8], we found that the benefit of skipping elements without matches could be enormous when the input lists are large while few of the elements appear in the output.

In this paper, we address the problem of efficient holistic processing of twig joins on indexed XML documents. In particular, we propose a generic algorithm, *TSGeneric*$^+$, which can utilize available indices, e.g. XR-trees [8], on element sets. The main feature of *TSGeneric*$^+$ is that it uses indices to quickly locate the first match for a sub twig pattern (starting from the current elements of the input lists in the sub pattern). Locating the first match for a sub twig pattern can be evaluated by "fixing" edges that do not comply with the desired structural relationship (e.g., ancestor-descendant relationship) with a structural join like algorithm. The main issue here is which edge to choose first so that more elements without matches can be skipped. We propose three edge-picking heuristics in this paper: *top-down*, *bottom-up* and *statistics-based*, and study their performance with our experiments in comparison with existing algorithms.

Our contributions can be summarized as follows:

1. We propose a general holistic twig join processing algorithm, namely *TSGeneric*, which makes use of a set of stacks to cache elements and a cursor interface that provides standard methods to return elements with possible matches. With different implementations of the cursor interface, algorithms can be developed to process twig joins based on available access methods.

2. We propose the *TSGeneric*$^+$ algorithm based on *TSGeneric* to exploit more opportunities to skip elements. In particular, three different heuristics, *top-down, bottom-up* and *statistics-based*, are proposed to select the first edge to start the processing. As such, the potential of skipping elements with various indices is further exploited.

3. An extensive performance study with datasets of various characteristics was conducted. Our results show that the *TSGeneric*$^+$ algorithm on the XR-tree indexed data (regardless of the edge-picking heuristics used) significantly outperforms the existing algorithms, namely *TwigStack* and its variant *TwigStackXB*, in terms of various evaluation metrics. While among the three edge-picking heuristics, the statistics-based heuristic is most robust.

The rest of the paper proceeds as follows. Section 2 is dedicated to some background knowledge and related work on XML. We present the *TSGeneric* algorithm in section 3. Then we proceed to present the *TSGeneric*$^+$ algorithm and its worst case performance analysis in section 4. Section 5 reports experimental results. Section 6 concludes the paper.

2 Background and related work

2.1 Data model and numbering schemes

XML data is commonly modelled by a tree structure, where nodes represent elements, attributes and text data, and parent-child pairs represent nesting between XML element nodes. Most existing XML query processing algorithms rely on a positional representation of element nodes, where each element is represented with a tuple of three fields: $(start, end, level)$[1], based on its position in the data tree [20, 14, 4, 8]. Such a numbering scheme is also called as *region encoding*. Formally, element u is an ancestor of element v iff $u.start < v.start < u.end$. For parent-child relationship, we also test whether $u.level = v.level - 1$.

Corollary 1 *Given two elements e_i and e_j, if $e_i.start < e_j.start$ and e_i is not an ancestor of e_j, then e_i will not be an ancestor of any element e_x with $e_x.start > e_j.start$.*

Figure 1 shows a fictitious XML document which contains a root and other elements with tag a or b or c. The region encoding is also shown for each element.

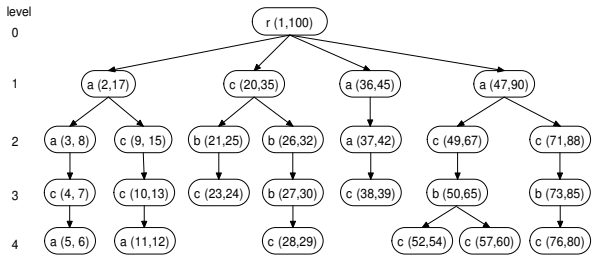


Figure 1: A sample XML document

2.2 Structural joins

A structural join is to find all occurrences of a given structural relationship between two sets of elements. A series of structural join algorithms were

[1]In an XML database with multiple XML documents, we may also need to record which document an element belongs to with an additional `DocId` field. Although we do not consider the `DocId` field in our algorithms, they can be easily extended to support `DocId`.

proposed in the literature. Merge-based algorithms include MPMGJN [20], $\mathcal{EE}/\mathcal{EA}$-Join [9] and Stack-Tree-Desc/Anc [14]. [4, 7, 8] are index-based approaches. In particular, XR-tree was recently proposed to index XML data to support efficient structural joins [8]. The experimental results showed that the XR-tree based algorithm, namely, *XR-stack*, performs the best among all existing structural join algorithms, especially when the join selectivity of (at least one of) the participating element sets is high [8] (i.e. few matched elements).

2.3 Twig pattern matching

A twig pattern is a selection predicate on multiple elements in an XML document. Such query patterns can generally be represented as node-labelled trees. Matching a twig pattern against an XML database is to find all occurrences of the pattern in the database. Formally, given a query twig pattern Q and an XML database D, a *match* of Q in D is identified by a mapping from nodes in Q to nodes in D, such that: (i) query node predicates are satisfied by the corresponding database nodes; (ii) the structural (parent-child or ancestor-descendant) relationships between query nodes are satisfied by the corresponding database nodes. The *answer* to Q with n nodes can be represented as an n-array relation where each tuple $(d_1, d_2, \cdots, d_n)$ consists of the database nodes that identify a distinct match of Q in D.

The most related work on twig pattern matching is a merge-based holistic twig join algorithm proposed in [3]. The recent work by Wu *et al* [19] emphasized on join order selection when a twig pattern is evaluated with the traditional decomposition approach.

3 The generic twig join algorithm

In this section, we first introduce a general setting of the twig join problem, followed by the description of a generic twig join framework, *TSGeneric*. Then we discuss how such generic work can be seamlessly extended to the case when element sets participating in the twig join are indexed.

3.1 Preliminaries

A twig pattern can be represented with a tree. The self-explaining functions `isRoot(`q`)` and `isLeaf(`q`)` examine whether a query node q is a root or a leaf node. The function `children(`q`)` gets all child nodes and `parent(`q`)` returns the parent node of q. The function `subtreeNodes(`q`)` returns node q and all its descendants. When there is no ambiguity, we may also refer to node q as the sub query tree rooted at q. In the rest of the paper, "node" refers to a tree node in the twig pattern (e.g., node q), while "element" refers to the elements in the dataset involved in a twig join.

We assume there is a data stream associated with each node in the query tree. Every element in the data stream is already encoded in the following region format: $(start, end, level)$. Each data stream is already sorted on the *start* attribute.

