Scalable Filtering of Multiple Generalized-Tree-Pattern Queries over XML Streams

Songting Chen, Hua-Gang Li, Jun’ichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan

Abstract—An XML publish/subscribe system needs to filter a large number of queries over XML streams. Most existing systems only consider filtering the simple XPath statements. In this paper, we focus on filtering of the more complex Generalized-Tree-Pattern (GTP) queries. Our filtering mechanism is based on a novel Tree-of-Path (TOP) encoding scheme, which compactly represents the path matches for the entire document. First, we show that the TOP encodings can be efficiently produced via shared bottom-up path matching. Second, with the aid of this TOP encoding, we can 1) achieve polynomial time and space complexity for postprocessing, 2) avoid redundant predicate evaluations, 3) allow an efficient duplicate-free and merge join-based algorithm for merging multiple encoded path matches, and 4) simplify the processing of GTP queries. Overall, our approach maximizes the sharing opportunity across queries by exploiting the suffix as well as prefix sharing. At the same time, our TOP encodings allow efficient postprocessing for GTP queries. Extensive performance studies show that GFilter not only achieves significantly better filtering performance than state-of-the-art algorithms but also is capable of efficiently filtering the more complex GTP queries.

Index Terms—XML filtering, XML streams, generalized-tree-pattern queries, result encoding.

1 INTRODUCTION

XML has become the de facto standard for data exchange. XML documents can be viewed as ordered tree structures. A sample XML tree is shown in Fig. 1. Much of the XML data that are prepared for data exchange purposes are made available in the form of XML data streams. In general, such XML data streams are dynamically generated, such as financial tickers, network traffic data, and web services. Since many applications, such as XML publish/subscribe systems [1], [2], [3], typically have real-time or interactive response constraints, such XML data are required to be processed as they are delivered, known as XML streams (Fig. 1). Processing queries over such sequentially accessed XML stream data has received significant attention in the literature recently [1], [2], [4], [5], [3], [6], [7].

In particular, the XML filtering problem is to find matchings from a large number of prestored XPath expressions for a given XML document. In most applications, not only the matched expressions but also their matching elements have to be returned. In other words, the extraction and transformation of the XML document is as important [8]. However, most existing works only consider XPath, while such transformation calls for more sophisticated query language such as XQuery. In this work, we focus on the more complex Generalized-Tree-Pattern (GTP) queries [9], [10], which is a key construct to support XQuery. Some sample GTP queries are shown in Fig. 2. More details about GTP will be discussed in Section 2. To our knowledge, the only XML filtering work that considers beyond XPath is [8], where a restricted subset of GTP queries (with a single for-binding node) is considered.

The key issue for XML filtering is the scalability. That is, there are a large number of queries to be processed simultaneously. As mentioned by a number of existing works, sharing is the key to achieve such scalability [11], [2], [3], [7], [12]. Among these works, path sharing has been proven to be effective to maximally exploit the sharing opportunity across different queries, and is thus scalable to handle millions of queries [11], [2], [3], [7]. However, supporting the more complex query language calls for various types of postprocessing and is nontrivial for the path-sharing-based approaches. Below, we list a number of common issues.

Time and space complexity. Note that path matches have to be enumerated before any postprocessing [3], [7]. In the worst case, there are exponential number of path matches when the queries are recursive with regard to the XML data. This poses a fundamental challenge for any path-sharing-based XML filtering scheme.

Predicate evaluation. When processing a single query, the predicate evaluation can be done on the fly even before any path matching. However, such approach is no longer appropriate in XML filtering scenario, since it may destroy the path-sharing opportunity across queries and introduce a significant extra bookkeeping cost [3]. Hence, predicate
evaluations [3] (and its shared processing [8]) are often considered after shared path matching.

**Path join for twig queries.** In order to support the tree pattern (twig) queries, the path matches need to be enumerated and joined together. When the number of matches is large, the cost of enumerations and subsequent joins is nontrivial. While holistic twig join has been extensively studied for processing a single query [15], [4], [7], [13], they are not directly applicable here. The reason is due to the following inherent discrepancy that the holistic approach takes the entire tree as a whole, while path-sharing scheme requires to consider each branch separately.

**More expensive operations for GTP queries.** In order to support the more complex GTP queries, various postprocessing operations such as duplicate elimination, sort, outer-join, and grouping have to be introduced. We will describe these operations in more details in Section 2.

Hence, while path-sharing scheme maximizes the sharing opportunity across queries, it calls for efficient postprocessing solution to address the above challenges. In this paper, we discover a unique opportunity for optimizing the postprocessing performance, illustrated via a motivating example below.

**Example 1.** Consider GTP2 (Fig. 2) over the XML document in Fig. 1. There are six matches for /A//B//C, namely, (a1, b1, c1), (a1, b2, c1), (a1, b3, c1), (a1, b1, c2), (a1, b2, c2), and (a1, b3, c2). The predicate “B[@p = 1]” is evaluated twice on b1, b2, and b3 during postprocessing. This is obviously rather undesirable. An analogous issue arises when we process query GTP1, where redundant join probes are evaluated. Such redundant postprocessing cannot be resolved through sharing, as it is inherent in processing a single query.

In this work, we provide a comprehensive solution to address the above challenges. In summary, the main contributions of this work are as follows:

- We develop a novel Tree-of-Path (TOP) encoding scheme to compactly represent the path matches for the entire XML document, which has a polynomial time and space complexity. The postprocessing of the complex GTP queries can proceed based on this encoding scheme without enumerating the possibly exponential number of path matches.
- We show how to efficiently obtain TOP encodings via bottom-up path matching. Furthermore, we propose to share the common suffix among multiple queries. At the same time, we can also exploit the common prefix sharing within a single GTP query.
- We demonstrate how to efficiently support predicate, join evaluation based on our TOP encoding scheme. Put together, we propose efficient filtering method for GTP queries. To our knowledge, the query language we support, i.e., GTP query, is more powerful than any existing XML filtering solutions.

Extensive performance studies on various data sets and queries show that our GFilter solution not only achieves significantly better filtering performance than state-of-the-art filtering algorithm, such as YFilter [3], [8] and AFilter [7], but is also capable of efficiently processing the more complex GTP queries.

**2 Background**

One fundamental task for processing XPath and XQuery is to match twig pattern queries. The concept of GTP is introduced in [10] (a similar concept called XNAV tree in [9]) to consider the evaluation of an XQuery as a whole to avoid repetitive work.

As shown in Fig. 2, GTP query may have solid and dotted edges, representing mandatory and optional structural relationships, respectively. In this paper, we consider parent-child (PC) and ancestor-descendant (AD) relationships. The mandatory semantics corresponds to those path expressions in the FOR or WHERE clauses. The optional semantics corresponds to those path expressions in the LET or RETURN clauses. For a given GTP, not all nodes are binding nodes. For the path expressions in the FOR clause, the last node is the *for-binding* node (the *B* node of GTP1 in Fig. 2). For the path expression in LET or RETURN clause, the last node is the *group-binding* node, which means that we need to group the matching elements under their
common ancestor element (the $C$ node of GTP$_1$). Please refer to [9] and [10] for details of GTP.

