Tree-Pattern Queries on a Lightweight XML Processor *

Mirella M. Moro, Zografoula Vagena, Vassilis J. Tsotras

Department of Computer Science & Engineering
University of California
Riverside, CA 92521, USA
{mirella, foula, tsotras}@cs.ucr.edu

Abstract

Popular XML languages, like XPath, use "tree-pattern" queries to select nodes based on their structural characteristics. While many processing methods have already been proposed for such queries, none of them has found its way to any of the existing "lightweight" XML engines (i.e. engines without optimization modules). The main reason is the lack of a systematic comparison of query methods under a *common* storage model. In this work, we aim to fill this gap and answer two important questions: what the relative similarities and important differences among the tree-pattern query methods are, and if there is a prominent method among them in terms of effectiveness and robustness that an XML processor should support. For the first question, we propose a novel classification of the methods according to their matching process. We then describe a common storage model and demonstrate that the access pattern of each class conforms or can be adapted to conform to this model. Finally, we perform an experimental evaluation to compare their relative performance. Based on the evaluation results, we conclude that the family of holistic processing methods, which provides performance guarantees, is the most robust alternative for such an environment.

1 Introduction

The widespread employment of XML requires the development of efficient methods for manipulating XML data. Query languages, such as XQuery [4] and XPath [3], take into consideration the inherent structure of the data and enable querying both on its structure and on simple values. The most general structural constraints have the form of "tree-patterns". For example, consider the query:

//article[/author[@last="DeWitt"]]//proceedings[@conf="VLDB"]

that requests all proceedings of articles that have an author with last name "DeWitt" and have appeared in a "VLDB" conference. The query consists of two types of conditions:

- **@last="DeWitt", @conf="VLDB"**: are value-based since they select elements according to their values.
- **//article[/author]//proceedings**: defines structural constraints as it imposes restrictions on the structure of the retrieved elements (e.g. a **proceedings** element must exist under an **article** with at least one **author** element as child).

While value-based conditions can be efficiently evaluated with traditional indexing schemes, supporting the structural query part is a unique and challenging task. Recent work [2] has shown that specialized, set-based algorithms are advantageous over straightforward approaches (e.g. document navigation) or adaptations of relational techniques. Moreover, *holistic* processing techniques (in which a tree-pattern is handled as a *single* physical operator [5]), have outperformed more conventional solutions (i.e., approaches where the query is first decomposed using *binary* operators, and then some optimization is applied to produce an efficient plan [29]). As a result, numerous proposals attempt to handle tree-pattern queries with holistic techniques [5, 11, 28, 25, 32]. Additionaly, index structures have also been introduced [14, 23, 10, 7, 5, 17, 18, 8] to further improve performance.

A common characteristic for all holistic approaches is that some preprocessing is required, either on the data [5], or on the data and the query [28, 25, 32]. This requires the existence of a "dedicated" (native or relational) XML storage manager where all data resides and can be preprocessed. Note that this storage assumption is not restrictive since it is orthogonal to the way data is published (using DOM or SAX APIs). It should also be noted that the term

*Research partially supported NSF grant IIS 0339032, UC Micro, and Lotus Interworks; Mirella Moro was supported by Capes (Brazil).

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

**Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005**

"holistic" is overloaded in the literature. In [5], it implies global query matching, while in [28, 25, 32] the query is preprocessed as a whole (but the matching is incremental).

Despite the variety of holistic solutions with promising performance results for tree-pattern queries, none of them has found its way to today's XML engines. This includes both fully-fledged XML database systems as well as "lightweight" XML engines. For a fully-fledged XML DBMS, one could simply implement all tree-pattern approaches and then combine them under a cost-based optimization framework so as to choose the best approach. However, there are many data processing applications that use "lightweight" XML engines [12, 13] which do not contain a cost-based optimization module. When an optimizer is not present, an effective and robust processing method is necessary. Nonetheless, a survey among existing lightweight engines reveals that most of them employ *binary* operators (e.g. indexed nested loops joins [13], or the more specialized structural joins [12]) which are then combined with statically defined execution plans. Given the potential of holistic approaches, it is surprising that none of these lightweight processors has implemented any of them.

The main reason for this situation is the lack of evaluation of these methods under a *common storage model*. Although empirical results have already been published, comparing small subsets of the methods with typically few and small datasets, no complete comparison exists. More importantly, there is no reported work on the integration of all methods under the same storage model. Without such common integration, it is impossible to quantify the relative advantages and disadvantages of each approach.

In this paper, we attempt to fill this gap by integrating all existing tree-pattern query processing methods over stored XML data in a unified environment. We assume tree-pattern queries with XPath semantics, and we target environments where XML data are physically stored within data management systems and can be indexed at will. We also propose a method categorization based on two main features that differentiate the techniques: the data access patterns, and the matching process. In particular, our main contributions are summarized as:

- We introduce a clear categorization of methods for tree-pattern queries processing. We describe techniques proposed so far, discuss their characteristics, place them within our classification.

- We set up a unified environment under a common storage model and describe the integration of all methods within it. We employ a storage model that is simple but versatile enough to capture the access characteristics of each method, and permit clustering of data with the aid of off-the-shelf access methods (e.g. $B+$-trees).

- We propose novel variations of methods that can use existing index structures to their advantage, when such structures exist. We also show how to adapt methods not directly applicable to tree-pattern query processing in order to handle tree-pattern queries as well.

- We perform an extensive comparative study with synthetic, benchmark and real datasets. To our knowledge, this is the first complete performance study. Our results identify the behavior of each method under varying circumstances. Furthermore, they allow us to make generalizations and decisions in the applicability, robustness and efficiency of each method.

The rest of the paper is organized as follows. Section 2 presents background information and describes the employed storage model. Section 3 details the method categorization and, for each category, it presents our contributions (in terms of method integration, new method proposals or both). Section 4 presents an extensive experimental study and discusses our findings. Section 5 summarizes related work, and Section 6 concludes the paper.

2 Background

An XML database is usually modeled as a forest of unranked, node-labeled trees, one tree per document. A node corresponds to an element, attribute or value, and the edges represent immediate element-subelement or element-value relationships [3]. Figure 1(a) shows the tree representation of an XML database containing bibliographic entries. In this work, we consider the unordered model, i.e. the order in which the elements appear in the document is not relevant, conforming to the semantics of XPath [3]. Moreover, we concentrate on tree-pattern queries with descendant edges. Queries with child edges are generally answered using the same methods augmented with a post-processing step, which filters out results that are not in accordance with the child constraints.

2.1 Document Encoding

A numbering scheme [9, 2, 7, 5, 28, 15] is usually embedded in the tree representation of each document node to efficiently capture the structural relationships among nodes. Common numbering schemes are *range-based*, and a range $(left, right)$ is assigned to each node in the document tree. Given a node pair (a, d), node d is a *descendant* of node a, if and only if, $a.left < d.left < d.right < a.right$. When queries with child constraints are involved, the level of the node within the tree is also used to capture parent-child relationships.