We also assume the join algorithms will make use of two types of data structures: cursors and stacks. Given a query tree T, we associate a cursor (C_q) and a stack (S_q) to every node $q \in T$, as shown in Figure 2.

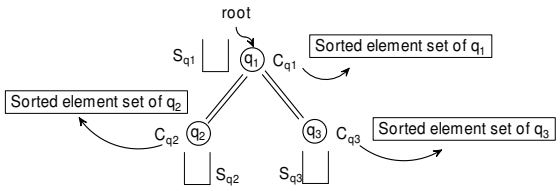


Figure 2: Cursors and stacks during execution

Each cursor C_q points to some element in the corresponding data stream of node q. Henceforth, "C_q" or "element C_q" will refer to the element C_q points to, when there is no ambiguity. The cursor can move to the element (if any) next to element C_q. Such behavior can be invoked with C_q→`advance()`. Similarly, we can access the attribute values of element C_q by C_q→*start*, C_q→*end* and C_q→*level*. Initially, all the cursors point to the first element of the corresponding data stream.

Initially, all stacks are empty. During query execution, each stack S_q may cache some elements before the cursor C_q and these elements are strictly nested from bottom to top, i.e. each element is a descendant of the element below it. We also associate with each element e in S_q a pointer to the lowest ancestor in $S_{parent(q)}$. Thus, we can efficiently access all e's ancestors in $S_{parent(q)}$. In fact, cached elements in stacks represent the partial results that could be further extended to full results as the algorithm goes on.

Next, we define an important concept, which is key to the understanding of the *TSGeneric* algorithm.

Definition 1 (Solution Extension) *We say that a node q has a solution extension if there is a solution for the sub query rooted at q composed entirely of the cursor elements of the query nodes in the sub query.*

Note that, if node q has a solution extension, since C_q is the ancestor of all cursor elements in the sub query tree nodes, C_q→*start* is smaller than all cursor start values of query nodes in the subtree q, based on the strictly nested property of XML data.

3.2 The generic twig join algorithm

Here, we briefly introduce algorithm *TSGeneric*, which is partly inspired by algorithm *TwigStack* proposed in [3]. The algorithm is shown in Algorithm 1.

getNext(q) returns a query node q_x in the subtree q, such that the following three criteria are met: (a) node q_x has a solution extension; and (b) if q_x has siblings,

Algorithm 1 *TSGeneric*(root)

```
1: while not end(root) do
2:    q = getNext(root);
3:    if not isRoot(q) then
4:       cleanStack(S_parent(q), C_q);
5:    if isRoot(q) or (not empty(S_parent(q))) then
6:       cleanStack(S_q, C_q);
7:       if not isLeaf(q) then
8:          push(S_q, C_q, top(S_parent(q)));
9:       else
10:         outputPathSolutionsWithBlocking(C_q);
11:   C_q→advance();
12: end while
13: mergeAllPathSolutions();
```

`Procedure cleanStack(`S_q`,` C_p`)`

1: pop all elements from S_q that are not ancestors of C_p;

`Procedure push(`S_q`,` C_p`,` *ptr*`)`

1: push the pair (C_p, *ptr*) onto stack S_q; {*ptr* is a pointer to an element in the parent stack;}

`Function end(`q`)`

1: return $\forall q_i \in \text{subtreeNodes}(q) : \text{isLeaf}(q_i) \Rightarrow \text{end}(C_{q_i})$

Algorithm 2 getNext(q)

```
1: if isLeaf(q) then
2:    return q;
3: for q_i in children(q) do
4:    n_i = getNext(q_i);
5:    if n_i ≠ q_i then
6:       return n_i;
7: end for
8: n_min = minarg_{n_i}{C_{n_i}→start};
9: n_max = maxarg_{n_i}{C_{n_i}→start};
10: while C_q→end < C_{n_max}→start do
11:    C_q→advance();
12: end while
13: if C_q→start < C_{n_min}→start then
14:    return q;
15: else
16:    return n_min;
```

then $C_{q_x}\rightarrow start < C_{q_s}\rightarrow start$, where q_s is a sibling of q_x (note that all q_x's siblings must have a solution extension, otherwise, a lower query node would have been returned through line 5-6 in the algorithm); and (c) if $q_x \neq q$, $C_{parent(q_x)}\rightarrow start > C_{q_x}\rightarrow start$.

The key to getNext is to apply Corollary 1. Called with the root of the query tree, getNext first traverses down to the left-most leaf node (by self recursive calls). Starting from the leaf node, it tries to find the highest possible query node with a solution extension by applying Corollary 1 (line 8-16). Given that all children have their own solution extensions (after line 7), in order for node q to be returned, we make sure that node q has a solution extension as well by advancing C_q (line 11). If no common ancestor for all C_{n_i} is found in q, we return the child node with the smallest start value (fulfilling criterion (b)), i.e. n_{min}. Note that as long as line 16 is executed and n_{min} is returned, the outer recursive calls to getNext will return the same node n_{min} all the way up through line 5-6.

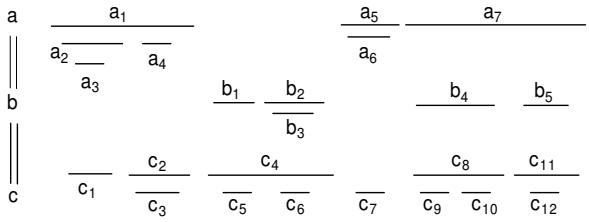


Figure 3: Query `//a//b//c` and element sets from the example document in Figure 1

Example 1 *Consider a path query* `//a//b//c` *on the element sets visualized in Figure 3. A subscript is added to each element in the order of their start values for easy reference. Initially, the three cursors are* (a_1, b_1, c_1)*. The first four calls of getNext(root) will always return node c (with cursors at* c_1 *to* c_4 *one by one) because none of elements* c_1 *to* c_4 *has an ancestor in set b. These c elements are consumed by the caller TSGeneric. Right before the fifth call, the cursors are* (a_1, b_1, c_5)*. The next call of getNext(root) will return node b, whose solution extension is* (b_1, c_5)*. In addition, the cursor of node a will be forwarded to* a_5*, the one after* b_1*. After several calls of getNext(root), we will eventually reach a cursor setup* (a_7, b_4, c_9)*, which is actually the first match of the query.*

It is not difficult to understand *TSGeneric* with the knowledge on how getNext works. In line 2, we retrieve the next node q to process with getNext(root). Some cached elements can be popped from $S_{parent(q)}$ (line 4) and S_q (line 6) according to Corollary 1. If q is not a leaf node, we push element C_q onto S_q (line 8); otherwise, all the path solutions involving C_q can be output (line 10). Note that path solutions should be output in root-to-leaf order so that they can be easily merged together to form final twig matches (line 13). As a result, we may need to *block* some path solutions during output, similarly as done in Stack-Tree-Anc [14] and showSolutionsWithBlocking [3].