These rich semantics introduce new challenges for handling the duplicates and ordering issues. We now briefly review how the query results are generated for GTP queries through the following three examples:

1. Consider the document tree in Fig. 1 and the query GTP$_3$ in Fig. 2. The path query \(///B//C\) has six matches, namely, $(b_1,c_1), (b_2,c_1), (b_3,c_1), (b_1,c_2), (b_2,c_2),$ and $(b_3,c_2)$. However, since $B$ is the only for-binding node, the results should be $(b_1), (b_2),$ and $(b_3)$. Clearly, duplicate elimination has to be performed and is nontrivial because the matches are not sorted on $B$.

2. Next consider the GTP$_1$ in Fig. 2. The path query \(///A//B//D\) has two matches, namely, $(a_1,b_1,d_2)$ and $(a_1,b_2,d_1)$. If only $D$ is the for-binding node, we should return $(d_1)$ and $(d_2)$. This order is different from the order for the entire path matches. If we were to generate these entire path matches first, sorting these $D$ elements becomes unavoidable.

3. Finally, in order to support the optional axis and the group-binding node, an outer-join has to be performed between the paths \(///B//C\) and \(///B//D\) and subsequently the matching $D$ elements have to be grouped together [8]. For example, we get the result $(b_1,\{c_1,c_2\})$ for GTP$_1$ by grouping $(b_1,c_1)$ and $(b_1,c_2)$. Both outer-join and grouping are nontrivial operations. In this paper, we show that these expensive operations can be avoided via a novel encoding scheme.

### 3 Bottom-Up Path Matching

Most existing works on XML filtering [11], [3], [6] employ a finite state automata (either NFA or DFA)-based approach to evaluate path queries. They evaluate the path queries top-down, i.e., from root to leaf, as shown in Fig. 3a.

In this paper, we propose a bottom-up path matching solution, i.e., evaluate the path query from leaf to root. The benefits are twofold. First, the bottom-up approach benefits from the heuristics that the leaf selectivity is often more stringent [7]. As can be seen in Fig. 3, the top-down approach will consider all paths underneath $b_1$ even when most of them do not contain any $C$ element. Second, we propose a TOP encoding scheme for compact representing the path matches, which has a polynomial time and space complexity and enables efficient postprocessing. This encodings can be efficiently produced via bottom-up path matching.

Unlike the top-down approach, such as YFilter [3], where the matching of common document prefix is naturally shared between different document paths, here special treatment is needed for bottom-up path matching. When we first visit $c_1$ in Fig. 3, if we were to evaluate the entire path from $c_1$ to root, the common document prefix may have to be reevaluated for each $c_i$. For this, we propose to delay the processing of the common document prefix until all the paths from $c_i$ to $b_1$ have been processed, i.e., when the end-tag of $b_1$ is visited (see Fig. 3b). This calls for a postorder document traversal, i.e., the order of end-tags in the XML streams.

While the top-down approaches call for the sharing of common prefixes among multiple queries [3], [6], bottom-up approaches naturally call for the sharing of common suffixes [7]. As mentioned in Section 1, we ignore the value predicates as well as how the nodes are bound (e.g., for grouping) when creating the shared plan in order to maximize the structural sharing opportunities. Fig. 4 shows the shared suffix plan for the GTP queries in Fig. 2. Take GTP$_1$ for example, two paths can be decomposed, namely, ////B//C and ////B//D. One unique feature of our technique is that we can also exploit the common prefix sharing within a single GTP query (Section 4.1). That is, we only need to generate ////B//C and ////B//D to save the path matching cost. For now, we delay the description of how the query plan is generated and focus on the path matching part assuming the plan is already created. We call each query node in the shared suffix query plan a suffix query node. Here, $S_i$ is the identity for each shared suffix query node and “+” denotes that it is root node of some queries.

For each suffix query node $S_i$ in the shared plan, $PCParent(S_i,l)$ denotes $S_i$’s parent with PC (parent/child) axis, whose label is $l$, while $ADParent(S_i,l)$ denotes $S_i$’s parent with AD (ancestor/descendant) axis, whose label is $l$. If $S_i$ is not parent of any other node, we call it leaf suffix query node. In Fig. 4, $ADParent(S_1,B) = S_2$, $PCParent(S_1,B) = S_0$, and $S_1, S_3,$ and $S_6$ are leaf suffix query nodes.

#### 3.1 Shared Bottom-Up Path Matching

Fig. 5 depicts the intuition for bottom-up path matching. Assume that $S_p$ has a child suffix query node $S_c$ and the axis of $S_p \rightarrow S_c$ is PC. Then, a document element $n$ can be a
match to \( S_p \) only when there exists a document element \( c \) that matches \( S_t \) and is child of \( n \). Similarly, assume \( S_q \) has a child suffix query node \( S_d \) and the axis of \( S_q \rightarrow S_d \) is AD. Then, a document element \( n \) can be a match to \( S_q \) only when there exists a document element \( d \) that matches \( S_d \) and is descendant of \( n \).

To achieve shared common document prefix matching mentioned earlier, we delay the matching for \( n \) and simply collect all the suffix nodes \( \{ S_q \} \) that are satisfied by \( n \)'s children (referred to as \( n \)'s PCTable) and all the suffix nodes \( \{ S_d \} \) that are satisfied by \( n \)'s descendants (referred to as \( n \)'s ADTable). Here, by all the suffix nodes \( \{ S_q \} \) or \( \{ S_d \} \), we mean the simultaneous execution of all the path queries in the shared suffix query plan.

As can be seen in Fig. 6, the matching suffix query nodes for \( n \) are \( PCParent(S_q, n) \) and \( PCParent(S_q, */*) \) for any \( S_q \) in PCTable, and \( ADParent(S_q, n) \) and \( ADParent(S_q, */*) \) for any \( S_q \) in ADTable. Those matching suffix query nodes, namely, \( \{ S_q \} \) and \( \{ S_d \} \), will be put into \( e.PCTable \) and \( e.ADTable \), where \( e \) is \( n \)'s parent. Also, all the suffix query nodes \( \{ S_d \} \) need to be copied into \( e \)'s ADTable.

Fig. 7 depicts the pseudocode of our bottom-up path matching algorithm. Here, \( docPath \) is a stack that contains those document elements with only their start-tags being visited. When a start-tag SAX event is generated, we push the document element into \( docPath \). When an end-tag SAX event is generated, we pop the top document element from the stack and call the path matching algorithm.

As can be seen, when the end-tag of \( n \) is seen, we compute the suffix query nodes that \( n \) satisfies. First, those leaf suffix query nodes with the matching labels \(^2\) are satisfied (lines 2-3 in Fig. 7). Next, based on the intuition in Fig. 6, other matching suffix query nodes of \( n \) include \( PCParent(S_q, n) \) and \( PCParent(S_q, */*) \) for any \( S_q \) in PCTable, and \( ADParent(S_q, n) \) and \( ADParent(S_q, */*) \) for any \( S_q \) in ADTable (lines 4-9). All the matching suffix query nodes need to be put into \( e.PCTable \) and \( e.ADTable \), where \( e \) is \( n \)'s parent (the propagateBoth procedure). In our implementation, we use hash table to implement \( PCParent \) and \( ADParent \) for fast lookup.

As an optimization, \( S \) is put to \( e.ADTable \) if \( S.ADParent \) is not empty or \( S \) is root of some queries started with "//". Similar optimization applies to \( e.PCTable \) (lines 5 and 7).

Lastly, the suffix query nodes in \( n.ADTable \) will be put into \( e.ADTable \) (the propagateAD procedure).