2.2 Storage Model

We regard the input (i.e. XML documents) as sequences (lists) of elements. There is one sequence per document tag, called *element list* from now on. The numbering scheme (described in Section 2.1) creates the mapping between the document tree structure and the element lists. Suitable indexes on the element lists define the necessary node clusterings and/or orderings. We chose to represent the input as (possibly indexed) element lists because (a)

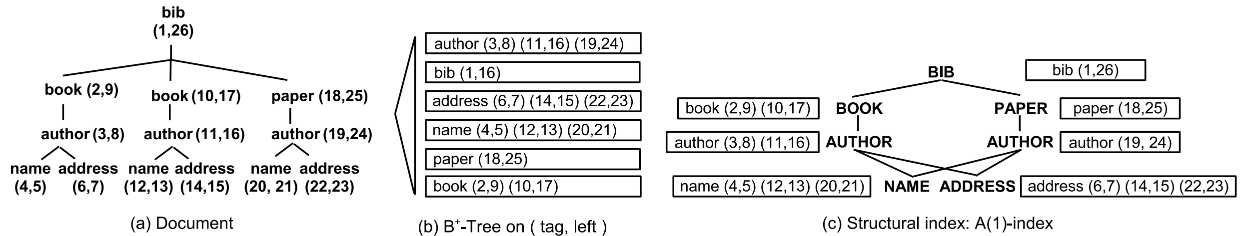


Figure 1: Storage Model: an XML document, its element lists stored in B^+-tree, and structural indexes

each method under consideration expects (or can be trivially adapted to expect) its input in that way; and (b) the model is versatile enough to permit effective node clustering through the use of common index structures. For example, Figure 1(b) shows the element lists indexed by (tag,$left$) in a B^+-tree. In Figure 1(c), a structural index partitions the data based on bisimilarity (as it will be explained in Section 3.4). If different clusterings are desired (e.g. clustering children of each node together, as in [28]), other indexing techniques can be easily applied.

3 Method Categorization

While examining the methods for tree-pattern matching, we discovered that they differ in two main features, namely: data access patterns, and matching algorithm. Considering these features, we define four categories whose characteristics are summarized in Table 1.

Category	Access pattern	Matching process
1. Set-based	Sorted/indexed	Join sorted input, merge individual paths
2. Query driven	Indexed/random	Incremental construction of each result instance
3. Input driven	Sequential	Input drives computation, merge individual paths
4. Structural summaries	Sequential/random	Merge-join partitioned input, merge individual paths

Table 1: Summary of categories features

Specifically, Section 3.1 presents Category 1, which comprises holistic *set-based* techniques [5, 21, 6]. Their input consists of element lists sorted on *left* and possibly indexed to improve performance. In addition, various indexing approaches have been proposed for set-based techniques [5, 7, 17] and are discussed in Section 3.1.1.

Category 2 consists of *query driven* methods, in which the query defines the way the input is probed, and is presented in Section 3.2. The main approaches in this category are ViST [28] and PRIX [25]. These methods convert the XML document into a sequence, and the query evaluation is reduced to subsequence matching. This subsequence matching can be treated as a specific static plan for evaluating an index nested loops join. However, a static plan cannot always give a good probe ordering. Based on this observation, we consider a dynamic approach which would be clearly advantageous. In order to be more general, we implemented an index nested loops join (INLJ) that considers all left-deep plans, as detailed in Section 3.2.1. We also show how a B^+-tree may be employed for skipping both descendants and ancestors. Furthermore, our experiments show that the performance of techniques in this category depends heavily on the data and queries.

Category 3, presented in Section 3.3, consists of the *input driven* methods in which, at each point in time, the flow of computation is guided by the input. This category is inspired by navigational approaches that identify the answer by directly navigating the tree structure of the input [32]. We further enhance their performance by adapting ideas from streaming environments [11] and employing Finite State Machines to keep partial matches. Hence, we propose a new method (called SINGLEDFA) as a representative of this category in Section 3.3.1. Furthermore, we show how this method can be adapted when an index is present (called IDXDFA) taking advantage of our environment (where data is stored and can be easily indexed).

Finally, Category 4 (*graph summary evaluation*) contains all methods that use a structural summary of the data. In Section 3.4, we show how these techniques can be adapted to work on our storage model, by combining a structural summary with a set-based solution.

3.1 Category 1: Set-based Techniques

The major representative is a stack-assisted XML pattern matching algorithm, called TWIGSTACK [5]. Each node q in a tree-pattern query is associated with an element list T_q that has all document nodes with tag q sorted by *left* position (i.e. the *left* value of the region-based code). Also, each node q is associated with a stack S_q, which stores pairs (*node positional representation in* T_q, *pointer to a node in* S_{parent_q}). This stack system enables the compact encoding of (a possible larger number) intermediate partial results, which is its main advantage.

Using the range-based numbering scheme, the structural constraints are converted to θ-joins. Then a holistic, merge-based method is utilized to process the tree-pattern query (by first sorting the element lists on the left position of each element). Any backtracking is eliminated by buffering elements that are bound to be used again in the future.

Figure 2 shows an example of the stacks usage. Figure

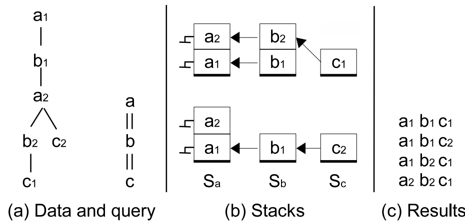


Figure 2: Compact encoding of results using stacks

2(a) has a section of an XML document (a_1 and a_2 correspond to nodes of tag a, etc) and a query for all nodes c that are descendant of b which are descendant of a. Figure 2(b) illustrates the content of the stacks in two different moments: when c_1 is read and when c_2 is read. While c_1 has b_2 as parent (with a_2 as its parent), c_2 has b_1 as parent (with a_1 as its parent). Figure 2(c) shows the query results.

The TWIGSTACK algorithm works in two phases: (a) some solutions to individual root-to-leaf paths are computed and stored on the stacks, and (b) these partial solutions are merge-joined to compute the results for the tree-pattern query. Hence, the output is produced in root-to-leaf sorted order. It is important to note that before a node is considered for further processing, the algorithm ensures that (a) the node has a descendant on each of the lists corresponding to its child nodes in the query, and (b) each of those descendant nodes recursively satisfies this property. Thus, the algorithm can guarantee worst-case performance linear to the query input and the size of the result. This is an important property and it contributes for the robustness of the techniques belonging in this category in comparison with other categories, as it will be illustrated in the experimental section.

3.1.1 TwigStack with Indexes

In order to skip elements that do not participate in the results, indexes may be used.

B^+-tree. A B^+-tree built on the *left* attribute for improving structural join processing is proposed in [7]. When an ancestor node is accessed, the B^+-tree of the descendant list can be probed to resume search on the first node with left position larger than the position of the ancestor node. The same technique can be used in the TWIGSTACK algorithm in order to skip nodes that belong to descendant lists and reside before the current node. Ancestor skipping is not that effective since the index can skip only up to the first element following the given one.

XR-Tree. The XR-TREE [17] indexes element nodes on their region-based codes, the (*left*, *right*) pairs, and it is basically a B^+-tree with complex index key entries and stab lists associated to internal nodes. Given a key k and an element E with region (*left*, *right*), k stabs E if $left \leq k \leq right$. The stab list of key k stores the elements that are stabbed by k, and there is no other key in ancestor nodes of the tree that stabs them. The main weakness of this approach is its inability to handle recursive ancestor elements efficiently (when a node happens to be ancestor of two or more other nodes, it will be searched for and retrieved as many times as the number of its descendant nodes).

