Navigation- vs. Index-Based XML Multi-Query Processing

Nicolas Bruno, Luis Gravano
Columbia University
{nicolas,gravano}@cs.columbia.edu

Nick Koudas, Divesh Srivastava
AT&T Labs–Research
{koudas,divesh}@research.att.com

Abstract

XML path queries form the basis of complex filtering of XML data. Most current XML path query processing techniques can be divided in two groups. Navigation-based *algorithms compute results by analyzing an input document one tag at a time. In contrast,* index-based *algorithms take advantage of precomputed numbering schemes over the input XML document. In this paper we introduce a new index-based technique, Index-Filter, to answer multiple XML path queries. Index-Filter uses indexes built over the document tags to avoid processing large portions of the input document that are guaranteed not to be part of any match. We analyze Index-Filter and compare it against Y-Filter, a state-of-the-art navigation-based technique. We show that both techniques have their advantages, and we discuss the scenarios under which each technique is superior to the other one. In particular, we show that while most XML path query processing techniques work off SAX events, in some cases it pays off to preprocess the input document, augmenting it with auxiliary information that can be used to evaluate the queries faster. We present experimental results over real and synthetic XML documents that validate our claims.*

1. Introduction

XML employs a tree-structured model for representing data. Queries in XML query languages, such as XQuery [4], typically specify patterns of selection predicates on multiple elements that have some specified tree structured (e.g., parent-child, ancestor-descendant) relationships, as the basis for matching XML documents. For example, the path query `//book[.//title = 'XML']` matches `book` elements that have a descendant `title` element that in turn has a child `XML` value. Finding all occurrences of such path queries in an XML document is a core operation in various XML query processing scenarios that are considered in the literature.

The traditional XML query processing scenario involves asking a single query against a (possibly preprocessed and indexed) XML document [1]. The goal here is to identify the matches to the input query in the XML document. Approaches to solving this problem, e.g., [2, 5, 18, 25], typically take advantage of indexes on XML elements and values, and use specialized join algorithms for composing results of index lookups and computing answers to the path queries.

XML query processing also arises in the scenario of information dissemination, where many (standing) XML path queries have been preprocessed, and a stream of XML documents is presented as input. The goal here is to identify the path queries and their matches in the input XML documents, and disseminate this information to the users who posed the path queries. Approaches to solving this problem, e.g., [3, 6, 9, 14, 15], typically navigate through the input XML document one tag at a time and use the preprocessed structure of path queries to identify the relevant queries and their matches.

These two scenarios considered in the literature are both instances of a general scenario where *multiple XML path queries need to be matched against an XML document*, and either (or neither) of the queries and the document may have been preprocessed. In principle, each of the query processing strategies (index-based and navigation-based) could be applied in our general scenario. How this is best achieved, and identifying the characteristics of our general scenario where one strategy dominates the other, are the subjects of this paper. The contributions of the paper are as follows:

- A straightforward way of applying the *index-based* approaches in the literature to our general scenario would answer each path query separately, which may not be the most efficient approach, as research on multi-query processing in relational databases has demonstrated. The first contribution of our paper is a novel index-based technique, *Index-Filter*, to answer multiple XML path queries against an XML document. *Index-Filter* generalizes the *PathStack* algorithm of [5], and takes advantage of a prefix tree representation of the set of XML path queries to share computation during multiple query evaluation. We experimentally show the superiority of *Index-Filter* over the independent use of *PathStack* for multiple queries.

- *Navigation-based* approaches in the literature could be applied to the general scenario as well. The second

[1] An XML database can be viewed as an XML document, once a dummy root node has been added to convert the forest into a tree.

contribution of our paper is to experimentally compare *Index-Filter* against *Y-Filter* [9, 10], a state-of-the-art navigation based technique, which we suitably extended to identify multiple query matches in an XML document. Both techniques have their advantages, and we use a variety of real and synthetic XML documents in our experiments to establish the following results:

- When the number of queries is small or the XML document is large, *Index-Filter* is more efficient than *Y-Filter* if the required indexes are already materialized, due to the focused processing achieved by the use of indexes.
- When the number of queries is large and the document is small, *Y-Filter* is more efficient due to the scalability properties of the *Y-Filter*'s hash tables.
- When we also consider the time spent for building indexes on the XML document on the fly, the trends remain generally the same, but the gap between the algorithms is reduced.

The rest of the paper is structured as follows. In Section 2 we discuss our data and query models. Section 3 is the core of the paper and discusses two query processing techniques. In Section 3.2 we review *Y-Filter*, a state-of-the-art navigation-based algorithm, suitably enhancing it for our application scenario. Then, in Section 3.3 we introduce our novel index-based algorithm, *Index-Filter*. In Section 5 we report experimental results using the setting of Section 4. Finally, we review related work in Section 6.

2. Data and Query Models

We now introduce the XML data and query models that we use in this paper, and define our problem statement.

2.1. XML Documents

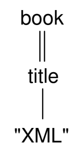
An *XML document* can be seen as a rooted, ordered, labelled tree, where each node corresponds to an element or a value, and the edges represent (direct) element-subelement or element-value relationships. The ordering of sibling nodes (children of the same parent node) implicitly defines a total order on the nodes in a tree, obtained by traversing the tree nodes in preorder. An XML database can be viewed as an XML document, once a dummy root node has been added to convert the forest into a tree.

Example 1 *Figure 1 shows a fragment of an XML document that specifies information about a book. The figure shows four children of the* `book` *root node, namely:* `title`, `allauthors`, `year`, *and* `chapter`, *in that order. Intuitively, we can interpret the XML fragment in the figure as a book published in the year* 2000 *by* `Jane Poe`, `John Doe`, *and* `Jane Doe`, *entitled* `XML`. *Its first chapter, also entitled* `XML`, *starts with the section* `Origins`.

The meaning and utility of the numbers associated with the tree nodes will be explained later in Section 3.3.1. Until then, we can simply think of those numbers as unique node identifiers in the XML tree.

2.2. Query Language

XQuery [4] path queries can be viewed as sequences of location steps, where each node in the sequence is an element tag or a string value, and query nodes are related by either *parent-child* steps (depicted using a single line) or *ancestor-descendant* steps (depicted using a double line) [2].



(a) Path query.

book	title	XML
(1:150,1)	(2:4,2)	(3:3,3)
(1:150,1)	(35:37,3)	(36:36,4)

(b) Query answer.

Figure 2. Query model used in this paper.

Given a query q and an XML document D, a *match of* q *in* D is a mapping from nodes in q to nodes in D such that: (i) node tags and values are preserved under the mapping, and (ii) structural (parent-child and ancestor-descendant) relationships between query nodes are satisfied by the mapped document nodes. The *answer* to a query q with n nodes is an n-ary relation where each tuple $(d_1, \ldots, d_n)$ consists of the document nodes identifying a match of q in D [3].