Example 2. Fig. 8 describes a running example for processing the XML document (Fig. 1) based on the shared plan in Fig. 4. First, \( c1 \) only matches \( S_1 \) since its PCTable and ADTable are empty. \( S_1 \) is then put into \( b3.ADTable \) (not \( b3.PCTable \) since \( S_1 \) only has AD parent). When visiting \( b3 \), since \( S_1 \) is in \( b3.ADTable \), \( S_1 \) is satisfied in addition to \( S_0 \). Now, \( S_1, S_2 \), and \( S_0 \) need to be put into \( b2.ADTable \). Eventually, \( a1 \) satisfies \( S_3 \) and \( S_7 \), which will be put into rootList. Now, the path queries rooted at \( S_3 \) and \( S_7 \) are satisfied. Also, those queries rooted at \( S_2 \) and started with "//" (\( S_2 \) in \( a1.ADTable \)) are satisfied.

3.2 Piggybacking Tree-of-Path Encodings

The path matching algorithm in Fig. 7 can only find the matching expressions rather than the matching elements. Inspired by the existing stack-based encoding schemes [15], [4], [7], [13], we propose to integrate a novel result encoding scheme into the path matching algorithm. In the earlier works, the results are encoded through recording the AD relationship between the elements that satisfy two adjacent query nodes (by using pointers), as well as the AD relationship between those that satisfy the same query node (by using stacks).

To capture the AD relationship between elements that satisfy different suffix query nodes, we extend ADTable and PCTable to contain the matching elements. We denote \( e.PCTable[S] \) as all \( e \)'s child elements that satisfy the suffix

---

\(^2\) There exist at most two such nodes in the shared suffix plan. One is the wildcard *, the other is the query node \( N \) with the matching label of \( n \). In other words, \( n \) always satisfies the following two queries, "/N" and "/*".
query node S and denote e.ADTable[S] as all e’s descendant elements that satisfy S.

Next, we need to capture the AD relationship between the elements that satisfy the same suffix query nodes. It is obvious that e.PCTable[S] simply contains a list of document elements with no AD relationship. In comparison, the elements in e.ADTable[S] may have AD relationship, which can be captured by the following TOP encoding scheme.

**Definition 1.** A matching tree for suffix query node S can be either n, (n//), or [n//]. Here, n is a document element that satisfies S. (n//) denotes all n’s descendant elements that satisfy S. [n/] combines the semantics of n and (n//). The difference between (n//) and [n//] is whether n satisfies the suffix query node S or not.

By definition, e.ADTable[S] equals (e//) with regard to S. Fig. 9 depicts one example how this TOP encoding scheme captures the AD relationship between the elements that satisfy the same query node. As can be seen, e has two matching trees that satisfy S, n1, and [n2], respectively. From [n2], we know that there exists n3’s descendants that also satisfy S, which can be found in n2.ADTable[S] = \{n3, (n6)\}. Note that n3 satisfies S while n6 does not. Next, both [n3] and (n6) can be expanded further similarly. Finally, n1, n2, n3, n4, n5, n7, and n8 are e’s descendant elements that satisfy S.

**Procedure propagateBoth(suffixNode S, docElement e)**
1. BEGIN
2. IF n is document root and S is root node of some queries
3. Put S into rootList;
4. ELSE
5. IF S has any PC parent
6. Add n to docPath[n.level-1][S];
7. IF S has any AD parent or S is root of some queries started with "/";
8. IF n.ADTable[S] is not empty
9. Add (n//) to docPath[n.level-1][ADTable[S];
10. ELSE
11. Add n to docPath[n.level-1][ADTable[S];
12. END

**Procedure propagateAD(suffixNode S, docElement n)**
1. BEGIN
2. IF n does not satisfy S
3. If n.ADTable[S] contains more than one matching tree
4. Add (n//) to docPath[n.level-1][ADTable[S];
5. ELSE
6. Add n ADTable[S] to docPath[n.level-1][ADTable[S];
7. END

The TOP encoding scheme theoretically is equivalent to the hierarchical stack encoding proposed in our prior work [13] (Fig. 9). However, the step of creating the stack hierarchy by merging stacks is redundant to the path matching algorithms. On the other hand, the TOP encodings not only are more succinct and completely avoid creating any stacks but also can be cheaply integrated into the path matching algorithms.

The key idea to integrate TOP encoding into the path matching algorithm is as follows: Assume that n’s parent element is e. When n satisfies the suffix query node S, the matching algorithm in Fig. 7 will report S to e. In fact, we can report the matching elements of S to e (in a TOP form) as well. Again, take Fig. 9 as one example. When visiting n3 in postorder, we find that n3 satisfies S and its n3.ADTable[S] is not empty. In this case, the TOP encoding [n3] will be appended to n2.ADTable[S]. When visiting n6, n6 does not satisfy S but there are descendants of n6 that satisfy S. In this case, the TOP encoding (n6) will be appended to n2.ADTable[S]. An alternative is to copy every tree in n6.ADTable[S] into n2.ADTable[S], which may not be efficient as n6.ADTable[S] could contain an arbitrary long list.

Fig. 10 depicts the details how the TOP encoding scheme can be piggybacked into the path matching algorithm. Assume that n’s parent element is e. If n satisfies S, first, it will be appended to e.PCTable[S]. Next, if n.ADTable[S] is not empty, then (n//) would be appended into e.ADTable[S]. Otherwise, n would be appended to e.ADTable[S]. Finally, the content in n.ADTable will be appended into e.ADTable as well. If n.ADTable[S] contains more than one matching tree,

**Fig. 8. Running example.**

**Fig. 9. TOP encoding versus hierarchical stack encoding [21].**
Example 3. Let us reconsider the running example in Fig. 11 when copying all these trees.

Fig. 11. Running example: TOP encodings.

then only \((n/\)\)) would be put into \(e.ADTable[S]\) rather than copying all these trees.

Example 3. Let us reconsider the running example in Fig. 11 when the TOP encoding scheme is piggybacked into the matching algorithm. As can be seen in Fig. 11, when visiting \(b_3\), since \(b_3.ADTable[S_1]\) contains more than one matching tree, \((b_3//)\) will be put into \(b_2.ADTable[S_1]\). When visiting \(b_2\), since it satisfies \(S_2\), \(S_6\) and \(b_2.ADTable[S_2]\) and \(b_2.ADTable[S_6]\) are not empty, \((b_2//)\) will be appended into \(b_1.ADTable[S_2]\) and \(b_1.ADTable[S_6]\), respectively. The semantics of \((b_2//)\) is that both \(b_2\) and its descendants in \(b_2.ADTable[S_2]\) \((b_2.ADTable[S_6])\) satisfy \(S_2\) \((S_6)\).