XB-Tree. A variant of TWIGSTACK algorithm, the XBTWIGSTACK, uses XB-Trees to speed up processing. The nodes in the leaf pages are sorted by their *left* values. The difference between XB-Tree and a B^+-tree is in the data stored on internal pages: each node has a bounding segment (*left*, *right*) and a pointer to its child page, whose nodes have bounding segments completely included in (*left*, *right*). This structure provides more efficient search for ancestor elements than XR-tree, as it enables identification of nodes starting at any node in the tree, not only the root one. Searching for descendants proceeds the same way as in B^+-tree.

Discussion. A study presented in [19] compares the performances of these indexes when evaluating structural joins, i.e. the base query that returns pairs of ancestor/descendant or parent/child elements. In summary, when skipping ancestors is necessary, XBTWIGSTACK performs better than XRTWIGSTACK since the size of the XB-Tree is smaller than that of XR-Trees (because of the stab lists that the latter maintains). Moreover, searching within stab lists in the XR-tree is less efficient than any index searching. Hence, XBTWIGSTACK also outperforms XRTWIGSTACK when there are recursive levels of ancestors. A plain B^+-tree with TWIGSTACK skips descendants as effectively as XBTWIGSTACK but is not as efficient in skipping ancestors. We performed an evaluation of these indexes for general tree-pattern queries and achieved similar observations (results not presented for lack of space). Since the XB-tree provides better performance while skipping both descendants and ancestors, we implemented it as our choice of index on TWIGSTACK when comparing set-based methods with the other three categories.

3.2 Category 2: Query Driven Techniques

This category is inspired by methods where the form (tree structure, node labels) of the query guides the matching process. Two methods fall into this category, namely VIST [28] and PRIX [25]. Although the specific details differ significantly, both methods use the same strategy. Each XML document is converted into a sequence. The sequencing process (*preorder traversal* for VIST and *prüfer construction* for PRIX) guarantees that a unique sequence is created for each XML tree. Next, each query is also converted into a sequence using the same sequencing process. Therefore, answering a query is mapped to subsequence matching, i.e. to the identification of the (non-contiguous) instances of the query sequence within the document sequence. A post-processing step, which can be performed on the fly ([25]), is necessary to filter out results that do not correspond to witnesses of the original query. Figure 3 illustrates the algorithms.

The heart of those methods is the subsequence matching module. The straightforward solution, which recursively

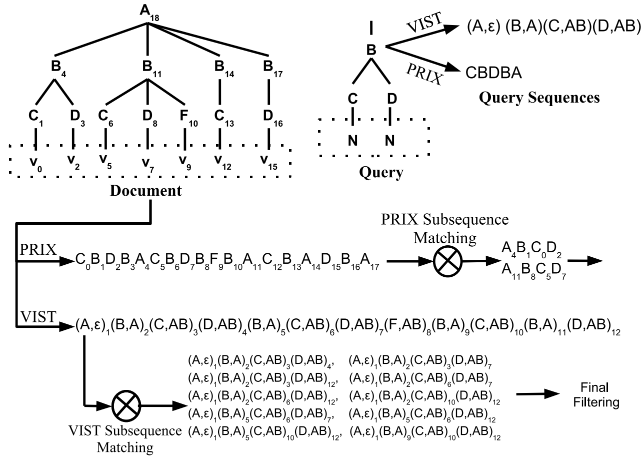


Figure 3: Tree-pattern matching by subsequence matching

identifies matches for each node within the query sequence in order, requires *quadratic* time in the document size and therefore becomes not competitive. Because there is no additional information, every input node yet to be visited has to be considered as candidate for matching, at each step of the matching process. Methods in this category attempt to optimize the naive algorithm by considering the special tree structure of the document and the query from which the sequences are derived. In particular, (a) they identify the candidate nodes for each matching step, and (b) they use index structures to cluster those candidates together. For instance, in Figure 3, when VIST has identified a match for the portion of the query subsequence ending at node (B, A), which corresponds to query node B, the structure of the query implies that only children nodes with label C have to be accessed. To optimize such access, it employs a B^+-tree which clusters document nodes first on label and then on the root-to-node paths. That way, each matching step (for a child query node) involves a B^+-tree probe and range search. Similar functionality is provided by PRIX.

Discussion. The previous description reveals that the subsequence matching process can be regarded as a plan consisting of indexed nested loop joins among relations, each of which groups document nodes with the same label. For a given query, the sequence of the joins is *statically* defined by the sequencing of the query. A natural question involves the efficiency of these techniques in comparison with a pure indexed-nested loops join plan whose order is achieved, for example, in a cost-based fashion. A relevant issue is raised in [27] where the authors consider XML schema or data distributions while defining the sequencing order for a query. Taking these issues into consideration, we decided to identify all *left-deep* plans for each query and report their performance in an attempt to draw general conclusions on the behavior of this category in comparison with other categories. Next, we describe the method we employed (INLJ) in detail. Clearly, the INLJ plans are a superset of the static plans that PRIX and VIST use.

3.2.1 Index Nested Loop Joins, INLJ

In order to be able to employ INLJ on the relations that group document nodes with the same label, we need to index the relations such that retrieving elements that satisfy a particular structural constraint (parent-child or ancestor-descendant) is efficient. We focus on the *ancestor-descendant* constraints and summarize the differences to support the *parent-child* ones.

Document nodes are first grouped by node label (i.e. all nodes with the same label belong to the same set) and the numbering scheme in Section 2 is applied. Subsequently, any of the indexes presented in Section 3.1 that supports identification of both ancestors and descendants (e.g. XR-Tree) can be utilized to index each of the sets. With those structures, the evaluation of a tree-pattern query resembles the evaluation of a relational plan consisting only of index nested loops joins among the element sets. As in the relational case, the problem is how to decide the order in which the joins are performed. In our experiments we report the left-deep plan that gives the best performance.

Another issue is the presence of a more robust index within a lightweight processor. In the absence of such an index and assuming that only B^+-tree structures exist, we propose a simple but efficient technique to identify ancestor nodes (retrieval of descendant nodes is very efficient, as it entails a simple range query [7]). Starting from a node x with label b. In order to identify its ancestors of label a, probe the B^+-tree to determine the first node in the a relation with the smallest *left* position which is larger than the *left* position of x. The B^+-tree clustering ensures that any ancestor node resides at previous positions within the leaves of the index. As a result, a backwards range search identifies all of them. In order to know how far back to search, we also keep in each node record, the *left* position of the topmost ancestor (within the XML document) with the same label as the node. For instance, node b_2 in Figure 2 keeps the *right* position of b_1. So, when c_1 requires its ancestors, the search starts in b_2 and ends at b_1, i.e. any other previous node is not evaluated.

As described, our technique presents important advantages over the proposed sequencing methods, namely independence of the ordered XML model and the total avoidance of false positives. Precisely, in the case of the unordered model, both VIST and PRIX need to invoke a potentially large number of sequence matching queries and then union the results. The same can happen if the query contains siblings with the same label. All these problems are avoided with our technique.

In the case of *parent-child* constrains, index structures like the one employed in VIST can be easily incorporated to identify candidate nodes. Nevertheless, the processing method remains unchanged.