3.3 Cursor interface for *TSGeneric*

Recently, there have been many proposals that advocate building certain indices on XML data to accelerate query processing [4, 7, 8, 3]. A natural question that follows is "Can we accelerate twig join processing in *TSGeneric* by taking advantage of available indices?".

Our answer to this question is affirmative. Briefly speaking, our solution is to extend the existing cursor interface to reflect new abilities to access elements through indices. In addition to the existing `advance()` method, we define two new methods:

1. C_q→`fwdBeyond`(C_p) forwards C_q to the first element e, such that $e.start > C_p$→$start$.

2. C_q→`fwdToAncestorOf`(C_p) forwards the cursor to the first ancestor of C_p and returns TRUE. If no such ancestor exists, it stops at the first element e, such that $e.start > C_p$→$start$, and returns FALSE.

The detailed discussion on various implementation issues of these methods will be presented in section 4.2. For now, the readers can simply assume the cursor interface as a black box.

With the additional methods, we can have an improved version of the *TSGeneric* algorithm. In fact, we only need to change the getNext implementation, which is now named to getNextCursor (Algorithm 3).

Algorithm 3 getNextCursor(q)

```
1: if isLeaf(q) then
2:    return q;
3: for q_i in children(q) do
4:    n_i = getNextCursor(q_i);
5:    if n_i ≠ q_i then
6:       return n_i;
7: end for
8: n_min = minarg_{n_i}{C_{n_i}→start};
9: n_max = maxarg_{n_i}{C_{n_i}→start};
10: if C_q→fwdToAncestorOf(C_{n_max}) == TRUE then
11:    if C_q is an ancestor of C_{n_min} then
12:       return q;
13: return n_min;
```

Line 10-13 in Algorithm 3 correspond to line 10-16 in Algorithm 2. The semantics is rather straightforward. If we find a common ancestor for all child cursors, we return q; otherwise, we return the child node with minimum start value, i.e. n_{min}.

The benefit of the new getNextCursor call can be illustrated with the following example:

Example 2 *Given the query* `//a//b//c` *and the element sets shown in Figure 3. Follow the example in Example 1. Right before the fifth call of getNextCursor (or getNext), the cursors are* (a_1, b_1, c_5)*. Although the next call of either getNextCursor or getNext will eventually forward the a's cursor to* a_5*, elements* a_2 *to* a_4 *need to be processed by getNext (line 10), while getNextCursor only needs to make a call of* C_q→*fwdToAncestorOf*($C_{n_{max}}$)*. As such, the processing time in getNextCursor is reduced.*

Example 2 illustrates the following fact: if an index can efficiently support the new cursor methods, the new algorithm getNextCursor can avoid accessing many of the elements that do not contribute to final results but have to be scanned in getNext. On the other hand, it is still possible to implement those methods based on the `advance()` method (though inefficient).

The significance of the improved *TSGeneric* algorithm can be summarized as that we give an integrated and flexible solution:

- Integrated: we can deal with all-indexed datasets or partly-indexed datasets (To the best of our knowledge, no existing algorithm can achieve this).
- Flexible: our generic join framework can be used with any (or almost all existing) index for region coded XML datasets (B^+-tree, R-tree, XB-tree and XR-tree).

4 The *TSGeneric*$^+$ algorithm

Although *TSGeneric* is able to skip some elements without matches through the generic cursor interface, the potential of skipping elements is in fact not fully exploited. In this section, we explore the advantage of skipping elements through the generic cursor interface with various heuristics.

Exploiting more benefit of skipping elements without matches relies heavily on a cursor-based structural join algorithm, namely, SJCursor, which is evaluated over edges of the query tree. The SJCursor algorithm (evaluated over an edge of the query tree) finds the first ancestor-descendant pair starting from the current cursors of the two nodes connected by the edge. Algorithm 4 shows the SJCursor algorithm. An edge (p, c) is defined as "broken" if elements C_p and C_c do not have an ancestor-descendant relationship (see, function `isBroken`). SJCursor works as follows. If the edge is not broken, or either C_p or C_c reaches the end, it returns. Otherwise, if C_p→$start$ is smaller than C_c→$start$, we call C_p→`fwdToAncestorOf`(C_c) to move C_p to the first ancestor element of C_c (or beyond C_c if no such ancestor exists); otherwise, we forward C_c to the first element whose start value is larger than C_p→$start$, with C_c→`fwdBeyond`(C_p), because a descendant element must have its start value larger than that of its ancestor element.

Algorithm 4 SJCursor (p, c)

```
1: while (not end(C_p)) and (not end(C_c))
   and isBroken(p, c) do
2:    if C_p→start < C_c→start then
3:       C_p→fwdToAncestorOf(C_c);
4:    else
5:       C_c→fwdBeyond(C_p);
6: end while
Function isBroken(p, c)
1: return not (C_p→start < C_c→start and C_c→start <
   C_p→end);
```

Example 3 *Consider again the sample query in Figure 3. Suppose the current cursors are* (a_1, b_1, c_1)*. To find the first match for the query* `//a//b//c`*, which*

is (a_7, b_4, c_9), a better evaluation strategy would be to first find the matching pair (a_7, b_4) between node a and node b by calling SJCursor(a,b), and then call SJCursor(b,c) to find the matching pair (b_4, c_9). It is easy to verify that SJCursor(a,b) only needs to access five elements, i.e. a_1, b_1, a_5, b_4 and a_7, while SJCursor(b,c) only accesses elements c_1 and c_9. As a result, with this evaluation strategy, only 7 elements need to be processed, which is significantly better than the TSGeneric algorithm.

The challenge of such unordered evaluation as shown in Example 3 is that it might violate the correctness of the *TSGeneric* algorithm, because we may erroneously skip elements that do have matches. The following lemma is important to identify when such unordered evaluation is possible.