Note that the matching results for all the path queries are already encoded in Fig. 11. Assume that \(n\) satisfies suffix query node \(S_n\). For any PC child \(S_n\) of \(S_n\), \(n.PCTable[S_n]\) records all the matching elements with regard to one query step \(S_n/\)\(S_n\) for \(n\). Now assume that \(n\) satisfies suffix query node \(S_n\). For any AD child \(S_n\) of \(S_n\), \(n.ADTable[S_n]\) records all the matching elements with regard to one query step \(S_n/\)\(S_n\) for \(n\). For instance, in Fig. 11, \(a_1\) satisfies \(S_1\). Its corresponding matches for \(S_2\) are

\[
a_1.ADTable[S_2] = \{[b_1//]\} = ([b_1,a_1.ADTable[S_2])
\]

\[
= ([b_1,b_2,b_2.ADTable[S_2]]) = ([b_1,b_2,b_3]).
\]

Example 4. Fig. 12 shows how the results of path query 
\(//A//B//C\) are compactly encoded. Here, the dotted edges represent the matching elements with regard to one query step, while the solid edges represent the internal hierarchy of the TOP encodings. First, \(a_1\) satisfies \(S_1\). The matches with regard to its child node \(S_2\) are \(a_1.ADTable[S_2] = \{[b_1//]\} = ([b_1,b_2,b_3])\). Next, \(b_1.ADTable[S_1], b_2.ADTable[S_1], b_3.ADTable[S_1]\) further provide the matches to their child query node \(S_1\). The right side of the figure provides a clearer view of how the total six path matches are compactly encoded.

Fig. 12. Example of result encoding for 
\(//A//B//C\).

We explicitly implement those pointers to avoid too many hash lookups during postprocessing. Assume that the document element \(n\) satisfies the suffix query node \(S_n\). We refer to the solid pointers as \(n.getChildSOT(S_n)\) which gives the descendant elements of \(n\) that also satisfy \(S_n\), while the dotted pointers as \(n.getMatch(S_n)\) which gives the matches with regard to one query step \(S_n /\)\(S_n\).

Sequence of Trees (SOT). The matching elements in \(e.PCTable[S_n]\) and \(e.ADTable[S_n]\) are organized in a TOP format and in document order, which we refer to as SOT, i.e., sequence of TOP trees.

Time and space complexity. The time and space complexity for TOP encodings is polynomial \((O(|Q|D))\), where \(Q\) is the size of the shared query plan and \(D\) is the size of the document. That is, in the worst case, each document element’s PCTable and ADTable holds all suffix query nodes. In the rest of this paper, we show that we can process the GTP queries based on the TOP encodings. This effectively resolves the exponential path enumeration problem.

4 GFILTER: EFFICIENT POSTPROCESSING

4.1 Decomposition of GTP Queries

As a first step, we decompose the GTP query into a set of paths. In this work, instead of creating full path for each leaf node, we also exploit the prefix sharing within the same GTP query. That is, when we enumerate the leaf to root path, we stop if the current query node has already been enumerated for other paths. Take the query GTP1 in Fig. 13 for example. We only need to match the partial path 
\(//B//D\) instead of the full path 
\(//A//B//D\).” The matches for 
\(//A//B//D\) can be obtained as follows: For any element \(a\) that satisfies \(S_n\), whether it also has matches for 
\(//A//B//D\) or not can be determined by checking

Fig. 13. Decomposition of GTP queries.
The inputs to the algorithm are the elements that satisfy the root query node in a SOT form, such as $a$ in the previous example. Then, it will evaluate the value predicate to each of the element in the SOT (line 3). If the element satisfies the condition, it will go further down to evaluate the child query node $P$ (lines 5-8). Line 7 is to update $n$’s child matches with regard to one query step. Line 8 maintains the output tree structure. Note that there is a chance to skip the entire subtree in the case of AD axis (lines 10-11). Take the query GTP3 in Fig. 2 for example. If we find that $b1$ satisfies $\@p > 0$ but does not have any path matches that satisfy the predicate on $C$, we can skip all $b1$’s descendants ($b2$ and $b3$). The reason is that the path matches below $b2$ and $b3$ must be a subset of those below $b1$.

![Fig. 15. Intuition of m-way merge join.](image)

### 4.2 Evaluating Value Predicates
Let us consider GTP2 in Fig. 2. By our path matching and encoding scheme described in Section 3, we obtain the compact representation of all the path matches as shown in Fig. 11. Now, in order to evaluate the predicate “$B[\@p = 1]$,” instead of checking the predicate for all six path matches, we check $b1$, $b2$, and $b3$ (from “$a.ADTable[S_2]$”) only once.

Fig. 14 depicts the algorithm for predicate evaluation over encoded path matches. It returns the satisfied path matches in the same compact form, which will be used for subsequent path joins if any. When the queries are Boolean ones, this algorithm can be tuned to stop earlier whenever a matching is found.

The inputs to the algorithm are the elements that satisfy the root query node in a SOT form, such as $a1$ in the previous example. Then, it will evaluate the value predicate to each of the element in the SOT (line 3). If the element satisfies the condition, it will go further down to evaluate the child query node $P$ (lines 5-8). Line 7 is to update $n$’s child matches with regard to one query step. Line 8 maintains the output tree structure. Note that there is a chance to skip the entire subtree in the case of AD axis (lines 10-11). Take the query GTP3 in Fig. 2 for example. If we find that $b1$ satisfies $\@p > 0$ but does not have any path matches that satisfy the predicate on $C$, we can skip all $b1$’s descendants ($b2$ and $b3$). The reason is that the path matches below $b2$ and $b3$ must be a subset of those below $b1$.

![Fig. 15. Intuition of m-way merge join.](image)

### 4.3 Evaluating Path Joins
Assume that we have $m$ encoded path matches $P_1, \ldots, P_m$. The matching SOTs of the root node $N$ for each path are $\text{SOT}(S_{N_1}), \ldots, \text{SOT}(S_{N_m})$, respectively, as shown in Fig. 15. Now, these paths need to be joined together at their root node $N$. For this, we only need to perform joins over these SOTs to find those elements that appear in each SOT.

The multiway join over encoded paths works as follows: We maintain a cursor for each SOT. Given a candidate element $n$, we move the cursor at $\text{SOT}(S_{N_n})$ until either a matching is found or an element after $n$ in document order is found. It is possible to skip the entire subtree when moving the cursor. Assume that the cursor currently points to an element $e$. If $e$ is before $n$ in document order and has no AD relationship with $n$ (by checking their region encodings [15] dynamically assigned in XML streams), we can skip the entire subtree of $e$, since none of $e$’s descendants can be $n$.

The details of the join algorithm is straightforward and omitted for space concerns [14].

Performing joins over these SOTs instead of over the full path matches provides the following two benefits. First, the elements in each SOT are duplicate-free. This avoids any redundant join probes as described in Section 1. Second, the elements in SOT are already sorted in document order. Hence, we can apply an efficient merge join algorithm to merge multiple SOTs. Such merge join is applicable even when the path query is recursive with regard to the document schema, a feature that we do not have if the join is operated over the full path matches [8]. As a simple example, when processing GTP1 in Fig. 2, the two SOTs to be joined are $a1.ADTable[S_2] = \{b1, b2, b3\}$ and $a1.ADTable[S_3] = \{b1, b2\}$, respectively.

### 4.4 Handling Nonbinding Nodes
As mentioned in Section 2, the duplicate and out-of-order problems may occur when handling nonbinding nodes in the GTP query. We formulate this as the problem of
computing the distinct child matches for a nonbounding node below.

Problem statement: Assume a nonbounding query node $E$ and its child query node $M$. For a given set of elements $e’SOT$ that match $E$ and maintained in $SOT$ format, we want to compute its distinct child matches on the query node $M$, namely, a set of elements $resultSOT$ that match $M$ and maintained also in $SOT$ format, with each element in $resultSOT$ having at least one element in $e’SOT$ that satisfies the query step $E \rightarrow M$.