3.3 Category 3: Input Driven Techniques

While the matching was driven by the query in the previous category, we considered having a method that is input driven. Typically, this appears in navigational meth-

ods [1, 32]. The first navigational methods (as presented in Lorel [1]) followed the edges of document tree in order to process the query, without keeping any intermediate state. This access pattern may be efficient for parent-child queries, but when there are ancestor-descendant edges, it starts to lose performance and there is no way to keep partial results. A similar problem, where keeping partial results was needed, appeared when filtering streaming documents over a collection of tree-pattern queries (e.g. [11]). In the filtering problem, the query is decomposed into its constituent paths and each path is processed through a FSM. The FSM stores partial results as the document is parsed sequentially (in document order). At each point, partial or total pattern matching is performed, depending on the existing partial matches and the current node. We have thus decided to combine navigational probing with FSMs and present a new method (SINGLEDFA) for this category.

The main advantage of this category is its simplicity, as well as its sequential access pattern. Moreover, a node is processed exactly once. On the other hand, with sequential access there is little ability to avoid visiting nodes that do not participate in the result. This might have a negative impact, if the tree-pattern query is very selective. Since in our environment data is already stored and can be easily indexed, we also present an improved FSM approach (IDXDFA), that utilizes an index and avoids visiting nodes that do not participate in the result.

3.3.1 A new FSM method

In our technique, the state of partial matches is encoded with the FSM that corresponds to the query. Documents are parsed one tag at a time. The *start* labels (reading *<element>*) trigger the events in the FSM, which chooses the next state according to the element read. When an *end* label is found (reading *</element>*), the execution backtracks to the state it was in when the corresponding *start* was processed. A run-time stack keeps the states reached and allows such a state backtracking.

Tree-Pattern Matching. In order to print matches and present the results in root-to-leaf order, we extended the mechanism proposed by [5]. The tree-pattern is processed whenever a pattern query root is to be popped from the element stacks (i.e. after processing all its descendants).

This implementation, called SINGLEDFA, keeps the advantage of the [5], i.e. intermediate results are compacted in the stacks. We use a DFA (Deterministic State Machine) as FSM, and the advantages are twofold: it speeds up the computation, and it ensures that only structure matched elements are stored in the stacks. Regarding the automaton unnesting, i.e. conversion of NFA to DFA, we considered the ideas from [20, 15] so that it has a superior performance since the DFA avoids the backtracking that the NFA computation entails.

Speeding Up the Navigation. Our experiments show that simply reading the whole input and processing it through a FSM does not give a competitive performance. This happens because all possible result candidates need to be kept in the stacks even though they may not participate in any partial matching. Hence, we propose an improvement, called IDXDFA, in order to speed up processing by using the input stored in an index structure. Instead of reading the whole input sequentially, we use indexes in order to skip some descendants that do not participate in any result. Figure 4 details the general algorithm.

```
Input: q is the query tree
       R is the query root element list
Function executeFSM()
1. currentNodes = searchDesc(R.firstElement)
2. while currentNodes.size > 0      // elements to be processed
3.   m = getMinimum(currentNodes)   // next element in order
4.   processElement(m)              // provide m to the FSM
5.   currentNodes += M.getNextElement // update elements
6.   if stack[R] empty              // update all current nodes
7.     currentNodes = searchDesc(currentNodes[R])
8.   else for each q.child c of R // update descendants
9.         verifyNode(c)
Function searchDesc (node n)
10. if n is q.leaf return n
11. else result = n
12. for each q.child c of n
13.   p = probe C for n            // return c with c.left > n.left
14.   result += searchDesc(p)
15. return result
Function verifyNode (node n)
16. if n is q.leaf return
17. else if update(n) return      // if n.subtree updated, skip
18.      else for each q.child c of n
19.             verifyNode (c)  // update subtrees of n.child
Function update (node n)
20. if stack[N] empty                // it may skip descendants
21.   currentNodes = searchDesc(n)   // update subtree
22.   return true
23. else return false
```

Figure 4: General algorithm for IDXDFA

The variable *currentNodes* has one entry per query element and stores the next nodes to be read from each element list. This variable is initialized with the function *searchDesc*, which probes all lists starting from the query root element list (the only list to be sequentially read). Then, *m* points to the node with minimum left position, which is the next to be processed. After *m* is processed through the FSM, its element list *M* is advanced, and *currentNodes* updated with the element after *m* in *M*. The stack mentioned in line 6 is based on the ones presented in TwigStack. When the query root stack (or any of its children stack) is empty, it is time to update all its descendants in order to skip those that do not belong to any partial match, since they will not have a valid ancestor. Finally, functions *verifyNode* and *update* work together with *searchDesc* to check when descendants of internal nodes (not query root) may be skipped.

IDXDFA keeps the features of compact intermediate results and in-order traversal of the documents. However, when there are some specific opportunities, it skips descendants. For instance, consider the example tree illustrated in Figure 5 as input for the query $a//b[//c]//d$. The triangles in the figure represent subtrees that do not have any element with tag *a* (query root). All elements in the grey regions are not processed by the FSM in IDXDFA, due to

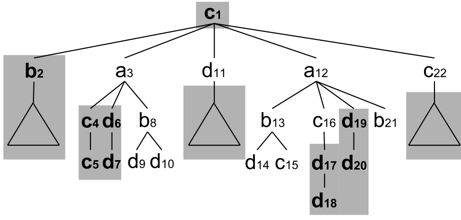


Figure 5: Example of skipping descendants in IDXDFA

the new functions added to the regular FSM processing.

Specifically, line 1 skips all nodes before the first instance of the query root (i.e. nodes prior to a_3). Starting from a_3, *searchDesc* gets *b8*, and from that, it probes for *c* and *d* (skipping nodes c_4, c_5, d_6, d_7). When d_{11} is processed through the DFA, all stacks are cleaned. Then, lines 6 and 7 provide that all nodes between d_{11} and a_{12} be skipped. Likewise, after processing c_{16}, *verifyNode* is called from b_{21}, skipping nodes from d_{17} to d_{20}. Finally, this situation happens again when processing c_{22}.

3.4 Category 4: Structural Summaries

The methods in this category utilize a structural summary of the data. The structural summary is a graph-structured index (usually smaller than the original document) that maintains all the structural characteristics of the data. Each index node represents a group of nodes in the original document. In order to keep the size of the index small enough to reside in main memory, an approximate index is constructed by relaxing some of the structural constraints. As a result, it may have false positives.

The tree-pattern matching proceeds in two phases. In the first phase, the structural summary of the data is utilized to identify the index nodes that satisfy the query. Those nodes may represent a superset of the original document nodes which participate in the query. Thus, a post processing step is necessary to filter out the false positives and identify the actual result nodes.

As of today, only navigation-based techniques have been reported to perform the matching on the index graph [18]. The structural index access pattern is hard to analyze and model, as little can be speculated about the clustering of the nodes. However, considering the fact that the index structure fits in main memory, we regard this cost as an index parameter and we investigate the cost of the second, disk-based phase, for different levels of structural relaxation. For the purposes of the comparison, we employed the family of techniques based on bisimilarity [23, 10, 18].