Lemma 1 *Suppose a call of getNextCursor(root) returns a query node q. If the stack S_{q_a} of any ancestor node q_a of node q is empty, then the current extension of node q does not contribute to any further results and element C_q can be discarded.*

Proof There are two cases. The first case is that the ancestor node q_a (whose stack is empty) is the parent of node q. Since $q \neq$ root(otherwise, it could not have any ancestor), according to the criterion (c) of a query node returned by getNext, the start value of C_q must be smaller than the start value of C_{q_a} upon return. Given that S_{q_a} is empty, it is clear that the extension of q could not contribute to any new results. The second case is that q_a is not the parent node of q. In other words, there exists some node q' which is the parent node of q (obviously, q_a is also an ancestor of q')[2]. Since $S_{q'}$ is not empty, the extension of q could possibly have an ancestor in $S_{q'}$. We now prove that the elements in $S_{q'}$ will eventually be popped without contributing to any further results. The reason is that for any element in $S_{q'}$, all its ancestors in q_a (if any) would have already been returned by previous calls of getNextCursor(root) and popped. □

According to Lemma 1, if the stack of some node q is empty, then, it is useless for getNextCursor(root) to return a node q' that is a descendant of q in the query tree. In other words, as long as we discover the stack of a node q is empty in the recursive call of getNextCursor, there is no need to further call getNextCursor for q's children. Rather, we should try to locate a solution extension for node q. Based on this, we improve getNextCursor by incorporating an extension-locating procedure, as shown in Algorithm 5. We name it as getNextExt. Lines 3-5 are newly added codes to getNextCursor: if the stack of node q is empty, the procedure LocateExtension (Algorithm 6) is called, which finds the first solution extension of node q, and then we simply return q. We call the *TSGeneric* algorithm *TSGeneric*$^+$ if it calls getNextExt(root), other than getNextCursor(root).

Algorithm 5 getNextExt (q)

```
1: if isLeaf(q) then
2:    return q;
3: if empty(S_q) then
4:    LocateExtension (q);
5:    return q;
6: for q_i in children(q) do
7:    n_i = getNextExt (q_i);
8:    if n_i ≠ q_i then
9:       return n_i;
10: end for
11: n_min = minarg_{n_i}{C_{n_i}→start};
12: n_max = maxarg_{n_i}{C_{n_i}→start};
13: if C_q→fwdToAncestorOf(C_{n_max}) == TRUE then
14:    if C_q is an ancestor of C_{n_min} then
15:       return q;
16: return n_min;
```

Algorithm 6 LocateExtension (q)

```
1: while (not end(q)) and (not hasExtension(q)) do
2:    (p, c) = PickBrokenEdge (q); {see section 4.1}
3:    SJCursor (p, c);
4: end while
Function hasExtension(q)
1: for each edge (p, c) in the sub query tree q do
2:    if isBroken(p, c) then
3:       return FALSE;
4: end for
5: return TRUE;
```

Consider the LocateExtension algorithm. It runs in a "pick-and-fix" fashion. Each time, it picks a broken edge (discussed in section 4.1) and fixes it with SJCursor, until node q has a solution extension or any cursor in the subtree reaches the end. Note that the overhead to check for broken edges in function hasExtension is minimal because all the operations are carried out on the cursor elements of the query nodes and only negligible CPU cost is involved.

It is obvious that algorithm *TSGeneric*$^+$, i.e. *TSGeneric* calling LocateExtension, works correctly and we have the following theorem (proof omitted in the interest of space):

Theorem 1 *Given a query twig pattern Q and an XML database D, the TSGeneric*$^+$ *algorithm correctly returns all answers for Q on D.*

4.1 Heuristics for picking an edge

Picking the next query edge to fix is essentially a query optimization problem. The optimization problem of join order selection has been extensively studied in the

[2]Note that, according to algorithm *TSGeneric*, it is possible to have a node q_x whose stack is not empty while the stack of some ancestor node of q_x is empty because we do not pop stacks downward.

context of relational databases [12, 11, 16]. Due to the subtle difference in problem contexts, such previous work is not directly applicable to holistic twig joins considered here. In this subsection, we present some edge-picking heuristics.

Intuitively, we should choose an edge whose next match is the farthest from the current cursors of its two nodes, so that we can skip the most number of elements (without matches) when fixing other query edges. This has been illustrated in Example 3. We denote this heuristic as MD, reading "maximum distance".

The MD heuristic can leverage the work on structural join size estimation [18, 17]. We now give the formula for estimating the average inter-match distance for a query edge (p, c), whose two nodes are associated with element sets p and c respectively. Assume that we have statistics about the total number of elements in each set, i.e. N_p and N_c, the width of the workspace for all regions in each element set, i.e. W_p and W_c. If we can estimate the percentage $s_p\%$ of p elements and the percentage $s_c\%$ of c elements that have at least one match in the structural join between p and c (e.g., using histograms or sampling techniques proposed in [17]), then, the average distance between each match can be approximated as (assuming a uniform distribution):

$$AvgDist_{p \lhd c} = min(\frac{W_p}{N_p \cdot s_p\%}, \frac{W_c}{N_c \cdot s_c\%})$$

Note that the two distance values estimated from set p and set c should be similar if elements in both sets are rarely nested. But the estimates might be different for highly nested datasets. We choose the smaller one.

Statistics might not always be available in realistic applications. We propose other two heuristics that work without assuming any knowledge about the element sets: *top-down* (TD) and *bottom-up* (BU).

Algorithm 7 shows the complete PickBrokenEdge algorithm. First, we retrieve all the broken edges in q using a breadth first traversal order. The total number of broken edges is assumed to be K (which is subject to change in different calls of PickBrokenEdge). Then, an edge is picked from the K broken edges according to the heuristic specified by variable `heuristic`. Ties are broken arbitrarily in heuristic MD. By the top-down heuristic, we always choose the first broken edge. The deepest, right-most edge is chosen in the bottom-up heuristic.

Algorithm 7 PickBrokenEdge (q)

```
1: Let Edges[1...K] be the vector containing all K broken
   edges in q in breadth first order;
2: if heuristic == MD then
3:    (p_s, c_s) = maxarg_{(p_i,c_i)} AvgDist_{p_i ◁ c_i}
4: else if heuristic == TD then
5:    (p_s, c_s) = Edges[1];
6: else
7:    (p_s, c_s) = Edges[K];
8: return (p_s, c_s);
```

4.2 Cost analysis of *TSGeneric*$^+$

Although utilizing indices of various kinds built on element sets is expected to speed up the efficiency of the cursor interface to different extent, it is not clear whether the same optimal worst-case I/O and CPU cost can be achieved when indices are used, compared to the *TwigStack* algorithm [3]. In this section, we address this problem.

Though, the effectiveness of different index structures varies, the worst-case I/O and CPU cost for accessing elements through cursor interfaces built on them can be shown to be linear to the size of the elements indexed. The intuition is that each method of the cursor interface always drivers the cursor forward. To put it differently, the cursor never goes back. Therefore, it can be concluded that *TSGeneric*$^+$ that utilizes indices through cursor interfaces is as efficient as *TwigStack* in terms of worst-case I/O and CPU cost.