When the query step $E \rightarrow M$ is an AD relationship, clearly only the root element of each tree in $e’SOT$ needs to be considered. The final $resultSOT$ is simply a union of all $root\cdot getChildMatch(M)$. All other elements in $e’SOT$ are guaranteed to only generate duplicates. For example, since $b2$ is descendant of $b1$ (Fig. 11), then $b2\cdot ADTable[S1]$ must be a subset (or equal) of $b1\cdot ADTable[S1]$. By only considering the root element in the TOP tree, we can avoid generating duplicates in the first place.

When the query step $E \rightarrow M$ is a PC relationship, a naive way to handle the order problem is to sort all the elements in $e\cdot getChildMatch(M)$ for all $e$ in $e’SOT$. In fact, sorting is not necessary since all the elements $e$ in $e’SOT$ and their child elements in $e\cdot getChildMatch(M)$ already preserved their respective document order by the path matching algorithm.

Fig. 16 provides basic intuition regarding how this order problem can be resolved. Assume that one element $e$ with children $e1, \ldots, en$ in an SOT tree and $e\cdot getChildMatch(M)$ equals $m1, \ldots, mp$. Both $e1, \ldots, en$ and $m1, \ldots, mp$ are in document order. Starting from $e1$ and $m1$, there are three possible positions for $m1: 1) m1$ is on the left side of $e1$. In this case, we should add $m1$ into $resultSOT$ since there will be no other result element that appears before $m1$ in the document order or is a descendant of $m1$. 2) $m1$ is an ancestor of $e1$. In this case, $m1$ must also be an ancestor for all $e1\cdot getChildMatch(M)$ and all $e\cdot getChildMatch(M)$, where $e$ is any descendant of $e1$ in $SOT$. Assume that we recursively compute the distinct child matches of $e1$ and all its descendant elements $e’$ as $SOT1$. A new SOT tree will be created with $m1$ being the root and $SOT1$ being its children. 3) $m1$ is on the right side of $e1$. In this case, we should add the distinct child matches of $e1$ and all its descendant elements $e’$ into $resultSOT$. Finally, since both lists are ordered, we need to scan them only once. Fig. 17 depicts the details of this algorithm. $tree(m, subSOT)$ in line 14 is to create a new tree with $m$ being the root and all the trees in $subSOT$ being its children. The time complexity is $O(n)$, where $n$ is the size of output.

Fig. 16. Intuition of computing distinct child matches under PC axis.

Fig. 17. Pseudocode for computing distinct child matches.

4.5 Put Together: Processing GTP Queries

In this section, we present a solution for processing GTP queries by exploiting the following two heuristics. The first one is the common predicate pushdown before any joins. The second one is that we conduct the path join at each branch node in a top-down manner. This heuristics is based on the fact that the join elimination of an element at higher level could possibly eliminate many path matches (Sections 4.2 and 4.3).

Fig. 18 depicts an example how the sample GTP query is processed. The GTP query is first decomposed into a set of paths (Section 4.1), which are then added into the shared path matching engine (Section 3). The postprocessing starts with the predicate evaluation on $D$. For this, as mentioned earlier, for each $A$ element in $SOT(S1)$, we obtain the path matches for “/A/B/D” and evaluate the predicate using the algorithm in Fig. 14. We denote $SOT(S1)$ as the resulting $A$ elements that have matches for both “/A/B/C” and “/A/B/D[σ].”

Next, the path joins are evaluated at each branch node. First, $SOT(S1) \times SOT(S1)$ is processed for branch node $A$. Note that $A$ is a for-binding node, i.e., each matching $A$ element will lead to a distinct result. Hence, for each join result $a$, we obtain the corresponding matching $B$ elements for path “/A/B/C” and “/A/B/D[σ]”, i.e., $a\cdot getChildMatch(S2)$ and $a\cdot getChildMatch(S3)$. Two these SOTs are joined further for branch node $B$. Note that $B$ is not a binding node, taking each individual matching $B$ element into consideration will generate duplicates or out-of-order results [13]. Instead, we take the entire matching tree of $B$ as a whole and compute their distinct child matches of $C$ (Section 4.4).

Finally, after element $a$ in $SOT$ is found to be participated in the final result, we evaluate the optional axis by obtaining $a\cdot getChildMatch(S4)$ and creating a list for them. Since the elements to be grouped already stay together, an explicit grouping operator (outer-join as well) [8] can be avoided.

Fig. 19 depicts the GTP postquery processing algorithm after path matching and predicate evaluation. The main idea is that

1. when the current query node $E$ is a for-binding node, we consider each element in the SOT one by one;
2. when \( E \) is a group binding node, the \textit{joinResult} in line 2 contains all the elements that need to be grouped together;
3. when the current query node \( E \) is a not binding node, we instead consider each entire tree in the SOT and get the distinct matches over its child query node; and
4. the optional axes are evaluated using the same algorithm once the mandatory portion of the GTP query is completed (not shown in Fig. 19).

Now, we briefly review our solution for addressing the challenges listed in Section 1. The first benefit of our approach is that we avoid enumerating any full path matches, which is potentially an exponential cost [3]. The second benefit comes from the predicate evaluation as demonstrated in Section 4.2. The third benefit comes from the join processing. In particular, all the joins evaluated in our framework are duplicate free and merge join based. Furthermore, the scope of the join at any nonroot branch node is also limited with regard to their parent element, e.g., \( a \) getChildMatch\((S_2) \) and \( a \) getChildMatch\((S_3) \). Finally, our solution avoids the potentially expensive duplicate elimination, sort, outer-join, and grouping operations, and is thus suitable to process any arbitrary complex GTP queries.

**Handling Boolean queries.** We now briefly present the solution for XML filtering, when only the satisfied query identities are interested. The algorithm for handling Boolean queries, which simply returns true or false information, is quite similar to the GTP query processing algorithm in Fig. 19. The main difference is that we assume each query node is a for-binding node and terminate the algorithm as long as one match is found.

**Complexity results.** The input to \textit{GF}ilter for post-processing are TOP encodings, which have polynomial space complexity. Furthermore, one nice feature of \textit{GF}ilter is that the intermediate results during postprocessing are always encoded, which also avoids any exponential explosion throughout the processing. For a simple XPath query, the time complexity for postprocessing is \( O(|Q||D|) \). The reason is that the joins at each branch node are merge join based and involve only distinct document elements. The total join cost is thus bounded by \( O(|Q||D|) \). At the same time, the computation of distinct child document elements (\textit{getDistinctChildMatch}() function in Fig. 19) has a linear cost during the top-down join processing. For arbitrary GTP query, the time complexity is \( O(|Q||D| + |R|) \), where \( R \) is the size of the results of a subtwig query, which is a minimal subtwig in the original GTP query that contains all the binding nodes. This is due to the fact that the join can still be done in \( O(|Q||D|) \) and enumeration of the query results does not require any sorting or duplicate elimination. As a comparison, the GTP query processing based on full path matches [3], [7] inherently has an exponential time complexity due to the possibly exponential number of path matches.