3.4.1 Query Evaluation using Structural Summaries

Recently proposed structural summaries [23, 10, 18] are based on the concept of bisimilarity. They partition the data nodes into groups, called *extents*. These groups are represented by index nodes such that only bisimilar nodes are represented by the same index node. The notion of bisimilarity is defined as:

Definition 1 *Let $G = (V, E)$ be a directed graph. A symmetric, binary relation on V is called (backward) bisimulation if, for any two data nodes $u \in V$ and $v \in V$ with $u \approx v$, we have: (a) u and v have the same label and (b) if u' is a parent of u, then there is a parent v' of v such that $u' \approx v'$ and vice versa.*

The above definition creates an index graph which is both *safe* and *precise*, i.e. it produces the exact results for a particular structural query. However, in several cases, the size of such graph can be large and sometimes comparable to the size of the original document, resulting in decreasing performance. As a solution, Kaushik et al. [18] proposed to encode paths up to a certain point, instead of all possible paths. That way, the index is still safe, as it does not miss any results of a structural index. However, for paths larger than the predefined size, it is no longer precise, and it returns a superset of the query output.

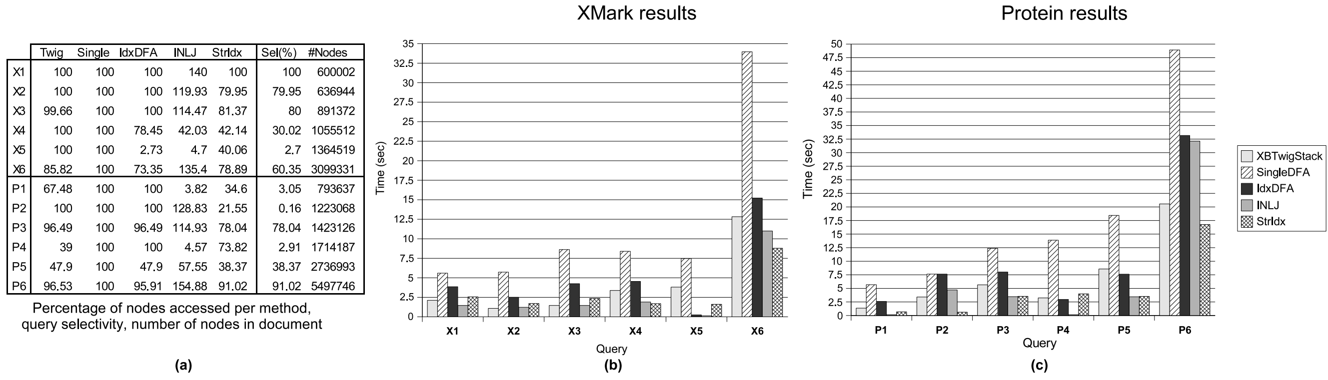
Tree-pattern query evaluation using those relaxed graphs proceeds as follows. The (memory resident) index structure is probed to identify the index nodes that represent the candidate document nodes. Then, the corresponding data extents are retrieved, and one of the previous query matching techniques is employed to perform the final filtering. In our experiments, we use TwigStack algorithm (from Section 3.1) to perform the matching. This choice is justified by its superiority identified while comparing all previous methods. To be able to use TwigStack, we maintain the extents of the index nodes as partitions of the element lists, sorted on the *left* value of the numbering scheme (Section 2.1). We call the new method that combines TwigStack over a structural index STRIDX.

4 Experimental Evaluation

In order to verify the relative performance and robustness of the tree-pattern query processing algorithms, we ran a wide range of experiments with all algorithms considered in this work. Specifically, we implemented algorithms: XBTWIGSTACK [5], our SINGLEDFA and IDXDFA, our INLJ with ancestor information, and STRIDX. We also implemented VIST [28] and PRIX [25], but as discussed in Section 3.2 and empirically shown in Section 4.4, their behavior can at most match the best plan of INLJ.

First, we present experiments with real datasets in order to get some estimates about the time required by each algorithm to compute the queries. Later, we add synthetic datasets to further analyze each algorithm. In such scenario, we are able to characterize the algorithms according to specific features available in each custom dataset. Finally, we have more sets of experiments in order to closely verify the performance of XBTWIGSTACK versus INLJ.

The performance measure is the total (system and user) time required by each algorithm to compute the query. We also measured the total number of input nodes accessed by



	Twig	Single	IdxDFA	INLJ	StrIdx	Sel(%)	#Nodes
X1	100	100	100	140	100	100	600002
X2	100	100	100	119.93	79.95	79.95	636944
X3	99.66	100	100	114.47	81.37	80	891372
X4	100	100	78.45	42.03	42.14	30.02	1055512
X5	100	100	2.73	4.7	40.06	2.7	1364519
X6	85.82	100	73.35	135.4	78.89	60.35	3099331
P1	67.48	100	100	3.82	34.6	3.05	793637
P2	100	100	100	128.83	21.55	0.16	1223068
P3	96.49	100	96.49	114.93	78.04	78.04	1423126
P4	39	100	100	4.57	73.82	2.91	1714187
P5	47.9	100	47.9	57.55	38.37	38.37	2736993
P6	96.53	100	95.91	154.88	91.02	91.02	5497746

Figure 6: Results for XMark and Protein datasets

each algorithm, as well as the number of page I/Os. We do not include all graphs for these last two measures for lack of space; however, the observations made (based on them) helped us explain the different results we report, and we refer to them when needed.

The system setup was as follows. All the experiments were conducted on an Intel Pentium 4 2.6GHz processor machine, with 1GB of main memory. Every algorithm was implemented in C++ and compiled with gcc version 3.3, optimized with the -O2 flag. All techniques were implemented on BerkeleyDB. The techniques that need indexes were implemented using the BerkeleyDB B^+-tree access method. Finally, Berkeley DB was setup to use 100 buffers, with page size of 8KB (therefore, the amount of memory used is uniform and limited for all algorithms).

4.1 Experiments on Benchmark and Real Data

We start with queries on data generated by the XMark XML benchmark [30], which models data from an Internet auction, and the Protein Sequence Database [31]. For the XMark, in particular, we used the database generated with a scale factor of 100, creating a dataset with 1.4GB of raw data (almost 17 million text nodes). As the Protein dataset requires no particular pre-processing, we used the original dataset presented in [31]. In both datasets, the numbering scheme was added by parsing the XML file with an event-based XML parser that conforms to the SAX [26] interface which we implemented in Java. Figure 7 illustrates the tree structures of a representative set of the queries we considered (other query structures produced similar results).

Figure 6 illustrates the results for the benchmark datasets. Queries X1 to X6 were executed on XMark data, with results presented in Figure 6 (b). Queries P1 to P6 were performed on Protein, with results presented in Figure 6(c). Furthermore, the queries are presented ordered by their dataset size, as informed in Figure 6(a).

It is possible to observe that INLJ (only the performance of its best plan is presented) has a consistent advantage when processing the smaller datasets of XMark (X1 to X5). Since most data fit in main memory, it does not need extra

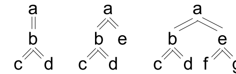


Figure 7: Queries used on XMark and Protein

I/O operations which are expensive due to its random access. On the other hand, XBTWIGSTACK incurs the overhead of checking for node participation in the final result. Finally, the cost of INLJ random access becomes apparent in larger datasets, as X6, when STRIDX is more efficient.