We focus ourselves on XR-tree as an example to show how a cursor interface based on indices can be implemented to achieve linear worst case I/O and CPU cost. The cursor interface implementation for other existing index structures should be similar. In the following, we first briefly introduce the structure of the XR-tree index, and then describe the details of cursor interface implementation based on XR-tree.

4.2.1 The XR-tree index

The XR-tree is an index structure recently proposed in [8] for indexing XML data based on the region encoding, i.e. $(start, end, level)$. An XR-tree is basically a B$^+$-tree (built on the *start* field of all indexed elements) augmented with stab lists and bookkeeping information in internal nodes. An element e is included in the stab list of an index page I if: (1) there exists some key k_i in I such that $e.start \leq k_i \leq e.end$ (or k_i stabs the region of element e); and (2) no ancestor page of I has a key that stabs e, i.e. I is the highest index page that stabs e. Figure 4 shows the XR-tree for the set of c elements in the example document in Figure 1. Note that element $(20, 35, 1)$ and $(49, 67, 2)$ are stabbed by index pages so that they are included in stab lists and also marked with "yes" in leaf pages.

Given an element e, searching for all its descendants in an element set R indexed by an XR-tree is as simple as a B$^+$-tree range search, i.e. $e.start < R.start < e.end$. The novelty of XR-tree is that all the ancestors of e can be collected from the stab lists of index pages and the leaf page when we navigate down the XR-tree using $e.start$ (similar to a B$^+$-tree equality search).

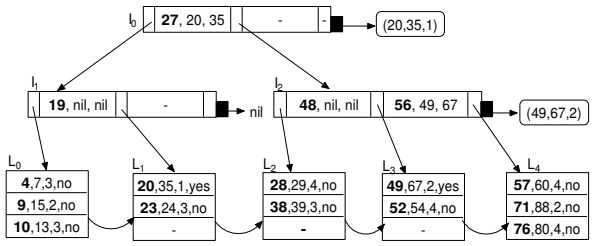


Figure 4: The XR-tree for c elements in Figure 1

4.2.2 Implementation of the cursor interface

In our implementation, we keep in buffer the pages from root to leaf containing the cursor, denoted as *hot path*. For example, if the cursor is at element $(23, 24, 3)$ (Figure 4), then the hot path is $I_0 \rightarrow I_1 \rightarrow L_1$. Besides, we also keep the index entries followed to form the hot path. The current entry for page I_0 is -1 (the left-most entry), the entry for I_1 is 0 and the entry for L_1 is 1. The entry information is useful to establish the linear CPU cost for index accesses.

To implement the C_q→`advance()` method, if C_q is not the last element of the current leaf page, we simply points it to the next element without changing the hot path. Otherwise, we free the current leaf page and fetch in the next leaf page through the parent index page. If there is no more entry in the parent index page, we fetch in another index page through its parent similarly. For example, if the cursor is at $(9, 15, 2)$, to advance it, we simply move to the next element $(10, 13, 3)$. To advance again, we need to free L_0 and fetch in L_1. Note that the entry information of I_1 is also incremented. If the cursor is at $(23, 24, 3)$, we need to replace I_1 with I_2.

C_q→`fwdBeyond`(C_p) is as simple as a B$^+$-tree search. Starting from the current index entry of the root, sequentially scan the entries until the largest entry k_i, such that $k_i \leq C_p$→$start$ is found. $k_i.rightChild$ is the child page to be searched. If $k_i.rightChild$ is different from the buffered page at the lower level, fetch in $k_i.rightChild$ and replace the old page. The search keeps going on until we reach a leaf page. Then we set the cursor to the first element whose start is larger than C_p→$start$.

Consider the C_q→`fwdToAncestorOf`(C_p) method. It is obvious that we do not need to touch any leaf pages or index pages (including their stab lists) that are before the hot path because what we want is the ancestor e_a of C_p, such that $e_a.start \geq C_q$→$start$. If a qualified ancestor is found in some leaf page, the path from root to this leaf page would be the new hot path. If a qualified ancestor is found in some stab list of an index page, then the new hot path must include this newly accessed index page.

Theorem 2 *In the TSGeneric$^+$ algorithm, the worst case I/O and CPU cost for accessing an element set R through the cursor interface implemented on the XR-tree of set R is linear to the size of R.*

4.3 Twig joins with parent-child edges

Algorithm *TSGeneric$^+$* can still be used to evaluate twig patterns with parent-child edges. The difference is that when we output a root-to-leaf path solution, we check the parent-child relationship using the *level* attribute of elements for parent-child edges. But the optimality in terms of worst case I/O and CPU cost is no longer guaranteed. In particular, the algorithm might produce path solutions that do not contribute to any final match.

We use an example to illustrate the point. Consider a twig pattern `//a[b]//c` to be evaluated again the sample dataset in Figure 3. In the *TSGeneric$^+$* algorithm, the first call of getNextExt(root) will return node a with cursors at (a_7, b_4, c_9) (Recall that LocateExtension is called for root node a since S_a is empty initially). Then a_7 is pushed onto stack (line 7 in Algorithm 1). The next call of getNextExt(root) returns b_4. Since b_4 does not qualify the parent-child relationship with a_7, the path (a_7, b_4) is not output. The third call of getNextExt(root) returns c_9, and the path (a_7, c_9) is output (line 10 in Algorithm 1). Similarly, the paths for c_{10} and c_{11} will both be output. But the twig pattern actually has zero result. The problem here is that we "wrongly" put a_7 onto stack, which does not have a qualifying b child element at all! (Recall that we only knew that a_7 has a descendant b_4 when we pushed it) Then, is it possible to efficiently check the existence of one b child before pushing a_7? The answer is, such a checking might be very costly because the first child of an element could be far away from its first descendant element.

The recent work by Choi *et al* [5] also provided some insight into the cause of the suboptimality in evaluating twig patterns with arbitrarily mixed ancestor-descendant and parent-child edges.

5 A performance study

In this section, we present the experiments conducted to evaluate the effectiveness of various algorithms and heuristics proposed in the paper and report some of the results obtained.