---

**Fig. 18.** Example of processing GTP queries.

**Fig. 19.** GTP postquery processing algorithm.

---

[QueryResult GTPPostQueryProcessing\(SOT \land SOT(n), queryNode E) \land \exists O(\exists O(\exists O(n)))]

1. \( \text{BEGIN} \)
2. \( \text{begin} \)  \hspace{1cm} \( \text{result} = \text{MEncPaths}(\text{SOT}, E); \)
3. \( \text{If} \) result is empty
4. \( \text{return} \) empty;
5. \( \text{If} \) \( E \) is a for-binding node
6. \( \text{FOR} \) each \( \text{elem} \) in result
7. \( \text{withResult} = \text{true}; \)
8. \( \text{FOR} \) each query node \( M \) with binding nodes below & \( \text{allWithResult} \)
9. \( \text{partialResult} = \text{GTPPostProcessing}([\text{elem} \land \text{getChildMatch}(S_n)], M); \)
10. \( \text{If} \) partialResult is empty
11. \( \text{withResult} = \text{false}; \)
12. \( \text{If} \) allWithResult
13. \( \text{output} = \text{output UNION} \text{getDistinctChildMatch}(); \)
14. \( \text{Else} \) if \( E \) is a group-binding node
15. \( \text{output} = \text{createList}(); \)
16. \( \text{Else} \) if \( E \) is not binding node, assume \( M \) is \( E \)’s child with binding node below
17. \( \text{FOR} \) each \( \text{tree} T \) in result
18. \( \text{output} = \text{output UNION} \text{GTPPostProcessing}(T \land \text{getDistinctChildMatch}(T, E, M), M); \)
19. \( \text{return} \) output;
20. \( \text{END} \)
5 EXPERIMENTAL EVALUATIONS

We implemented our GFilter (GF) solution using Java 1.5 and performed experiments on a PC with a Pentium M-2.2-GHz processor and 2 Gbytes of main memory. The Java virtual machine memory size is set to 800M. We compare GFilter with the state-of-the-art XML filtering solution, YFilter (YF) [3], [8] and our earlier work AFilter (AF) [7]. We use the release version of YFilter, which is also implemented in Java. YFilter employs a top-down NFA for path matching, and evaluate predicates and path joins during postprocessing based on full path matches. AFilter also employs a bottom-up path matching approach. A major difference is that AFilter uses a more stringent triggering condition in terms of the leaf query step, while GFilter employs the leaf query node itself. Second, AFilter indirectly addresses the sharing of the common document prefix matching via prefix caching, while GFilter uses explicit sharing through postorder document traversal.

Data sets. Three different types of data sets are used in our experiments, namely, news industry text format (NITF) [3], Bib, and Book from XQuery user case in [16]. Note that Book has a recursive DTD. We use ToXGene [17] to generate the XML document. The default document size is 10 Kbytes. All the values generated are integers between 1 and 100. The default self-nesting of section element, i.e., recursion, for the Book data set is 3.

Queries. We use the query generator in YFilter test suite [3]. We vary the parameters such as query depth, probability of "//" and wildcard "*", number of values predicates, and number of branches. The default setting is depth = 6, probability of "//" = 0.1, probability of "*" = 0.2, number of value predicates = 0, number of branches = 0, and distinct = TRUE. All the value predicates are equality conditions over integers between 1 and 100. We vary one or two parameters in each of the experiments below. All the times reported are the filtering time excluding the XML document parsing time.

5.1 Filtering of Boolean Path Queries

First, we consider the filtering of path queries. A total of 50,000 path queries are generated for NITF data set. We vary the probability of // and * during query generation and investigate how that affects the filtering performance. Table 1 depicts the results.

As can be seen, both bottom-up approaches, namely, AFilter and GFilter, significantly outperform the top-down approach YFilter. This is due to the fact that the bottom-up solution can benefit from the generally more stringent data selectivities at the query leaves, a phenomenon that has already been observed in our prior work AFilter [7].

Between the two bottom-up approaches, AFilter performs faster than GFilter over this data set. The reason is that the path selectivity is very high for NITF data set. As expected, since AFilter employs a more stringent triggering mechanism, i.e., leaf query step rather than leaf query node, less computation is required.

Next, we consider Book, which has a recursive DTD. We are only able to generate 2,500 distinct queries due to its extremely small DTD. In this experiment, we vary the maximum recursive depth of the XML document, i.e., the self-nesting of section element, from 2 to 4. Table 2 depicts the results.

As can be seen in Table 2, the cost of YFilter increases significantly when the maximum recursion increases. This is due to the excessive (theoretically exponential) number of prefix matches for recursive data set. In comparison, the bottom-up approaches perform much better as fewer document paths need to be considered. For this data set, AFilter becomes slower than GFilter. The reason is that since the path selectivity is low, the leaf query step trigger condition is no longer effective. Although shared prefix matching are saved in a prefix cache to avoid repetitive computation, the corresponding caching cost is still nontrivial.

Next, we consider Bib, which is a nonrecursive data set and has a very small DTD. On this data, only 486 distinct path queries can be generated. The results depicted in Table 3 show trends similar to those observed earlier.

We also experimented with different query workloads by varying the skewness and by increasing the number of queries for NITF data set. The results are similar to the ones presented here. Finally, GFilter can be extended with AFilter’s feature to selectively evaluate the document paths. Due to limited space, we omit the details here.

5.2 Postprocessing for Boolean Queries

In this section, we focus on postprocessing for handling Boolean XML queries. This includes predicate evaluation and path joins. Since AFilter only focuses on path filtering, we will not include AFilter for comparison in the rest of the experiments. Also, due to the very high path selectivity of NITF data set, the postprocessing cost between YFilter and GFilter is close since there are only a few path matches to be handled. The benefit of GFilter primarily comes from the path matching part. Thus, we will not show the results for NITF data set here.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Varying Probability of // and * (NITF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P, R)</td>
<td>YF</td>
</tr>
<tr>
<td>0.2, 0.1</td>
<td>70ms</td>
</tr>
<tr>
<td>0.3, 0.1</td>
<td>75ms</td>
</tr>
<tr>
<td>0.4, 0.1</td>
<td>80ms</td>
</tr>
<tr>
<td>0.2, 0.2</td>
<td>70ms</td>
</tr>
<tr>
<td>0.2, 0.3</td>
<td>75ms</td>
</tr>
<tr>
<td>0.2, 0.4</td>
<td>75ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Varying Recursion Depth (Book)</th>
</tr>
</thead>
<tbody>
<tr>
<td>r = 2</td>
<td>YF</td>
</tr>
<tr>
<td></td>
<td>70ms</td>
</tr>
<tr>
<td>r = 3</td>
<td>110ms</td>
</tr>
<tr>
<td>r = 4</td>
<td>150ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3</th>
<th>Path Queries over Bib</th>
</tr>
</thead>
<tbody>
<tr>
<td>YF</td>
<td>AF</td>
</tr>
<tr>
<td>30ms</td>
<td>20ms</td>
</tr>
</tbody>
</table>

Authorized licensed use limited to: Arizona State University. Downloaded on February 17,2010 at 00:13:25 EST from IEEE Xplore. Restrictions apply.
5.2.1 Predicate Evaluation

We first report the results of predicate evaluations over path queries by varying the number of predicates per query. In particular, when there is only one predicate, we distinguish between two cases. In the first case, all predicates are at the leaf query nodes for all queries. In the other case, the predicates are randomly generated over all query nodes. The number of queries is 20,000.