Note that there is no steady classification pattern followed by all algorithms in the XMark dataset. For example, X1 has INLJ in first place followed by XBTWIGSTACK, while X4 and X6 have STRIDX in first, followed by INLJ and XBTWIGSTACK. A similar situation is observed in the Protein dataset, when there is no general pattern (involving all techniques) in the results as well.

Although there is such a variation in the results, the only fixed position is the last one, granted to SINGLEDFA. Hence, we exclude its results from our next graphs as it is always the worst performance, regardless of the query or the dataset. Additionally, the remaining four algorithms alternate their relative performances depending on the query.

We can also affirm that STRIDX is at least as good as the algorithm used in its post processing step. Considering the algorithms presented, any of them can process the query over the structural index. STRIDX provides a very good performance when the index selectivity is close to the query selectivity. Hence, the next sets of experiments focus on the performances of XBTWIGSTACK, IDXDFA, and INLJ, in order to achieve a more complete classification.

4.2 Experiments with Custom Data

In this section, we investigate the performance of the algorithms for different structural characteristics of the input data as well as with varying structural join selectivities (i.e. join selectivities among the pairwise joins that constitute the tree-pattern query).

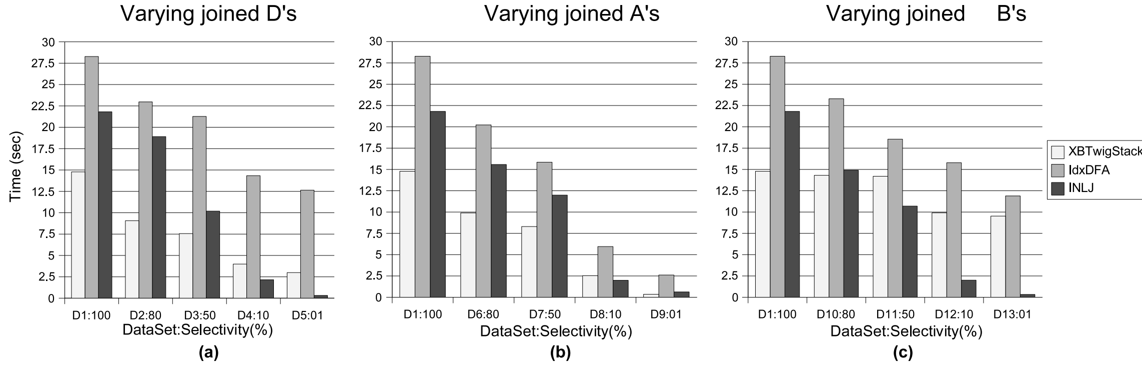


Figure 8: Performance when varying structural join selectivity (i.e. selectivity equal to 80% means that 80% of all nodes in the dataset belong to the result, and so on).

4.2.1 Custom Data Generation

We chose a number of parameters that identify the behavior of the presented techniques considering the features that we investigate (i.e. data access and matching process). Although existing benchmarks can be used to provide general insights for the performance of the algorithms, none of them managed to effectively isolate the features that we are interested in. Therefore, we evaluate the algorithms on custom data, where we explicitly vary the necessary parameters. Here, we consider the query: $//a//b[//c]//d$. The reason for this choice is twofold: (a) this query is simple enough to permit the detailed investigation of the processing that occurs for each of the query nodes, and (b) it has the required complexity to allow inspecting a large number of different data access possibilities (varying join selectivity considering the number of root, internal nodes and leaves separately), as it becomes apparent in the next section. In the following paragraphs, we explain how we varied the selectivity such that this single query is enough to show the behavior of each algorithm.

We first varied the number of input elements that participate in the result. We started with data set $D1$, where each input node participates in exactly one query result. From the 100% participation, we gradually and uniformly removed elements until we reached 1% participation (we considered 80%, 50%, 10%, and 1%, i.e. 80% of all nodes belong to the result and so on). To better understand the behavior of the algorithms, we decided to change the stream of only one query node at a time. Hence, our experiments are based on the benchmark idea of isolating and evaluating each feature separately. In particular, we varied the number of joined elements in the stream corresponding to the query node d (datasets $D2$ to $D5$), a (datasets $D6$ to $D9$) and b (datasets $D10$ to $D13$). The structural characteristics of these datasets had no repetition or recursion among the input data. Each input stream has around 1 million elements, i.e. the total input size is around 4 million nodes.

We proceed with varying the structural characteristics of the data by adding recursive nodes. The depth of the recursion was controlled from 2 to 10 elements. We generated datasets with either a or d as the recursive element. The choice of those elements was coupled with varying the selectivity of other elements, in order to investigate the effect of indexed access on recursive elements. Dataset $D14$ contains recursive d elements and 100% participation of all elements in the join result, while datasets $D15$ and $D16$ contain a varied degree of b elements (50% and 1% are the variations of elements in these sets of experiments), and datasets $D17$ and $D18$ contain a varied degree of a elements. Datasets $D19$-$D23$ have a as the recursive element. $D19$ has 100% partitipation of all elements, $D20$ and $D21$ vary the participation of b, and $D22$-$D23$ vary the participation of d in the result.

4.2.2 Experimental results from Custom Data

In this section, we report the results of the experiments when we vary the join selectivity and the structural characteristics of the input data.

Varying the number of joined root, leaves and internal nodes. The results for these experiments are presented in Figure 8, where (a) shows the results when varying the query leaf, (b) when varying the query root, and (c) when varying the query internal node. When varying each element list at a time, the experiments show a pattern depending on the query selectivity. The pattern is XBTWIGSTACK followed by INLJ and IDXDFA when selectivity is bigger than 50%. XBTWIGSTACK is faster because its sequential processing outperforms the random I/O performed by INLJ. Moreover, although IDXDFA is able to skip some descendants, it still stores unnecessary nodes as partial matches, which is better than the previous SINGLEDFA but not as good as XBTWIGSTACK. Finally, as the number of elements in the results decreases, INLJ starts to improve performance, becoming the first place when less than 10% of nodes belong to the result. In these cases, its optimal probe order compensates for its random I/O.

Varying the input structure. In this group of experiments, we added recursion to input data and then per-

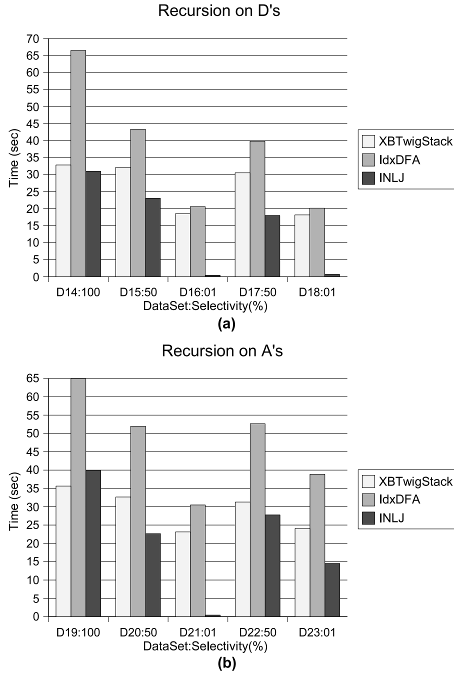


Figure 9: Performance when adding recursion

formed similar measures. Figure 9 presents the results. The conclusions we make are similar to the corresponding previous experiments with no recursion. We would expect that recursion hampers INLJ performance because of probing for the same nodes many times (e.g. multiple ancestors with same descendants and probing for descendant elements) as well as its random access. However, the optimal probe ordering that we chose compensates this fact by accessing only the nodes that participate in final results. On the other hand, XBTWIGSTACK incurs the overhead of visiting a large number of nodes that do not belong to any result (although it does not take them in consideration afterward). IDXDFA is always the slowest among these three algorithms, hence we discard its results from the next set of experiments and concentrate on the comparison of XBTWIGSTACK and INLJ.