5.1 Experimental setup

As Table 1 shows, with different combinations of choices in the dimensions of *index* and *algorithm*, we have many algorithms for twig joins. Here, we will focus on three kinds of algorithms, namely the *TwigStack*, *TwigStackXB* and *XRTwig* algorithms. The first two algorithms were chosen as they were the best twig join algorithms prior to this paper. We also implemented *TSGeneric$^+$* with other kinds of indices, however, *TSGeneric$^+$* with the XR-tree index, i.e.,

XRTwig, usually performs best. We have also implemented the three variants of *XRTwig* based on different edge-picking heuristics (top-down, bottom-up and maximum-distance), resulting in the *XRTwig(TD)*, *XRTwig(BU)* and *XRTwig(MD)* algorithms respectively. We will later compare those algorithms from several perspectives.

Table 1: Diagrammatic view of algorithms

Index \ Algo	*TSGeneric*	*TSGeneric*$^+$
No Index	*TwigStack*	
XB-tree	*TwigStackXB*	
XR-tree		*XRTwig*
B$^+$-tree		
R-tree		

We evaluated the performance of those join algorithms using the following three metrics:

- *number of elements scanned.* We measure the total number of elements scanned during a join, which reflects the ability of each algorithm to skip elements that do not belong to the final result.
- *number of page accesses.* This metrics measures the performance of algorithms in terms of I/O cost.
- *running time.* The running time of an algorithm is obtained by averaging the running times of several consecutive runs with *hot* buffers.

In this paper, we mainly report our results that demonstrate the performance of algorithms for data with different characteristics. We fixed a set of queries and executed those queries on different datasets, designed with different kinds of *selectivity*. Intuitively, a high selectivity (i.e., few matches) tends to favor algorithms utilizing indices.

Three query patterns were selected to represent different classes of twig patterns. They include a simple path query (Q1), a deep twig (Q2) and a bushy twig (Q3) as shown in Figure 5. All edges in the queries are ancestor-descendant relationships because all these algorithms deal with parent-child edges much the same way as ancestor-descendant edges, though, without guarantee of optimality.

We chose to generate synthetic datasets so that we can better control the relationship between the algorithms and the characteristics of the datasets. We generated 8 datasets for Q1, 10 for Q2 and 10 for Q3. There are two types of datasets: with varying selectivities and with the same selectivity. For example, we generated 4 datasets $DS_i, 1 \leq i \leq 4$, for Q1 in which the selectivities of different edges are different. For DS_1, the selectivity of the $A-B$ edge is 1% while the selectivity of the $B-C$ edge is 10%. We used a "round-robin" fashion method to generate other datasets by

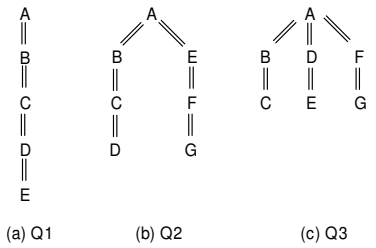


Figure 5: Three query patterns used

cyclically assigning the selectivity of the current edge to the next edge. For datasets of fixed selectivity, we simply assigned the same selectivity to all the edges. For example, all the edges of Q1 on dataset DS_5 have the same selectivity, i.e. 1%.

Table 2 gives the description of datasets with varying selectivities for Q1. We also generated 4 datasets, DS_5 to DS_8, with fixed edge selectivity values of 1%, 10%, 50% and 100% respectively.

Table 2: Selectivity of edges in Q1 for DS_1 to DS_4

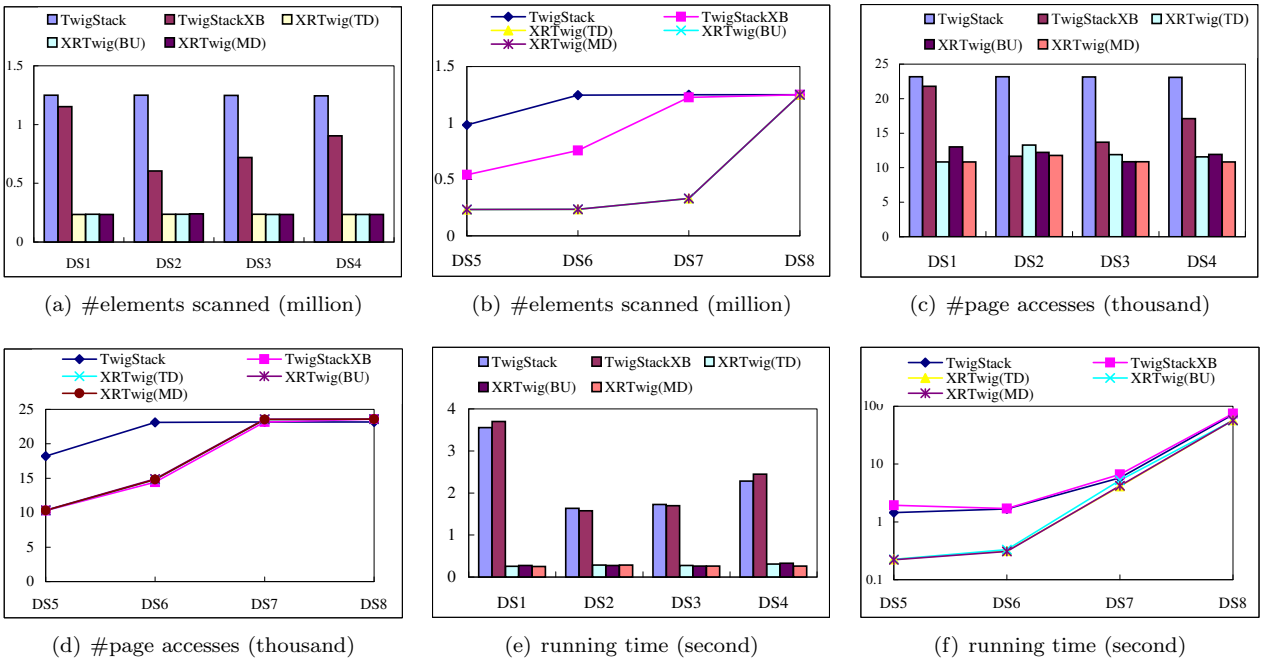
datasets	$A \triangleleft B$	$B \triangleleft C$	$C \triangleleft D$	$D \triangleleft E$
DS_1	1%	10%	50%	100%
DS_2	10%	50%	100%	1%
DS_3	50%	100%	1%	10%
DS_4	100%	1%	10%	50%

Since Q2 has six edges in the pattern, we used six different join selectivity values to generate six "selectivity round-robin" datasets, as shown in Table 3. Similarly as for Q1, four other datasets DS_7 to DS_{10} were generated to test the case when all edges in Q2 have the same selectivity, i.e. 1%, 10%, 50% and 100% respectively. The 10 datasets for Q3 were similarly generated and the description is omitted in the interest of space.