Example 2 *Consider again Figure 1. The path query* `//book[.//title = "XML"]` *identifies* `book` *elements that have a descendant* `title` *element that in turn has a child* `XML` *node. This query can be represented as in Figure 2(a). A match for this query in the XML fragment of Figure 1 is the following mapping, where nodes in the document are represented by their associated numbers:* `book` $\rightarrow (1:150,1)$, `title` $\rightarrow (2:4,2)$, *and* `XML` $\rightarrow (3:3,3)$. *It is easy to check that both the name tags and the structural relationships between nodes are preserved by the mapping above. Figure 2(b) shows the answer to the query in Figure 2(a) over the XML fragment of Figure 1.*

2.3. Problem Statement

Finding all matches of a path query in an XML document is a core operation in various XML query processing scenarios that have been considered in the literature. In this paper, we consider the general scenario of matching multiple XML

[2] XQuery path queries permit other axes, such as *following* and *preceding*, which we do not consider in this paper.

[3] Actually, an XQuery path query is a 1-ary projection of this n-ary relation. Compositions of path queries need to be used (as in the XQuery FOR clause) to obtain an n-ary relation. To allow for this generality keeping the exposition simple, we identify the path query answer with the n-ary relation.

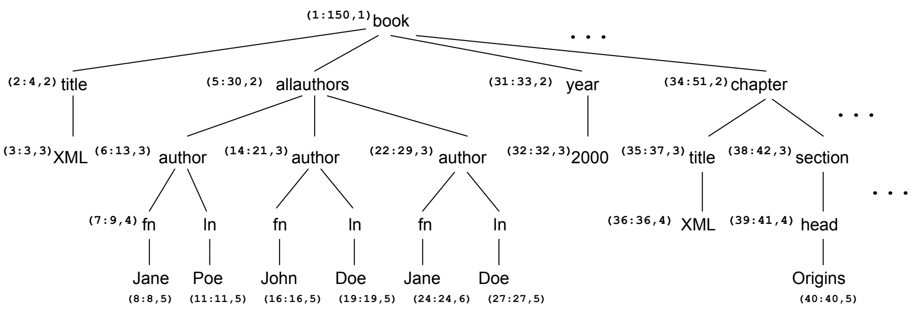


Figure 1. A fragment of an XML document.

path queries against an XML document, and focus on the following problem:

> **XML Multiple Query Processing**: Given an XML document D and a set of path queries $Q = \{q_1, \ldots, q_n\}$, return the set $R = \{R_1, \ldots, R_n\}$, where R_i is the answer (all matches) to q_i on D.

We now describe some common scenarios that are covered by the problem formulation above. In the traditional XML query processing scenario, Q includes a single query q_1, and the document D is large and indexed. When the traditional XML query processing scenario is augmented to deal with multi-query processing, Q includes multiple queries, and the document D is large and indexed. Finally, in the XML information dissemination scenario, Q includes many queries, and the document D is small and not indexed. In the next section we study algorithms for our problem of processing multiple XML path queries efficiently.

3. XML Multiple Query Processing

We now address the central topic of this paper: processing strategies for multiple path queries over an XML document. First, in Section 3.1 we describe a mechanism to compress the representation of multiple input path queries that is used by the different algorithms in this section. Then, in Section 3.2 we review *Y-Filter* [9, 10], a state-of-the-art algorithm that consumes the input document one tag at a time and incrementally identifies all input path queries that can lead to a match [4]. Finally, in Section 3.3 we introduce *Index-Filter*, a new algorithm that exploits indexes built over the document tags and avoids processing tags that will not participate in a match. Section 3.3.3 addresses the problem on how to efficiently materialize the indexes needed by *Index-Filter*.

To draw an analogy with relational query processing, *Y-Filter* can be regarded as performing a sequential scan of the input "relation," while *Index-Filter* accesses the relation via indexes. Extending the above analogy, not surprisingly, neither algorithm is best under all possible scenarios. In Section 5, we experimentally identify the scenarios for which each algorithm is best suited.

[4]Another navigation-based technique, *X-Trie* [6], was presented concurrently with *Y-Filter*. We do not analyze *X-Trie* in this paper, since *X-Trie* is specifically designed to identify at most one match for a given query, and it is not clear how to modify this technique to return *all matches*.

3.1. Prefix Sharing

When several queries are processed simultaneously, it is likely that significant commonalities between queries exist. To eliminate redundant processing while answering multiple queries, both the navigation- and index-based techniques identify query commonalities and combine multiple queries into a single structure, which we call *prefix tree*. Prefix trees can significantly reduce both the space needed to represent the input queries and the bookkeeping required to answer them, thus reducing the execution times of the different algorithms. Consider the four path queries in Figure 3(a). We can obtain a prefix tree that represents such queries by sharing their common prefixes, as shown in Figure 3(b). It should be noted that although other sharing strategies can be applied, we do not explore them in this work.

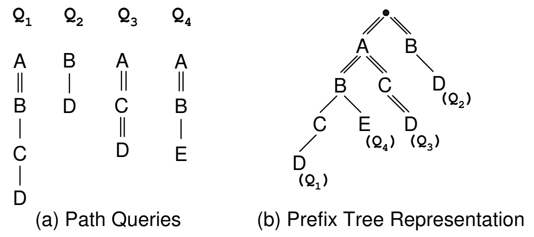


Figure 3. Using prefix sharing to represent path queries.

3.2. Y-Filter: A Navigation-Based Approach

Y-Filter is a state-of-the-art navigation-based technique for processing multiple path queries. The main idea is to augment the prefix tree representation [5] of the input queries into a *non-deterministic finite automaton* (NFA) that behaves as follows: (i) The NFA identifies the exact "language" defined by the union of all input path queries; (ii) when an output state is reached, the NFA outputs all matches for the queries

[5]Actually, *Y-Filter* uses a slightly different representation of the prefix tree, but we omit details to simplify the presentation.

accepted at such state. Unlike an NFA used to identify a regular language, the filtering of XML documents requires that processing continues until all possible accepting states have been reached. The incoming XML document is parsed one tag at a time. While parsing, *start-tag* tokens trigger transitions in the NFA (the automaton is non-deterministic, so many states can be active simultaneously). When an *end-tag* token is parsed, the execution backtracks to the state immediately preceding the corresponding start-tag. To achieve this goal, a run-time stack structure is used to maintain the active and previously processed states.

Example 3 *Consider the NFA shown in Figure 4(a), which corresponds to the prefix tree of Figure 3(b). Note that each node in the prefix tree is converted to a state in the NFA, and the structural relationships in the prefix tree are converted to transitions in the NFA, triggered by the corresponding tags. As each start-tag from the document fragment in Figure 4(b) is parsed, the NFA and the run-time stack are updated. Figure 4(c) shows the run-time stack after each step, where each node in the stack contains the set of active states in the NFA. Initially, only the starting state, 0, is active. When reading the start-tag for node A_1, state 0 fires the transition to state 1, and both states 0 and 1 become active (state 0 remains active due to the descendant edge from state 0 to state 1; otherwise we would not be able to capture the match that uses A_2, which is a descendant of A_1). As another example, after reading the start-tag D_1, both states 5 and 8 become active, and therefore a match for queries Q_2 and Q_3 is detected (note that after reading D_1, node 2 is not active anymore, since the firing transition from node 2 to node 5 used a child –not descendant– structural relationship)* [6]*. As a final example, after reading the close-tag for D_1, the run-time stack is backtracked to the state immediately before reading the start-tag for D_1.*

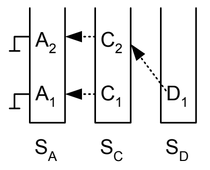
Implementation-wise, *Y-Filter* augments each query node in the NFA with a hash table. The hash table is indexed by the children node tags, and is used to identify a particular transition of a given state. Therefore, the NFA can be seen as a tree of hash tables. This implementation is a variant of a hash table-based NFA, which has been shown to provide near constant time complexity to perform state transitions (see [10] for more details).