Number of nodes. In order to have a quantitative idea about how many nodes the algorithms skip, Table 2 presents the number of nodes (in percentage) processed by each approach. Note that these values do not reflect the number of nodes stored as partial matches.

4.3 XBTWIGSTACK **versus** INLJ

In this section we present some interesting comparisons between XBTWIGSTACK and INLJ over random datasets. The query is still the same $//a//b[//c]//d$.

Running on large dataset. In this group of experiments, we evaluate the performance of both algorithms in a worst case scenario where the number of results in the query is very small, while the number of results for any subquery is large. This is the case where conventional query optimization techniques (which would try to identify the

Dataset	XBTwig	IdxDFA	INLJ
D1:100	100	100	100
D2:80	98.95	100	84.20
D3:50	92.88	100	57.18
D4:10	73.87	100	12.90
D5:01	67.43	100	1.32
D6:80	96.86	88.4	84.18
D7:50	78.58	64.31	57.17
D8:10	21.59	15.79	12.89
D9:01	2.32	1.66	1.33
D10:80	97.88	93.67	84.20
D11:50	85.68	78.54	57.09
D12:10	47.81	44.89	12.96
D13:01	34.89	34.56	1.64

Dataset	XBTwig	IdxDFA	INLJ
D14:100	100	100	100
D15:50	65.68	62.55	53.17
D16:01	14.56	14.43	1.12
D17:50	62.52	56.25	59.36
D18:01	1.52	1.25	1.72
D19:100	100	100	100
D20:50	93.76	90.63	53.17
D21:01	73.88	73.75	1.13
D22:50	96.88	100	59.39
D23:01	86.95	100	1.28

Table 2: Percentage of nodes processed in custom datasets

best execution plan by decomposing it into subplans) would fail. The reason for this is because the number of necessary intermediate results that has to be produced is large, and a technique that depends on such type of processing would have to produce all of them.

Figure 10(a) illustrates the results of XBTWIGSTACK (horizontal line) and all left-deep plans of INLJ (columns) performed in one single dataset with 40 million nodes (almost 1GB stored in B^+-trees). The selectivity of the query is around 1% (practically 400 thousand nodes in the result). As the results show, XBTWIGSTACK outperforms all INLJ plans. Hence, even if an optimizer module existed to provide the best order for probing the joins, its results would be meaningless since XBTWIGSTACK outperforms INLJ in all the possible left-deep query order. Furthermore, note that there is a difference of about 40 seconds (around 30%) from XBTWIGSTACK and INLJ best plan.

Running on random datasets. Figure 10(b) illustrates the results of XBTWIGSTACK and all left-deep plans of INLJ performed in ten random datasets whose sizes varies from 9 million to 25 million nodes. XBTWIGSTACK is uniformly better than all INLJ plans, with the exception of datasets R4 and R7 in which only the best INLJ plan has better or equal performance to XBTWIGSTACK. In each dataset, the difference between the best and worst plans vary in a range from 4 seconds (R3) to 236 seconds (R10). Furthermore, there is no single plan that has the best or one of the three best times over all datasets.

The results from this random datasets reinforce the robustness of XBTWIGSTACK. Most of the time (8 out of 10), XBTWITSTACK clearly outperforms the best plan of INLJ. This fact leads us to conclude that XBTWIGSTACK should be seriously considered as the algorithm for evaluating tree-pattern queries on lightweight processors.

4.4 VIST **and** PRIX **versus** INLJ

For completeness, Table 3 reports a representative set of results comparing INLJ, VIST, and PRIX for various of the custom datasets. This table depicts the percentage of nodes processed by each method. For datasets where the leaf elements are more selective (D2:80 and D5:01) PRIX is faster than VIST. The opposite holds when root elements are more selective (D6:80 and D9:01), while both methods

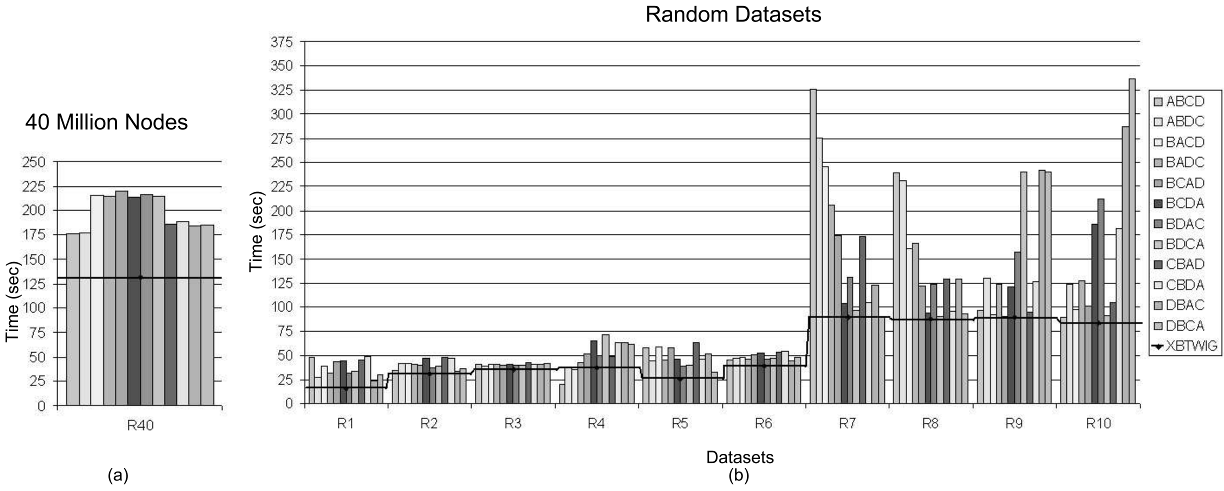


Figure 10: Random datasets comparing XBTWIGSTACK and INLJ

Dataset	VIST	PRIX	INLJ
D1:100	100	100	100
D2:80	100	84.23	84.20
D5:01	100	1.33	1.32
D6:80	84.22	100	84.18
D9:01	1.33	100	1.33
D10:80	89.48	89.49	84.20
D13:01	34.24	34.22	1.64

Table 3: Percentage of nodes processed by VIST, PRIX, and INLJ

have similar performances when the internal nodes are selective (D10:80 and D13:01). Nevertheless, the best plan from INLJ has always similar or better performance than the best between VIST and PRIX.