Table 3: Selectivity of edges in Q2 for DS_1 to DS_6

DS	$A \triangleleft B$	$A \triangleleft E$	$B \triangleleft C$	$E \triangleleft F$	$C \triangleleft D$	$F \triangleleft G$
DS_1	1%	10%	25%	50%	75%	100%
DS_2	10%	25%	50%	75%	100%	1%
DS_3	25%	50%	75%	100%	1%	10%
DS_4	50%	75%	100%	1%	10%	25%
DS_5	75%	100%	1%	10%	25%	50%
DS_6	100%	1%	10%	25%	50%	75%

Each element set generated contains 250K elements. As a result, each dataset for Q1 involves more than one million elements while each dataset for Q2 and Q3 has near two million elements. Elements can be self-nested up to five levels. The join result size for each dataset varied according to the selectivity of edges. Take Q1 as an example. The numbers of path solutions for DS_1 to DS_4 are 29K, 36K, 38K and 35K respectively. There is no output solution when all edges are very selective (1% for DS_5) while there are more than 70



(a) #elements scanned (million) (b) #elements scanned (million) (c) #page accesses (thousand)

(d) #page accesses (thousand) (e) running time (second) (f) running time (second)

Figure 6: Experimental results for query Q1

million path solutions when the selectivity of edges is very low (100% for DS_8).

Our test-bed is an experimental database system that includes a storage manager, a buffer pool manger, B$^+$-tree, XB-tree and XR-tree index modules. All the algorithms were coded using Microsoft Visual C++ 6.0. All the experiments were conducted on a Pentium IV 1.60GHz PC with 512M RAM and a 30G hard disk, running Windows XP. The page size used is 4K and we used the file system as the storage. All the experimental results presented below were obtained with a fixed buffer pool size: 500 pages, just large enough to *cache* the hot paths.

Figure 6 shows all the results on Q1 for the five different algorithms/heuristics and 8 datasets. Figure 7 and 8 give the numbers of elements scanned and numbers of page accesses on Q2 and Q3 respectively (we ignore the results for *TwigStack* because it performed almost always worse than *TwigStackXB* as can be seen from Figure 6).

5.2 Effects of using indices

[3] reported experiment results of *TwigStack* and *TwigStackXB*. Our results coincide with theirs. Specifically, both the number of elements scanned and the number of page accesses for *TwigStackXB* are significantly smaller than those for *TwigStack* in most cases. For example, *TwigStack* scanned about 2 times as many elements as *TwigStackXB* for DS_2 on Q1 (See Figure 6(a)). This is simply because *TwigStack* always scans almost all the elements while *TwigStackXB* can identify and skip unmatched elements. On the other hand, under the extreme case where there is hardly any element that can be skipped, for example, the DS_8 on Q1, *TwigStack* performs the best, scanning the least number of elements and incurring the least amount of I/O. *TwigStackXB* will have some overhead due to the access to the internal pages of the index, although such overhead is negligible.

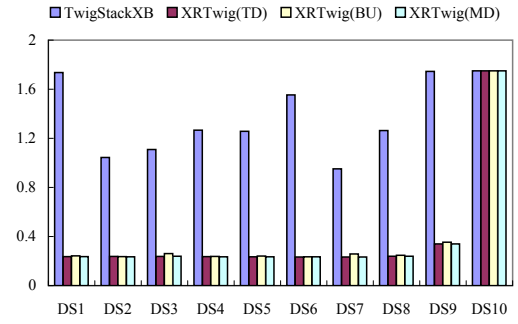
It is interesting to compare their performance, for example, in terms of the number of elements scanned, of DS_1 on Q1. For this dataset, the selectivity of edges changes from 1% to 100% from top to bottom. As a result, there is not much opportunity for *TwigStackXB* to skip elements through the `fwdToAncestorOf` operation (line 10 in Algorithm 3) because the operation is very frequently called due to the low selectivity of lower edges. On the contrary, for DS_2 for which the lowest edge has selectivity 1%, the matches between D and E are rare, therefore, there is high potential for skipping elements in upper query nodes.

We conclude that twig join algorithms can enhance their performance consistently with the help of indices in most cases, without incurring noticeable overhead even for the worst case scenario.

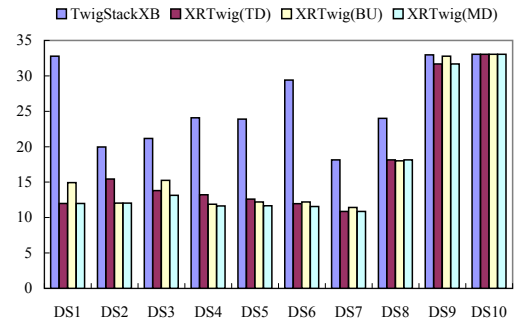
5.3 *TSGeneric*$^+$ vs. *TSGeneric*

Now we compare the performance of all the algorithms based on *TSGeneric*$^+$ with the algorithm based on *TSGeneric* (i.e., *TwigStackXB*). Both classes of algorithms can make use of indices to skip some unmatched elements.

It can be observed that, for example, from experiment results on Q1 (Figure 6), *TSGeneric*$^+$ based algorithms are usually much more efficient than *TwigStackXB*. For Q1 in terms of running time,

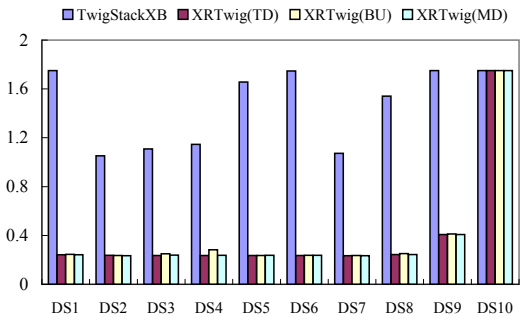


(a) #elements scanned (million)

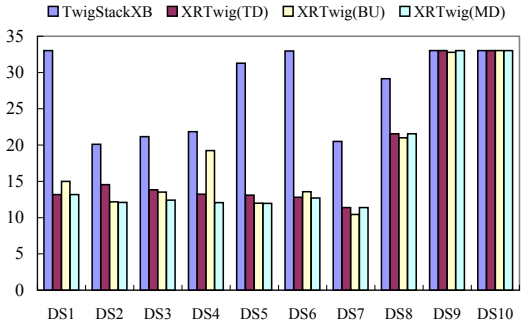


(b) #page accesses (thousand)

Figure 7: Experimental results for query Q2



(a) #elements scanned (million)



(b) #page accesses (thousand)

Figure 8: Experimental results for query Q3

TSGeneric$^+$ based algorithms could be up to 10 times faster than *TwigStackXB*. Similar trends can be observed in other examples and in terms of other metrics as well. For example, we find that the advantage of *TSGeneric*$^+$ based algorithms over *TwigStackXB* is even greater for most datasets on Q2 and Q3. In particular, for DS_1 in terms of the number of elements scanned (see, Figure 7(a) and 8(a)), *TSGeneric*$^+$ only scanned less than 1/7 of the total elements scanned by *TwigStackXB*. Meanwhile, in terms of the number of page accesses, *TwigStackXB* performed much worse than *TSGeneric*$^+$, even for datasets with low selectivity DS_7 to DS_{10} (see, Figure 7(b) and 8(b)), though it showed similar performance with *TSGeneric*$^+$ for those fixed selectivity datasets on Q1.