Compact Solution Representation. The original formulation of *Y-Filter* [9, 10] just returns the set of queries with a non-empty answer in a given document. However, we are interested in returning all matches as the answer for each query (see Section 2). Consider again Figure 4 when the algorithm

[6]The actual implementation of *Y-Filter*'s NFA is slightly more complex than described above, to address a special situation. In particular, when a given state has two children with the same tag but different structural relationships (child and descendant), a new intermediate state is added to the NFA to differentiate between the two transitions.

A	C	D
A_1	C_1	D_1
A_1	C_2	D_1
A_2	C_2	D_1

(a) Answer for Q_3



(b) Stack encoding for Q_3

Figure 5. Compact solution representation.

processes D_1. The partial matches for query Q_3 are shown in Figure 5(a). When parsing node D_1, the original *Y-Filter* algorithm would get to the final state for Q_3, only guaranteeing that there is *at least one match* for Q_3 in the document. In other words, there is no way to *keep track* of repeating tags that might result in multiple solutions. To overcome this limitation, in this paper we augment *each node q* in the prefix tree (NFA) with a stack S_q. These stacks S_q efficiently keep track of all matches in the input document from the root to a given document node. Each element in the stack is a pair: $\langle$ node from the XML document, pointer to a position in the parent stack $\rangle$. At *every* point during the computation of the algorithm, the following properties hold:

1. The nodes in S_q (from bottom to top) are guaranteed to lie on a root-to-leaf path in the XML document.
2. The set of stacks contains a *compact encoding* of partial and total matches to the path query, which can represent in linear space a potentially exponential (in the number of query nodes) number of answers to the path query.

Given a chain of stacks in a leaf-to-root path in the prefix tree, corresponding to some input query, the following recursive property allows us to extract the set of matches that are encoded in the stacks: given a pair $\langle t_q, p_q\rangle$ in stack S_q, the set of partial matches using tag t_q that can be extracted from the stacks is found by extending t_q with either the partial matches that use the tag pointed by p_q in $S_{\texttt{parent}(q)}$ or any tag that is below p_q in stack $S_{\texttt{parent}(q)}$. The following example clarifies this property.

Example 4 *Consider again Figure 4. The set of all matches for the NFA in Figure 4(a) is shown in Figure 5(a). Figure 5(b) shows the chain of stacks for query Q_3 and the stack encoding for the document fragment at the point that D_1 is processed. The match $[A_2, C_2, D_1]$ is encoded since D_1 points to C_2, and C_2 points to A_2. Since A_1 is below A_2 on S_A, $[A_1, C_2, D_1]$ is also encoded. Finally, since C_1 is below C_2 on S_C and C_1 points to A_1, $[A_1, C_1, D_1]$ is also encoded. Note that $[A_2, C_1, D_1]$ is not encoded, since A_2 is* above *the node (A_1) on S_A to which C_1 points.*

It can be shown that in order to maintain the stacks in the NFA we need to proceed as follows: (i) every time an open tag t_o is read and consequently some state n becomes active due to a transition from state n_p, we push into n's stack t_o along with a pointer to the top of n_p's stack; and (ii) every time a close tag t_c is read and the top of the (global) run-time

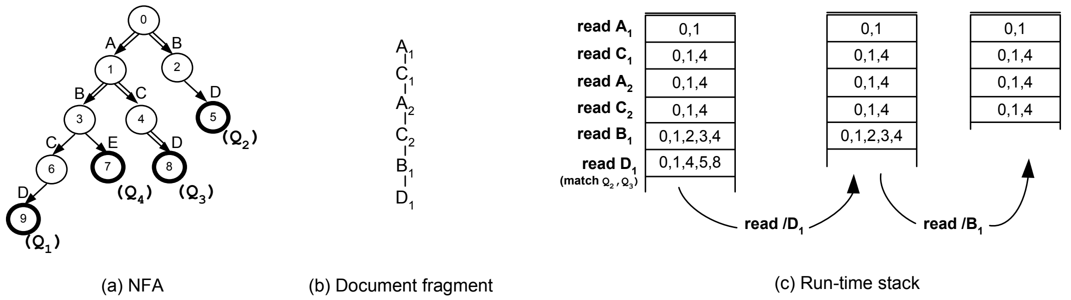


Figure 4. *Y-Filter* algorithm.

stack structure contains states $\{n_1, \ldots, n_k\}$, we pop the top element from the stacks, associated with states n_i, that were modified when the corresponding open-tag t_o was read. It is important to note that the stacks are shared among queries in the prefix-tree. In fact, we only need one stack per state in the NFA to represent all partial matches of the queries that share such state. We refer to [5] for more details about maintaining compact solutions using stacks.

In conclusion, our modification to the original *Y-Filter* algorithm allows us to return not only the queries that have at least one match in the document, but all its matches. Moreover, this can be done by using a limited amount of memory (proportional to the height of the input XML document and the number of active states). We present now an alternative technique to return the set of all matches for the given input queries that is based on a different design principle: the use of index information over the input XML document.

3.3. Index-Filter: An Index-Based Approach

In this section we present *Index-Filter*, a novel technique to answer multiple path queries by exploiting indexes that provide structural information about the tags in the XML document. By taking advantage of this additional information, *Index-Filter* is able to avoid processing certain tags in the document that are guaranteed not to be part of any match. We first discuss the index structure that we use in *Index-Filter*, and then we present the main algorithm.

3.3.1 Indexing XML Documents

We now describe how to extend the classic inverted index data structure used in information retrieval [21] to provide a positional representation of elements and values in the XML document. This representation was introduced in [7, 8] and has been used in [2, 5, 25] for matching XML path queries. As we will see, this representation allows us to efficiently check whether two tags in the XML documents are related by a parent/child or ancestor/descendant structural relationship. The position of an element occurrence in the XML document is represented as the pair `(L:R,D)` where `L` and `R` are generated by counting word numbers from the beginning of the document to the start and the end of the element being indexed, respectively, and `D` is the nesting depth of the element in the document (see Figure 1 for examples of pairs associated with some tree nodes based on this representation).