4.5 Discussion

In conclusion, we have identified the following choices while evaluating different algorithms for tree-pattern queries on lightweight processors:

1. When a structural index is available for the query such that it covers only nodes in the result, STRIDX is generally faster than the other approaches. This happens because the structural index size is much smaller than the actual data size and, consequently, the number of nodes to access is also smaller. Moreover, the controlled data access does not entail random I/Os as those performed by INLJ.

2. In the absence of such an index, XBTWIGSTACK is the most robust and predictable solution. Although INLJ may present better result when the selectivity is small, there is no guarantee that the chosen plan is the best without having an optimizer module to calculate that. As an optimizer is usually not available in lightweight processors, then XBTWIGSTACK should be the preferred choice, as reinforced by our experiments with random datasets.

3. The experiments also showed that the DFA-based solutions usually have the worst performance. Even though we improved their general performance by adding index access to the DFA in IDXDFA, it is not enough to outperform XBTWIGSTACK.

5 Related Work

Early studies on structural selection over XML data regard the simpler problem of matching single path patterns consisting of two (structural joins) or more nodes (path joins). All these methods operate over the sequences of nodes with the same tag and perform the matching in a merge fashion. Among them, the stack-based ones proposed in [2] have been shown to achieve superior performance.

There is an abundance of techniques that employ Finite State Machines to answer tree-pattern queries over streaming data [16, 24]; however, their main memory requirements are large. As a result, they are not scalable to large databases and, hence, they are excluded from our study. Instead, as a representative of this category, we extended and modified the technique in [11] to handle tree-pattern queries using bounded main memory. Moreover, we proposed a new algorithm that can utilize existing index structures to further improve the method's performance. To the best of our knowledge, this is the first time that a method integrating FSMs and value based indexes has been reported.

Many structural summaries have already been proposed, such as the bisimilarity-driven family of indexes [18], dataguides [14], and suffix tree-like structures [10, 27]. They all group the document nodes according to a structural property. Those groups create a new structure, smaller than the original document, which permits the processing of structural queries directly on it. For the class and semantics of queries that we target, those summaries alone are not

enough, and a post-processing matching step is necessary. Assuming that the index probing cost is a parameter of the structural summary, we regarded those structures as a partitioning of the document nodes and examined the cost of the post processing matching process, as this is the most expensive phase of the method.

Cost-based query optimization techniques for XML [22, 29] are also related to our work. They investigate the applicability of common query optimization techniques to answer tree-pattern queries. They are complementary to our study as they target an environment where a cost-based optimization module is available.

6 Conclusion

We proposed a classification of tree-pattern query processing algorithms considering important features such as data access and matching process. We also identified the common behavior of the algorithms within the categories. Furthermore, we adapted previous and successful XML query processing techniques for handling tree-pattern queries as well. Specifically, we adjusted a DFA-based approach, and we improved its performance by accessing nodes from a B^+-tree instead of purely sequential scan. Such an improvement provided better results in comparison to the plain DFA. We further showed that query-driven methods can be considered as static plans for index nested loops join . Hence, we introduced a generalization that examines all left-deep plans. Finally, we introduced an approach that combines a structural summary with a set-based matching algorithm. We then performed the first thorough and extensive analysis of tree-pattern query processing techniques using real, benchmark and custom data. In summary, if a structural summary exists covering the query, it should be the method of choice (assuming this index is considerably smaller than the real dataset). The DFA-based solutions (even with the use of indexing) typically had the worst performance for the stored data environment we used. The query-driven methods have performance that can vary drastically with the data and query. Since existing methods use static plans, we experimentally showed that they cannot provide guarantees for lightweight (non-optimized) XML engines. Hence, the holistic approach (i.e. the indexed TWIGSTACK) was the most robust and predictable method, and it should be definitely implemented on lightweight processors.

References

[1] S. Abiteboul et. al. The Lorel Query Language for Semistructured Data. *Intl. Journal on Digital Libraries*, 1(1), 1997.

[2] S. Al-Khalifa et.al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. of ICDE*, 2002.

[3] A. Berglund et. al. XML Path Language (XPath) 2.0. *W3C Recommendation. http://www.w3.org/TR/xpath20*, Nov 2003.

[4] S. Boag et. al. XQuery 1.0: An XML query language. In *W3C Working Draft. http://www.w3.org/TR/xquery*, Nov 2003.

[5] N. Bruno et.al. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. of SIGMOD*, 2002.

[6] T. Chen, J. Lu, and T. W. Ling. On Boosting Holism in XML Twig Pattern Matching using Structural Indexing Techniques. In *Proc. of SIGMOD*, 2005.

[7] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proc. of VLDB*, 2002.

[8] C.-W. Chung, J.-K. Min, and K. Shim. APEX: An Adaptive Path Index for XML Data. In *Proc. of SIGMOD*, 2002.

[9] M. P. Consens and T. Milo. Optimizing Queries on Files. In *Proc. of SIGMOD*, 1994.

[10] B. F. Cooper et. al. A Fast Index for Semistructured Data. In *Proc. of VLDB*, 2001.

[11] Y. Diao et. al. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM TODS*, 28(4), Dec 2003.

[12] eXist native XML database. In *http://exist.sourceforge.net*.

[13] Galax XQuery processor. In *http://www.galaxquery.org/*.

[14] R. Goldman and J. Widom. DataGuides: Enabling Formulation and Optimization in Semistructured Databases. In *Proc. of VLDB*, 1997.

[15] A. Halverson et. al. Mixed Mode XML Query Processing. In *Proc. of VLDB*, 2003.

[16] T. J. Green et.al. Processing XML Streams with Deterministic Automata. In *Proc. of ICDT*, 2003.

[17] H. Jiang et.al. Holistic Twig Joins on Indexed XML Documents. In *Proc. of VLDB*, 2003.

[18] R. Kaushik et.al. Covering Indexes for Branching Path Queries. In *Proc. of SIGMOD*, 2002.

[19] H. Li et.al. An Evaluation of XML Indexes for Structural Joins. *Sigmod Record*, 33(3), Sept. 2004.

[20] H. Liefke. Horizontal Query Optimization on Ordered Semistructured Data. In *Proc. of WebDB*, 1999.

[21] J. Lu, T. Chen, and T. W. Ling. Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach. In *Proc. of CIKM*, 2004.

[22] J. McHugh and J. Widom. Query Optimization for XML. In *Proc. of VLDB*, 1999.

[23] T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proc. of ICDT*, 1999.

[24] F. Peng and S. S. Chawathe. XPath Queries on Streaming Data. In *Proc. of SIGMOD*, 2003.

[25] P. R. Rao and B. Moon. PRIX: Indexing and Querying XML Using Prufer Sequences. In *Proc. of ICDE*, 2004.

[26] SAX. Simple API for XML. In *http://sax.sourceforge.net*.

[27] H. Wang and X. Meng. On the Sequencing of Tree Structures for XML Indexing. In *Proc. of ICDE*, 2005.

[28] H. Wang et.al. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proc. of SIGMOD*, 2003.

[29] Y. Wu et.al. Structural Join Order Selection for XML Query Optimization. In *Proc. of ICDE*, 2003.

[30] XMark. The XML benchmark project. In *http://www.xml-benchmark.org*.

[31] XML Data Repository. In *http://www.cs.washington.edu/research/xmldatasets/*.

[32] N. Zhang, V. Kacholia, and M. T. Ozsu. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *Proc. of ICDE*, 2004.