For the Bib data set, we also vary the average number of authors per publication from 1 to 3 (without changing the document size). Note that the latter is more common in practice. Table 4 depicts the results. Here, “1L” means one predicate at the leaf query node and “P:number” is the number of predicates evaluated. “A = 1 or A = 3” means one or three authors per publications. As can be seen, first, when all the predicates are at the query leaf nodes, both approaches evaluate the same number of predicates. When the predicates are randomly placed at the query nodes, GFilter reduces the number of predicate evaluations. When there are three authors per publication, GFilter saves even more predicate evaluations. Take the query /book[@year = 2005]/author for example. When there are multiple authors per book, GFilter will save redundant predicate evaluations over the same book. In comparison, YFilter does not have such capabilities.

For the Book data set, we also vary the maximum recursion of the XML document from 2 to 4 (without changing the document size). Table 5 depicts the results, where “R = 2 or R = 4” means the maximum recursion, i.e., the maximum nesting of section element. As can be seen, when the predicates are all at the leaf query node, the number of predicates evaluated is similar for both YFilter and GFilter. When the value predicates are randomly generated, i.e., some are at nonleaf nodes, GFilter reduces the number of predicate evaluations. Take the query /book//section//figure[@width = 2]/title for example. If the current section element does not have a match to the query, we can ignore all its descendant section elements (see Section 4.2). Hence, the deeper the recursion, the more savings. For this data set, GFilter outperforms YFilter by a wider margin than that for the Bib data set. The reason is that YFilter requires to enumerate the full path matches for evaluating predicates. When the path query is recursive with regard to the document schema, there are potentially exponential number of path matches, resulting in expensive enumeration cost. In comparison, GFilter always guarantees polynomial time and space complexity.

5.2.2 Twig Queries

We now report the results for filtering twig queries. We use the simple nested-loop join for YFilter since the join stops as soon as one match is found when handling Boolean queries. First, a total of 20,000 twig queries are generated with maximum three branches and two predicates per query for both Book and Bib data sets. Table 6 depicts the results, where “J:number” is the number of join probes. As can be seen, when there are more authors per publication (Bib data set) or deeper recursion (Book data set), GFilter evaluates much less number of predicates and join probes than YFilter.

5.3 Processing GTP Queries

In this section, we evaluate the performance of GFilter for processing GTP queries. In particular, we vary the complexity of the GTP queries by varying the number of for-binding and group-binding nodes. Since current YFilter implementation does not support arbitrary GTP queries, we choose YFilter’s XPath performance as the baseline for comparison. We assume that the DTD is available to YFilter; thus, it uses merge joins for Bib data set and hash joins for the Book data set which has a recursive schema. For GFilter, the merge join-based encoded path join algorithm is always applied for any data set.

First, a total of 20,000 twig queries with maximum three branches and two predicates are generated for the Bib data set. We randomly convert the query nodes into for-binding nodes or group-binding nodes. The XML documents contain three authors per publication. Fig. 20 depicts the results. Here, y-axis denotes the execution time and x-axis denotes the complexity of the GTP queries. “F = 1, G = 0” is the XPath case. “F = 2, G = 3” means there are two for-binding nodes and three group-binding nodes per query (on average, there are six nodes per query). These are fairly complex GTP queries. Not only they require more computations but the result size also becomes significantly larger. As can be seen in the figure, the query processing cost of GFilter increases just 20 percent from the XPath case to the

<table>
<thead>
<tr>
<th>Table 4</th>
<th>Predicate Evaluation on Bib Data Set</th>
</tr>
</thead>
<tbody>
<tr>
<td># Predicates</td>
<td># Authors</td>
</tr>
<tr>
<td>1L</td>
<td>A=1</td>
</tr>
<tr>
<td></td>
<td>A=3</td>
</tr>
<tr>
<td>1</td>
<td>A=1</td>
</tr>
<tr>
<td></td>
<td>A=3</td>
</tr>
<tr>
<td>2</td>
<td>A=1</td>
</tr>
<tr>
<td></td>
<td>A=3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5</th>
<th>Predicate Evaluation on Book Data Set</th>
</tr>
</thead>
<tbody>
<tr>
<td># Predicates</td>
<td>Recursion</td>
</tr>
<tr>
<td>1L</td>
<td>R=2</td>
</tr>
<tr>
<td></td>
<td>R=4</td>
</tr>
<tr>
<td>1</td>
<td>R=2</td>
</tr>
<tr>
<td></td>
<td>R=4</td>
</tr>
<tr>
<td>2</td>
<td>R=2</td>
</tr>
<tr>
<td></td>
<td>R=4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 6</th>
<th>Twig Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>Property</td>
</tr>
<tr>
<td>Bib</td>
<td>A=1</td>
</tr>
<tr>
<td></td>
<td>A=3</td>
</tr>
<tr>
<td>Book</td>
<td>R=2</td>
</tr>
<tr>
<td></td>
<td>R=4</td>
</tr>
</tbody>
</table>
more complex “F = 2, G = 3” case. In fact, even in the most complex case, it still performs better than \(Y\)Filter’s XPath case. Note that in order for the full-path-based solution, such as for \(Y\)Filter, to support the complex GTP queries, the cheap semijoins used for XPath queries are no longer sufficient. More costly operations, such as inner, outer-joins, and/or grouping have to be introduced.

In the above experiments, the query selectivity in terms of the number of queries matched is 20 percent (same for the rest of the experiments in this section), which is fairly high in XML filtering scenarios [3], [8]. Hence, \(G\)Filter scales quite well with increasing complexity of the GTP queries even when there are a large number of data to be transformed.

Next, a total of 20,000 twig queries with maximum three branches and two predicates are generated for the Book data set. Similar to the previous experiment, we randomly convert the query nodes into for-binding nodes or group-binding nodes. The maximum nesting of section elements in the XML document is 4. In this experiment, we further vary the probability of “//,” i.e., descendant axis (DS) from 0.2 to 0.4. Having more “//” in the queries means that there will be more recursive queries with regard to the XML schema. This poses a big challenge for full-path-based approach in elimination of duplicates [8]. This is, however, no longer a challenge for \(G\)Filter. Fig. 21 depicts the results. As can be seen from the figure, the query processing cost of \(G\)Filter increases just 20 percent from the simpler XPath case to the more complex “F = 2, G = 3” case. Varying the probability “//” increases \(Y\)Filter’s cost a lot even under the XPath case. Processing more complex “F = 2, G = 3” GTP queries consistently adds about 20 percent cost to that for the \(G\)Filter XPath case.

For the Book data set, the experiment setting is similar to Fig. 21 except we fix the probability of “//” to 0.2 and vary the number of queries. The results are depicted in Fig. 23. As can be seen from the figure, similar results as in Fig. 21 are observed. That is, \(G\)Filter outperforms \(Y\)Filter more than 50 percent when processing the XPath queries. Processing more complex “F = 2, G = 3” GTP queries consistently adds about 20 percent cost to that for the \(G\)Filter XPath case.

Memory usage. Lastly, the memory used to store the TOP encodings is 1.78 Mbytes for processing 200,000 twig queries over 10-Kbytes Book XML document. It increases to merely 1.83 Mbytes (maximum memory usage in all experiments) when processing 300,000 twig queries. Hence, it is not an issue to hold the TOP encodings in memory in the filtering scenario, where the XML document is small (just several kilobytes) [3]. This result also indicates that path-sharing scheme scales quite well in terms of the increasing number of queries.

In summary, \(G\)Filter saves significant amount of redundant predicates and join evaluation during postprocessing. The gain enlarges when the XML document has deep recursions and/or high repetitions. \(G\)Filter also scales well under the increasing complexity of GTP queries since it has a polynomial time and space complexity and processing the XPath queries. Processing the more complex “F = 2, G = 3” GTP queries consistently adds about 20 percent more cost to that for the \(G\)Filter XPath case.