We can easily determine structural relationships between tree nodes using this indexing scheme. Consider document nodes n_1 and n_2, encoded as $(\mathtt{L}_1\!:\mathtt{R}_1,\mathtt{D}_1)$ and $(\mathtt{L}_2\!:\mathtt{R}_2,\mathtt{D}_2)$, respectively. Then, n_1 is an *ancestor* of n_2 (and n_2 is a *descendant* of n_1) if and only if $\mathtt{L}_1 < \mathtt{L}_2$ and $\mathtt{R}_2 < \mathtt{R}_1$. To check whether n_1 is the *parent* of n_2 (n_2 is a *child* of n_1) we also need to verify whether $\mathtt{D}_1\mathtt{+1=}\mathtt{D}_2$.

Example 5 *Consider the XML fragment in Figure 1. The* `author` *node with position* $(6\!:\!13,\ 3)$ *is a descendant of the* `book` *node with position* $(1\!:\!150,\ 1)$*, since* $L_{\mathtt{book}} = 1 < 6 = L_{\mathtt{author}}$*, and* $R_{\mathtt{author}} = 13 < 150 = R_{\mathtt{book}}$*. Also, the* `author` *node just mentioned is the parent of the* `fn` *node with position* $(7\!:\!9,\ 4)$*, since* $L_{\mathtt{author}} = 6 < 7 = L_{\mathtt{fn}}$*,* $R_{\mathtt{fn}} = 9 < 13 = R_{\mathtt{author}}$*, and* $D_{\mathtt{fn}} = 4 = 3 + 1 = D_{\mathtt{author}} + 1$*.*

An important property of this positional representation is that checking an ancestor-descendant relationship is computationally as simple as checking a parent-child relationship, i.e., we can check for an ancestor-descendant structural relationship without knowledge of intermediate nodes on the path. We now introduce the *Index-Filter* algorithm. Later, in Section 3.3.3 we address the issue of how to efficiently materialize a set of indexes for a given XML document.

3.3.2 Algorithm Index-Filter

Based on the representation of positions in the XML document described above, we now present the *Index-Filter* algorithm. Analogously to the case of *Y-Filter*, we augment the input prefix tree structure for *Index-Filter*. Specifically, before executing *Index-Filter*, we associate with each node q in the input prefix tree the following information: (i) an *index stream* T_q, which contains the indexed positions of document nodes that match q sorted by their `L` values, (ii) an empty *stack* S_q as discussed in Section 3.2, and (iii) a priority queue P_q that allows dynamic and efficient access to the child of q having the smallest `L` value in its stream. To ensure correctness, initially we add the index entry $(-\infty : +\infty, 0)$ to the stack S_{root}. In the rest of the section, the concepts of a prefix tree and its root node are used interchangeably. We denote the current element in stream T_q as the *head of* T_q,

```
Algorithm Index-Filter(q)
01 while (true) // find candidate node
02   repeat
03     $min$ = getMin($P_q$)
04     if ($\neg min \vee ($isAccept($q$) $\wedge$ nextL($T_{min}$) $>$ nextR($T_q$)))
05       $q$.knowSolution=true; return
06     while (nextR($T_q$) $<$ nextL($T_{min}$))
07       advance($T_q$) // advance $q$'s stream
08     if (eof($T_{min}$)) $q$.knowSolution=false; return
09     while ($\neg$empty($S_q$) $\wedge$ topR($S_q$) $<$ nextL($T_{min}$))
10       pop($S_q$) // clean $q$'s stack
11     while (nextL($T_{min}$) $<$ skipToL($q$))
12       advance($T_{min}$)
13       $min$.knowSolution=false
14     knewSolution= $min$.knowSolution
15     if ($\neg min$.knowSolution) Index-Filter($min$)
16   until (knewSolution)
17   // process candidate node
18   if (nextL($T_{min}$) > nextL($T_q$))
19     $q$.knowSolution= true
20     return
21   else
22     push($S_{min}$, (next($T_{min}$), $ptr$_top($S_q$)))
23     if (isAccept($min$))
24       outputSolutions($min$)
25     advance($T_{min}$); if (isLeaf($min$)) pop($S_{min}$)
26     Index-Filter($min$)
27 end while

Function skipToL($q$)
01 if (empty($S_q$)) return nextL($T_q$)
02 else return topL($S_q$)
```

Figure 6. Algorithm *Index-Filter*.

and we access the head's L and R components by the functions nextL and nextR, respectively (if we consume T_q entirely, nextL(T_q)=nextR(T_q)=$+\infty$). Similarly, we access the L and R components of the top of S_q by the functions topL and topR, respectively. We now describe *Index-Filter*, which is shown in pseudocode in Figure 6.

We execute *Index-Filter*(q) to get all the answers for the prefix tree rooted at q. The algorithm's invariant ensures that after executing *Index-Filter*(q), we are guaranteed that either (1) T_q's head participates in a new match when all structural relationships are regarded as ancestor/descendant (outputSolutions in line 24 will later enforce the appropriate relationships); or otherwise (2) the stream T_q is consumed entirely. Additionally, we can guarantee that for all descendants q' of q in the prefix tree, every index entry in $T_{q'}$ with L component smaller than nextL(T_q) was already processed. To avoid redundant computations, we memorize this property by carefully manipulating the boolean variable q.knowSolution: if q.knowSolution=**true**, we know that T_q's head participates in at least one new match; otherwise all we can say is that T_q's head *might* participate in a new match, but we do not know for sure (initially, q.knowSolution is set to **false** for every node q). The algorithm iterates through two phases until all matches are returned. In the first phase (lines 2-16), we identify min, the child of q with the minimal L value in its stream's head that participates in some match. In the second phase (lines 17-26), we process min depending on the actual relationship with T_q's head. We now give some details on each phase.

To identify min, we first use the priority queue P_q to select the child of q with the smallest stream head (line 3). Lines 4-5 cover the special case that node q is a leaf node in the prefix tree (so q has no children and there is no min child), or q is an internal node in the prefix tree but some query has q as its accept state and q's position ends before the position of any of q's children. In such cases, we simply update q.knowSolution $=$ **true** and return. Otherwise, in the general case, if T_{min}'s head starts after T_q's head ends, we can guarantee that no new match can exist for T_q's head, so we advance T_q (see Figure 7(a)). At this point, if T_{min} is consumed entirely, we know that there are no new solutions for q so we return (line 8). Otherwise, we clean from q's stack all elements that cannot participate in any new match, i.e., those elements in S_q whose R component is smaller than the L component of T_{min}'s head (see Figure 7(b)). After that, we compute the value skipToL, which is the smallest L value for a node from q for which a new match can exist. If T_{min}'s head starts before skipToL, we know that T_{min}'s head cannot participate in any new match, so we advance T_{min} (see Figure 7(c)). In such a case, we need to *reset* the value min.knowSolution (line 13), since we can no longer guarantee that T_{min}'s head participates in a new match after advancing T_{min} in line 12. At this point, in line 15, if we cannot guarantee that T_{min}'s head participates in a match (i.e., min.knowSolution $=$ **false**), we recursively call *Index-Filter*(min). After we return from this recursive call, we can guarantee (from the algorithm's invariant) than T_{min}'s head participates in a new match. However, T_{min} could have been advanced in the recursive call, so we cannot guarantee that min is the children of q with the *minimal* value of L participating in a match. Therefore, we only continue with the second phase if we could guarantee that min participated in a match *before* the recursive call (see lines 14 and 16). Otherwise, we simply repeat the procedure of finding the minimal child of q with a match (of course, in the next iteration node min could be the minimal one, although it is not always the case).