The performance advantage of *TSGeneric*$^+$ over *TSGeneric* mainly attributes to the fact that with a "pick-and-fix" strategy for locating solution extensions, *TSGeneric*$^+$ stands more chances of taking the advantage of edges with high selectivity to skip elements. Furthermore, for twig pattern queries, the cursor advances in one branch help more in *TSGeneric*$^+$ than in *TSGeneric* to skip elements in another branch due to the unordered evaluation of edges in the extension location procedure of *TSGeneric*$^+$.

Generally speaking, *TSGeneric*$^+$ has stronger capability to identify and skip unmatched elements, especially when the twig pattern is complex.

5.4 Comparison of heuristics

In this subsection, we study the three proposed heuristics for edge-picking. To that end, we also summarize, in Figure 9, the numbers of page accesses for all the queries with varying edge selectivities on Q1, Q2 and Q3.

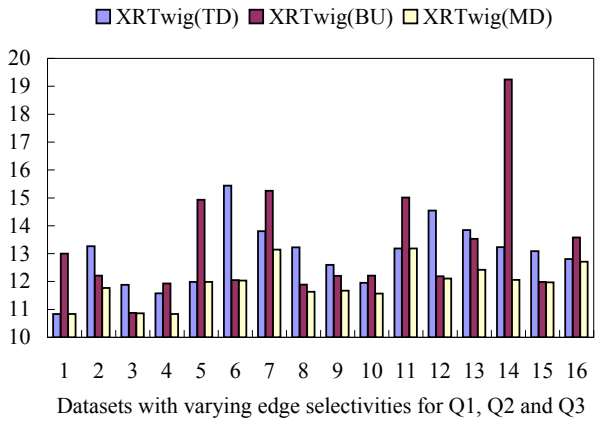


Figure 9: #Page accesses under different edge-picking heuristics (thousand)

We can draw several interesting observations from Figure 6 to Figure 9:

- It is quite surprising that the difference between the three edge-picking heuristics of the *TSGeneric*$^+$ algorithm is not significant, in terms

of the number of elements scanned (see, Figure 6(a), 6(b), 7(a) and 8(a)). This is mainly due to the fact that although MD always picks the most selective edge (with accurate estimates) to fix, other heuristics somehow also have the opportunity to select such an edge (after other *high priority* edges have been fixed).

- Despite their similar numbers of elements scanned, the three heuristics show various I/O performance (Figure 9). In particular, *XRTwig(MD)* has the best overall I/O performance, which indicates that, although *XRTwig(MD)* did not scan much less elements than algorithms based on the other heuristics, it managed to *cluster* those element scans better. For example, *XRTwig(BU)* for DS_4 on Q3 (X = 14 in Figure 9) performed 1.6 times as much I/O as *XRTwig(MD)*.

- The relative performance of *XRTwig(TD)* and *XRTwig(BU)* varies with the different formation of selectivities on the twig edges. Generally speaking, better I/O performance can be observed when the more selective edges have higher priority in terms of edge-picking heuristics. For example, *XRTwig(TD)* performed better than *XRTwig(BU)* for DS_1 and DS_4 on Q1, while *XRTwig(BU)* is better than *XRTwig(TD)* for DS_2 and DS_3 on Q1.

In summary, MD algorithm always performs the best as long as accurate statistics are available.

6 Conclusions

In this paper, we addressed the problem of efficient evaluation of holistic twig joins on all/partly indexed XML documents. In particular, we proposed *TSGeneric*$^+$ with three different evaluation heuristics, namely, top-down, bottom-up and statistics-based. Experimental results indicated that the *TSGeneric*$^+$ algorithm on XR-tree indexed datasets performs significantly better than the existing ones by much more effectively skipping elements that do not contribute to final results, especially when binary structural joins in the twig pattern have varying selectivities. Among the three heuristics we considered, the statistics-based heuristic is most robust, given that the statistics used are accurate. As such, existing work on join selectivity estimation for XML data can be leveraged.

Regarding our future work, we will investigate new heuristics for the edge picking process in *TSGeneric*$^+$, which may require more sophisticated estimation techniques for XML data. Another future work is to consider more complex queries, like a query pattern consisting of multiple twig patterns.

References

[1] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon. XML path language (XPath) 2.0. Technical report, W3C, 2002.

[2] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. Technical report, W3C, 2002.

[3] N. Bruno, D. Srivastava, and N. Koudas. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, pages 310–321, 2002.

[4] S.-Y. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *VLDB*, pages 263–274, 2002.

[5] B. Choi, M. Mahoui, and D. Wood. On the optimality of holistic algorithms for twig queries. In *DEXA*, 2003.

[6] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[7] T. Grust. Accelerating XPath location steps. In *SIGMOD*, pages 109–120, 2002.

[8] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient structural joins. In *ICDE*, pages 253–264, 2003.

[9] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, pages 361–370, 2001.

[10] J. McHugh and J. Widom. Query optimization for XML. In *VLDB*, pages 315–326, 1999.

[11] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, pages 314–325, 1990.

[12] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[13] J. Shanmugasundaram, K. Tufte, C. Zhang, H. Gang, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, pages 302–314, 1999.

[14] D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, pages 141–152, February 2002.

[15] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, pages 204–215, 2002.

[16] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *SIGMOD*, pages 35–46, 1996.

[17] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Containment join size estimation: Models and methods. In *SIGMOD*, pages 145–156, 2003.

[18] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating answer sizes for XML queries. In *EDBT*, pages 590–608, March 2002.

[19] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *ICDE*, pages 443–454, March 2003.

[20] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, pages 425–436, 2001.