In summary, \(G\)Filter saves significant amount of redundant predicates and join evaluation during postprocessing. The gain enlarges when the XML document has deep recursions and/or high repetitions. \(G\)Filter also scales well under the increasing complexity of GTP queries since it has a polynomial time and space complexity and processing the XPath queries. Processing the more complex “F = 2, G = 3” GTP queries consistently adds about 20 percent more cost to that for the \(G\)Filter XPath case.

For the Book data set, the experiment setting is similar to Fig. 21 except we fix the probability of “//” to 0.2 and vary the number of queries. The results are depicted in Fig. 23. As can be seen from the figure, similar results as in Fig. 21 are observed. That is, \(G\)Filter outperforms \(Y\)Filter more than 50 percent when processing the XPath queries. Processing more complex “F = 2, G = 3” GTP queries consistently adds about 20 percent cost to that for the \(G\)Filter XPath case.

Memory usage. Lastly, the memory used to store the TOP encodings is 1.78 Mbytes for processing 200,000 twig queries over 10-Kbytes Book XML document. It increases to merely 1.83 Mbytes (maximum memory usage in all experiments) when processing 300,000 twig queries. Hence, it is not an issue to hold the TOP encodings in memory in the filtering scenario, where the XML document is small (just several kilobytes) [3]. This result also indicates that path-sharing scheme scales quite well in terms of the increasing number of queries.
avoids any expensive operations such as duplicate elimination, sort, outer-join, and grouping.

6 RELATED WORK

Most existing works on XML filtering employ an FSA-based approach [1], [3], [6], [18]. An XML path expression can be translated into an NFA [3], [19]. Multiple path expressions are further shared by their common prefix [3], [8]. The benefit of taking an NFA-based approach is that it is very easy to insert and delete queries, which is important for the scenario of large-scale XML filtering. There is also some work to improve the filtering throughput by translating the NFA into DFA lazily [6]. While the number of runtime active states/transition decreases, the static automata states can be prohibitively large and expensive to build. One main restriction of the DFA-based approach is that it is hard to insert and delete queries and its query expressiveness is limited.

There are also a number of nonautomaton XML filtering approaches. In [11], the authors propose an index-based solution for filtering path queries, which is more efficient for handling large XML documents. Their earlier work [1] mainly focuses on optimizing the path matching performance via a bottom-up approach. It does not consider how to further improve the postprocessing performance. MatchMaker [12] proposes an index structure for matching multiple tree patterns, which requires two passes of the XML document. F1st [20] proposes to represent the twig query as sequence and translate the twig filtering as a sequence matching problem. Its query expressiveness, however, is limited to handle Boolean ordered twig queries. In [21], the authors propose a XML filtering solution using a relational database system in order to support huge number of subscriptions. To our knowledge, YFilter [3] and its extension in [8] are the only works that consider the optimization of filtering as well as query processing for both path and twig queries. Their main technique is to exploit the sharing of path, predicate, and join evaluations among multiple queries [3], [8]. In this work, we demonstrate that such path-sharing scheme essentially provides a unique optimization opportunity by avoiding redundant postprocessing, which cannot be resolved through sharing. The query language we support, i.e., GTP query, is also more powerful than existing solutions.

There are also a number of XML stream engines [4], [22], which mainly focus on processing a single XML query. XML query processing has also been extensively studied for stored XML documents. In particular, holistic approaches have shown to be effective by minimizing the intermediate results. Most existing works use stacks for result encodings [15], [4], [7], [13]. While similar in principle to the hierarchical stack encoding [13], our current TOP encoding scheme is more succinct and avoids the cost of creating any stacks. Stack-based encoding has also been exploited in XML filtering scenario. For example, a single stack is used in YFilter [3]. Multiple stacks, one per query node and one per node label, are used in [11] and [7], respectively. Note that these stack-based encodings are created top-down, while our TOP encodings are constructed bottom-up. The top-down approach, however, is not helpful to address the problem of redundant predicate and join evaluations. The main reason is that when the leaf element is reached, it has to traverse back to find the path matches. Some elements that correspond to the higher level query nodes have to be repetitively visited. It is, however, unclear how our TOP encodings can be cheaply obtained through the top-down path evaluation. Also, none of these holistic algorithms is designed to handle a large number of GTP queries for XML filtering.

Lastly, the path matches that are encoded in a TOP form is similar to nested relations. The main difference is that, first, our encoded path matches may not be well nested, e.g., b1 and b2 refer to the same descendant C elements in Fig. 11. Second, we further capture the hierarchy (AD relationship) within the same nesting level such as b1 and b2. Note that our selection and join operations defined in Section 4 are similar to the recursive algebra over nested relations in [23] in terms of its recursive definition. It is an interesting future work to build a formal query algebra based on TOP encodings. One property of our TOP encoding is that the elements that are unnested from the encodings are guaranteed be part of the path matches.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced a bottom-up NFA-based path matching algorithm. Then, a novel TOP encoding scheme is designed to compactly represent the path matches for the entire XML document. We further show how to piggyback this encoding scheme into the path matching algorithm and how to efficiently evaluate predicates, joins, and GTP queries based on encoded path matches. Similar to an NFA-based solution, GFilter can easily allow queries to be inserted or deleted. Future work includes to share the postprocessing among multiple queries.

REFERENCES


Hua-Gang Li received the BE degree in computer science from Nanjing University of Aeronautics and Astronautics, Nanjing, China, the MS degree from National University of Singapore, and the degree from the University of California at Santa Barbara. He is currently affiliated with Oracle Corporation. His research interests include the area of data warehousing, XML databases, and streaming databases.

Jun’ichi Tatemura received the BE degree in electrical engineering and the ME and PhD degrees in information engineering from the University of Tokyo. He was an assistant professor in the Institute of Industrial Science, University of Tokyo from 1994 to 2001. He is currently a senior research staff member leading the data management group in NEC Laboratories America, Inc., Cupertino, California. His current research interests include data-centric middleware (such as XML and data stream), data integration, and user interaction in data management.

Divyakant Agrawal is a professor of computer science at the University of California (UC) at Santa Barbara. Since January 1997, he has been a visiting senior research scientist at NEC Research Laboratories of America at San Jose, California, and Cupertino, California. His research interests include the areas of dynamic web caching, web services, XML data processing, real-time data filtering, as well as data querying and information retrieval.

K. Selcuk Candan received the BS degree in computer science from Bilkent University, Ankara, Turkey, in 1993 and the PhD degree from the University of Maryland, College Park. He has been an associate professor in the Department of Computer Science and Engineering, Arizona State University, since August 1997. He has worked extensively on the integration and presentation of heterogeneous and distributed information and information sources. His research projects are funded by the US National Science Foundation, the US Department of Defense, and the US Department of Education (Rehabilitation Services Administration). He has published more than 100 articles in respected journals and conferences in related areas. He served in the organization and program committees of various international conferences. Most recently, he is an editorial board member of the Very Large Databases (VLDB) Journal and the ACM SIGMOD Digital Symposium Collection (DSC). In 2006, he served as an organization committee member for SIGMOD ’06. This year, he is serving as a PC chair for another leading, flagship conference of the ACM, this time focusing on multimedia research (SIGMM ’08). He has been collaborating with NEC America Research Labs since 1996.