When we enter the second phase, we can guarantee that T_{min}'s head participates in some match and its position relative to q can be just one of the two cases of Figure 7(d). In the case that T_{min}'s head starts after T_q's head (case 2 in Figure 7(d)), we know that T_q's head participates in a match

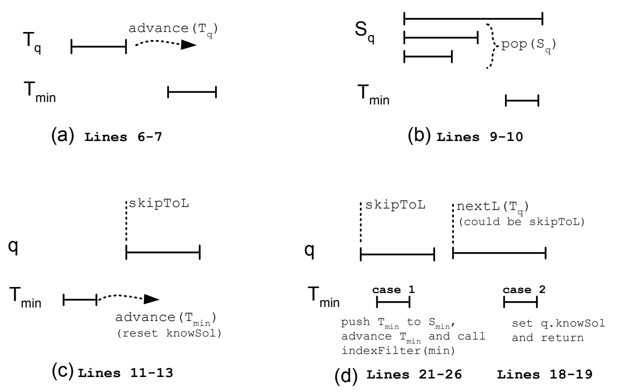


Figure 7. Possible scenarios in the execution of *Index-Filter*.

as well, so we set q.knowSolution and return (lines 19-20). Otherwise (case 1 in Figure 7(d)), T_{min}'s head ends before T_q's head starts but participates in a match with nodes in S_q. Therefore, we need to process node min before returning with any match for T_q's head. We first push T_{min}'s head to S_{min}, and if some query has min as its accept node, we expand the new matches from the chain of stacks. Finally, in preparation for the next iteration, we advance T_{min} and recursively call *Index-Filter*(min) to process any remaining entries from the subtree rooted at min.

3.3.3 Building Indexes

When analyzing the *Index-Filter* algorithm in the previous section, we assumed that the required indexes were already precomputed and available to *Index-Filter*. We now give more details on how to efficiently materialize a set of indexes given an input XML document. Conceptually, we can assign the positional representation of the nodes in the XML document by traversing the XML tree in preorder as explained next. We maintain a global counter and increment it every time we move to a new node (either by moving to a new child or when returning to the parent node after traversing all of its subtrees). Whenever we reach a node for the first time, we assign the current value of the counter to the `L` component of the positional representation, and when we leave a node after traversing all its subtrees we assign the current value of the counter to the `R` component of the positional representation (the `D` component of the positional representation is easily derived from the number of ancestors of each node). We now present a concrete implementation of this procedure that uses little main memory and scales gracefully with the size of the input XML document. The general procedure to obtain the indexes for the nodes of the XML document consists of two steps that can be summarized as follows (see Figure 8):

1. Use a SAX-based parser on the input XML tree. The i-th tag found (irrespective of whether it is a start- or an end-tag) receives integer i as its identifier [7]. Every time we parse a start-tag, we push into a global stack the value of the tag along with its identifier, which becomes the `L` value for the node, and the level value, which is simply the current number of elements in the stack. (Figure 8 shows a snapshot of the execution right after parsing the start-tag for node C_1.) On the other hand, every time we parse an end-tag, we know (assuming the XML document is consistent) that the top element in the stack has the information of the corresponding start-tag, so we pop the top of the stack, assign the current identifier to the `R` component, and output the index entry to a temporary file.

2. The order of the tags in the intermediate file produced in the previous step would match that of a post-order traversal of the XML tree (in particular, the different tags are not even grouped together). As seen in the description of *Index-Filter* (Section 3.3) a crucial property of the index entries for a given tag is that they are sorted by `L` value. For that reason, in the second step we sort the intermediate file by $\langle tag, L\rangle$. In that way, all tags are grouped together and sorted by their `L` value, as desired (see Figure 8).

To provide efficient access to the indexes of individual tags, we build a B-tree over the tags. Throughout the index-building process, and with the exception of the sorting phase, memory requirements are proportional to the height of the XML tree to maintain the stack. Interestingly, the memory requirements are independent of the size of the XML tree.

Main Memory Optimization. It turns out that if the whole document and the indexes fit in main memory, we can build the indexes without the sorting step. The intuition is to use growing arrays in memory to hold the index for each tag separately. Every time we parse a start-tag, we append a new index entry to the corresponding tag array with the `L` and `D` entries as before (the `R` entry remains unknown). We still use a global stack, but this time we just store in it pointers to index entries in the arrays, which still contain unknown `R` values. When we parse an end-tag, we pop the top pointer from the stack and update the corresponding index entry in the array with the `R` value as explained before. This way, each index is created independently and in the right order, so there is no need to sort any intermediate result.

4. Experimental Setting

In Section 3 we studied two algorithms to answer multiple path queries over XML documents simultaneously: *Y-Filter* and *Index-Filter*. Although both techniques can be used for

[7]To keep the presentation simple, we treat values, such as 'Jane' or 'XML', as if they were composed of adjacent pairs of open- and close-tags, e.g., <Jane></Jane>, but we assign the same integer to both the open- and close-tags.

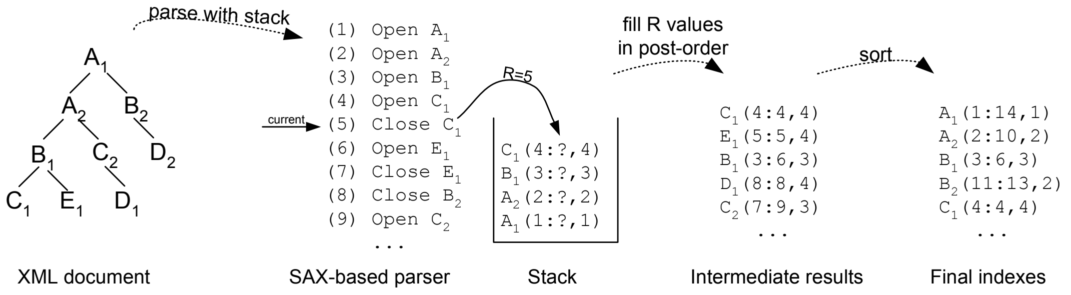


Figure 8. Materializing the positional representation of XML nodes.

the same purpose, there are important conceptual differences between them. On one hand, *Y-Filter* computes results by analyzing input documents one tag at a time, using very efficient data structures to process each tag. On the other hand, *Index-Filter* takes advantage of indexes built over the input document to avoid processing large portions of the input that are guaranteed not to be part of any match. It is therefore interesting to study under which conditions each technique is more efficient than the other. Below we describe the data sets and query workloads that we use in our experiments of Section 5, as well as the metric we use to compare the algorithms of Section 3.

4.1. XML Documents

XML is a general and flexible encoding that can be used in a large variety of scenarios. In fact, both strictly regular data sets (such as relational tables) and highly unstructured information can be modeled by XML documents. We tried to cover this wide spectrum of possibilities by using both semi-structured and highly structured XML documents in our experiments, with both deep and shallow nesting. Specifically, we use three families of XML documents, which we call *DBLP*, *TPC-H*, and *Random*, and are explained below.

DBLP: This family of documents consists of "unfolded" fragments of the *DBLP* database [16], which represents information about authors and their papers. We generated the *DBLP* document by starting with an arbitrary author and converting its information to XML format. Then, for each paper, we recursively replaced each coauthor name with its actual XML information. We continued "unfolding" authors until we reached a previously traversed author, a depth of 100 authors, or a predefined document size. The resulting XML document has depth 805 and around 3 million nodes, representing 93,536 different papers from 36,900 unique authors. To obtain a family of *DBLP* documents of different sizes, we post-filtered the original XML document and created *DBLP* documents with sizes ranging from 1Kb to 1Mb (roughly corresponding to 100 to 100,000 tags).

TPC-H: This family of XML documents was generated from the popular *TPC-H* benchmark for relational databases [23]. We first created *TPC-H* databases of sizes 1M, 5M, and 10MB, and then used the strategy described in [22] to translate relational databases to XML documents. In particular, we used the chain of foreign-key joined tables `lineitem, orders, customer, nation,` and `region`. The resulting XML documents ranged from 225,000 to around 2,000,000 tags.

Random: This family of XML documents was produced with a similar data generation tool as the one presented in [1], and consists of random, binary, and balanced XML trees with depths ranging from 10 to 18 (corresponding to 1K to 256K tags in the resulting XML documents). We allowed repeating tags in any leaf-to-root path, and the tags were generated from a uniform distribution. We used 25 distinct tags for internal nodes in the XML document and 250 distinct tags for leaf nodes.

4.2. Query Workloads

In each experiment, the set of queries consists of 1 to $25,000$ path queries, with a random number of nodes between 2 and 10. We varied these and other parameters and obtained comparable results, so we do not report those experiments in this work.

4.3. Metrics

To evaluate the relative merits of *Y-Filter* and *Index-Filter*, we implemented both algorithms in C++, sharing as much code and data structures as possible for a fair comparison. In most of the experiments, we compare the execution time of both *Y-Filter* and *Index-Filter* to answer multiple path queries. In particular, we report the relative performance of *Y-Filter* with respect to *Index-Filter* (i.e., we divide *Y-Filter*'s execution time by that of *Index-Filter*). Hence, ratios that are lower than 1 correspond to cases in which *Y-Filter* is more efficient, and ratios greater than 1 correspond to cases in which *Index-Filter* is more efficient.

4.4. Techniques Compared

In our experiments, we compare the techniques presented in Section 3 against each other, and also against other proposed approaches in the literature for processing XML queries one at a time. In particular, we will study the following techniques:

- *Y-Filter*: The navigation-based technique of Section 3.2 augmented with our stack-based extensions to return all matches.
- *Index-Filter*: The index-based technique of Section 3.3. As explained before, *Index-Filter* uses indexes built over certain tags of the input XML document. In some situations the relevant indexes can be already materialized for *Index-Filter* to use (e.g., when the input documents are static and we receive batches of input queries to process). Other times (e.g., when processing multiple streaming documents), relevant indexes must be built on the fly before using *Index-Filter*. In the next section we compare these two scenarios in detail.
- *PathStack*: Both *Y-Filter* and *Index-Filter* combine query commonalities in a prefix tree to speed up query processing (see Section 3). To evaluate the effectiveness of multi-query processing algorithms, we use *PathStack* [5], the state-of-the-art algorithm to answer individual path queries, as a baseline technique. To process multiple queries using *PathStack*, we simply executed each query in the workload separately and then aggregated the results.

5. Experimental Results

We now report the results we obtained with the experimental setting of Section 4. We ran all experiments on a 550Mhz Pentium III processor with 768MB of main memory. In Section 5.1, we compare *PathStack* against our algorithm *Index-Filter*, for varying number of input queries. Then, in Section 5.2 we present the main experimental results comparing the navigation-based technique, *Y-Filter*, against our index-based technique, *Index-Filter*.

5.1. *Index-Filter* vs. *PathStack*

In this section we compare *PathStack* [5], the state-of-the-art algorithm to answer individual path queries, against our proposed algorithm *Index-Filter*, when varying the number of queries asked. In [5] it is shown that *PathStack* is CPU and I/O optimal among all sequential algorithms that read the complete input. Therefore, an important validation is to compare *PathStack* against our algorithm *Index-Filter*, which was specifically designed to answer multiple queries simultaneously. As we can see in Figure 9, *PathStack* is slower than *Index-Filter* when the number of input queries is increased. With a small number of queries it is almost impossible to distinguish between the algorithms. However, for a large number of queries *Index-Filter* results in execution times that are three to five times more efficient than those of *PathStack*. As expected, *Index-Filter* takes advantage of query commonalities by using the prefix tree representation to avoid processing the same portions of similar queries multiple times.

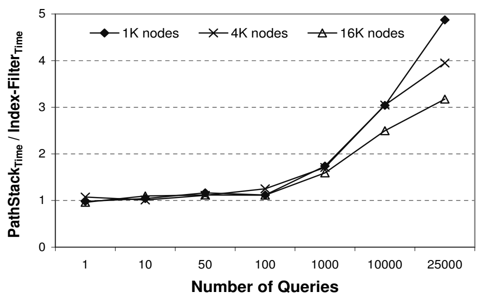


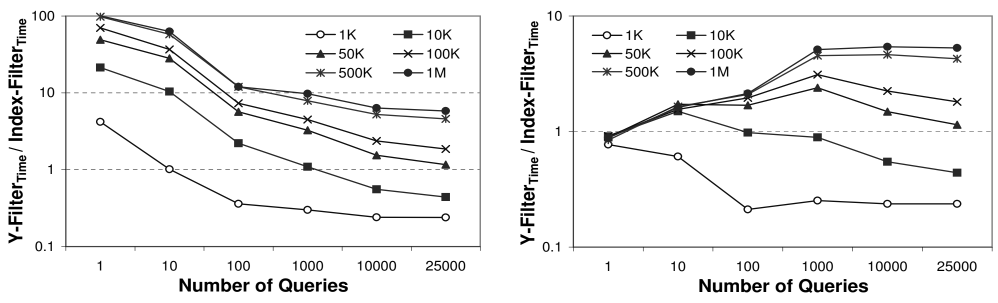
Figure 9. *Index-Filter* vs. *PathStack* for *Random* data.

5.2. *Index-Filter* vs. *Y-Filter*

We now present experimental results comparing *Y-Filter* against *Index-Filter* for a variety of scenarios. In particular, Figures 10, 11, and 12 show the performance of *Y-Filter* relative to that of *Index-Filter* (as explained in Section 4.3) for the *DBLP*, *TPC-H*, and *Random* document families, respectively. Each figure shows two scenarios: one in which the indexes for *Index-Filter* are materialized in advance, and one in which *Index-Filter* creates the indexes on the fly for each incoming document.

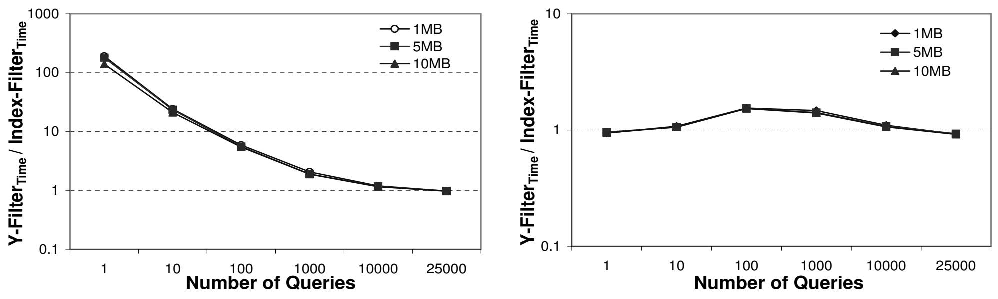
In general, when the number of input queries is small (i.e., fewer than 500 queries), *Index-Filter* is much more efficient than *Y-Filter* if the required indexes are already materialized (see Figures 10(a), 11(a), and 12(a)). The reasons for this behavior are as follows. First, in these scenarios *Index-Filter* effectively traverses a small fragment of the input document, since it only processes the indexes whose tags are present in the input queries. Also, since each node in the prefix tree is relatively sparse (due to the moderate number of input queries), the efficiency of the priority queues is comparable to that of *Y-Filter*'s hash tables. Finally, for larger document sizes, *Index-Filter* takes advantage of the structural (containment) properties of the index elements to avoid processing significant portions of the document that are guaranteed not to participate in any match.

In contrast, when we continue increasing the number of queries, the situation is reversed. The nodes in the prefix tree become more and more populated, and *Y-Filter*'s hash tables start scaling better than *Index-Filter*'s priority queues. Moreover, the prefix tree becomes larger, and *Index-Filter* spends more time analyzing its structure to decide which nodes to process next. For those reasons, in the scenario explained above, *Y-Filter* results in faster executions than *Index-Filter*, especially when using small documents. The tradeoff involving number of queries and document sizes mentioned above is further illustrated in Figure 13 when the lines cross each



(a) Ratio not including index creation. (b) Ratio including index creation.

Figure 10. *Y-Filter* vs. *Index-Filter* for *DBLP* data.



(a) Ratio not including index creation. (b) Ratio including index creation.

Figure 11. *Y-Filter* vs. *Index-Filter* for *TPC-H* data.

other. The figure shows the absolute execution times for both algorithms over the different document families [8].

When we also consider the time spent for building indexes on the fly, the gap between both algorithms is reduced. If the number of queries is large, the results are similar to the precomputed-index case, because the index creation cost is small compared to the cost of answering the queries. In contrast, the largest differences between both scenarios occur for small numbers of queries. For instance, in Figure 10(a) *Index-Filter* is close to 100 times more efficient than *Y-Filter* for answering a single query over large documents, and in Figure 10(b) both techniques behave roughly the same. In fact, for a small number of input queries the creation of indexes is the actual bottleneck of *Index-Filter*. In any case, it is interesting to note that even when indexes over the input documents need to be created on the fly to answer queries, *Index-Filter* is still more efficient than *Y-Filter* in several situations. This behavior has an analogous counterpart in traditional relational query processing, where sometimes the most efficient plan for a join query is to create an index over one operand and then use an index-based join to get the results.

Finally, it is interesting to note that for the *TPC-H* documents (Figure 11), the relative performance of both *Index-Filter* and *Y-Filter* algorithms is insensitive to the document size. This is surprising given the large variations between both algorithms when considering other document families, and the fact that *Index-Filter* and *Y-Filter* are based on significantly different approaches. We believe that this conduct is caused by the flat and highly regular structure of the *TPC-H* documents (which are exported from traditional relational data sets). We plan to further investigate this behavior in future work.

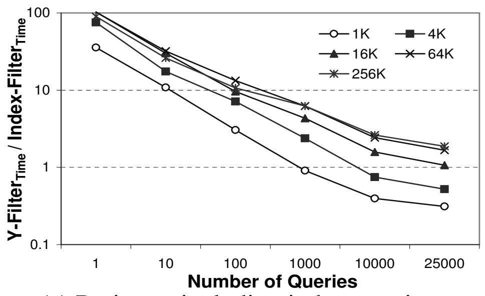
6. Related Work

In the context of semistructured and XML databases, query evaluation and optimization has attracted a lot of research attention. In particular, work done in the Lore DBMS [17, 18, 20] and the Niagara system [19] has considered various aspects of query processing on such data. XML data and various issues in their storage and query processing using relational DBMSs have been considered in [11, 12, 13, 22]. In [13, 22], the mapping of XML data to a number of relational tables was considered along with translation of a subset of XML queries to relational queries.

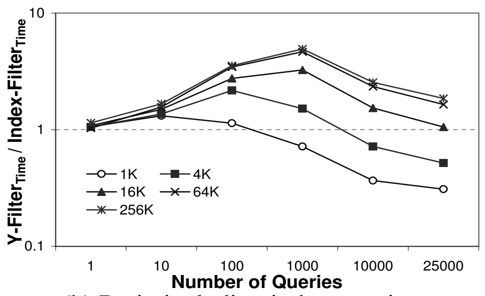
The representation of positions of XML elements is similar to that of [7, 8], who considered a fragment of the PAT text searching operators for indexing text databases. This representation was used to compute containment relationships between "text regions" in the text databases. The focus of that work was on theoretical issues, without elaborating on efficient algorithms for computing these relationships.

References [2, 5, 25] introduced various index-based structural join algorithms as primitives for matching a single path or twig query against an XML document. In particular,

[8]Figure 13(c) shows results for varying number of queries over the *TPC-H* documents since we obtained almost the same ratios for both algorithms when varying document sizes.

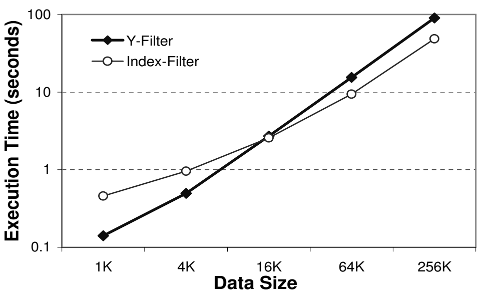


(a) Ratio not including index creation.

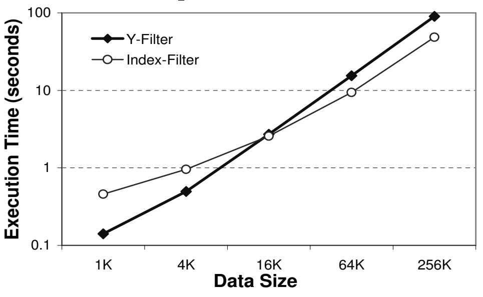


(b) Ratio including index creation.

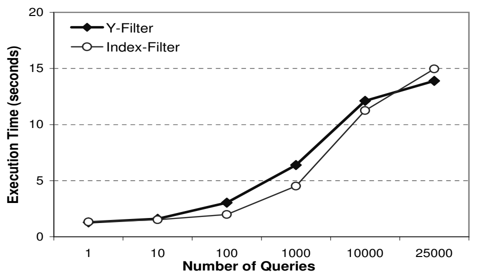
Figure 12. *Y-Filter* vs. *Index-Filter* for *Random* data.



(a) 25,000 queries and *DBLP* data.



(b) 25,000 queries and *Random* data.



(c) 5M document and *TPC-H* data.

Figure 13. Execution times for *Index-Filter* and *Y-Filter*.

[2, 25] proposed binary structural join algorithms as primitives for matching twig queries. While [25] uses an adaptation of a merge-sort technique to process the input documents, reference [2] introduces stack-based algorithms that are I/O optimal for the binary case. The algorithms in [5] are generalizations of the MPMGJN algorithm of [25] to match path queries, and the algorithms of [2] to match path and twig queries. The main contribution of the algorithms in [5] is that no large intermediate results are generated for complex path or twig queries, eliminating the need for an optimization step that was needed when stitching together partial results from the algorithms in [2, 25]. Our *Index-Filter* algorithm of Section 3.3 is loosely based on the *TwigStack* technique [5].

Finally, [3, 6, 9, 14, 15] proposed various navigation-based techniques to match single and multiple, path and twig queries. Reference [14] introduces the X-Scan operator, which matches path expression patterns over a streaming (non materialized) XML document. References [3, 9] consider the problem of answering multiple path queries over incoming documents. The algorithms and data structures in both [3] and [9, 10] are tailored for the case of very large numbers of queries and small input documents. While [3] uses separate finite state machines to represent each query, the follow-up work [9, 10] compresses the pool of input queries by sharing prefixes, as explained in Section 3.1. It is interesting to note that reference [3] presents an optimization of the main algorithm called *prefiltering*, which can be seen as a simple index-based preprocessing step that takes into account the occurrence of tags but not the structure of the incoming XML documents. The idea of prefiltering is to eliminate from consideration any query that contains an element tag that is not present in the input document, and it is adapted from previous algorithms for filtering plain text documents [24]. Reference [6] proposes a trie-based data structure, called XTrie, to support filtering of complex twig queries. The XTrie, along with a sophisticated matching algorithm, are able to reduce the number of redundant matchings. We note that the query model in [3, 9, 6] is slightly different from ours. They are mainly concerned on the set of queries for which at least one match exists (therefore several optimizations are available to avoid processing queries beyond their first match). In contrast, in this work we are interested in returning the set of all matches for each input query. Finally, reference [15] addresses the problem of obtaining all matches for a set of path and tree queries. The algorithms use an index structure, denoted the "requirements index," which helps to quickly determine the set of queries for which a certain structural relationship (e.g., parent-child, or ancestor-descendant) is relevant. The main difference with our query model is that in [15] each input query identifies a unique "distinguished" query node, so the result matches are 1-ary relations. The algorithms in [15] make at most two passes on the input document, and provide performance guarantees on the number of I/O invocations required to find the resulting matches.

7. Conclusions and Future Work

In this paper we studied algorithms to answer multiple path queries over XML documents efficiently. In particular, we reviewed *Y-Filter*, a state-of-the-art *navigation-based* algorithm. *Y-Filter* computes results by analyzing an input document stream one tag at a time, typically by using SAX-based parsers. We extended *Y-Filter*'s original formulation so that it returns all matches for the set of input queries. We also introduced a novel *index-based* algorithm, *Index-Filter*, which avoids processing portions of the XML document that are guaranteed not to be part of any match. *Index-Filter* takes advantage of precomputed indexes over the input document, but can also build the indexes on the fly. Finally, we compared *Y-Filter* and *Index-Filter*, both conceptually and experimentally. We showed that both techniques have their advantages, and we discussed the scenarios under which each technique is superior to the other one. In particular, we showed that while most XML query processing techniques work off SAX events, in some cases it pays off to parse the input document in advance and augment it with auxiliary information that can be used to evaluate the queries faster. As part of future work, we plan to extend our results to cover more general XML queries and incorporate other recent navigation-based algorithms from the literature (e.g., [6, 15]). We also plan to study other sharing schemes for path queries and the applicability of such schemes on each approach.

References

[1] A. Aboulnaga, J. Naughton, and C. Zhang. Generating synthetic complex-structured XML data. In *Proceedings of the WebDB'01 Workshop*, 2001.

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of the 2002 International Conference on Data Engineering*, 2002.

[3] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 2000 International Conference on Very Large Data Bases*, 2000.

[4] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML query language. In *W3C Working Draft. Available from* `http://www.w3.org/TR/xquery`, Dec. 2001.

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002.

[6] C. Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of the 2002 International Conference on Data Engineering*, 2002.

[7] M. Consens and T. Milo. Optimizing queries on files. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, 1994.

[8] M. Consens and T. Milo. Algebras for querying text regions. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1995.

[9] Y. Diao, P. Fischer, M. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *Proceedings of the 2002 International Conference on Data Engineering*, 2002.

[10] Y. Diao, H. Zhang, and M. Franklin. NFA-based filtering for efficient and scalable XML routing. Technical report, Computer Science Division, University of California, Berkeley, 2002.

[11] M. Fernandez and D. Suciu. SilkRoute: Trading between relations and XML. *WWW9*, 2000.

[12] R. Fiebig and G. Moerkotte. Evaluating queries on structure with access support relations. *Proceedings of WebDB'00*, 2000.

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

[14] Z. Ives, A. Levy, and D. Weld. Efficient evaluation of regular path expressions on streaming XML data. *Technical report, University of Washington*, 2000.

[15] L. V. S. Lakshmanan and S. Parthasarathy. On efficient matching of streaming XML documents and queries. In *Proceedings of EDBT*, 2002.

[16] M. Ley. DBLP. Computer Science Bibliography. *Available at* `http://www.informatik.uni-trier.de/~ley/db`.

[17] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3), 1997.

[18] J. McHugh and J. Widom. Query optimization for XML. In *VLDB'99, Proceedings of the 25th International Conference on Very Large Data Bases*, 1999.

[19] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24(2), 2001.

[20] D. Quass, J. Widom, R. Goldman, K. Haas, Q. Luo, J. McHugh, S. Nestorov, A. Rajaraman, H. Rivero, S. Abiteboul, J. D. Ullman, and J. L. Wiener. Lore: A lightweight object repository for semistructured data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996.

[21] G. Salton and M. J. McGill. *Introduction to modern information retrieval*. McGraw-Hill, 1983.

[22] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. In *Proceedings of the 2000 International Conference on Very Large Data Bases*, 2000.

[23] TPC Benchmark H. Decision support. *Available at* `http://www.tpc.org`.

[24] T. W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *TODS*, 19(2), 1994.

[25] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, 2001.