An Efficient Bottom-up Filtering of XML Messages by Exploiting the Postfix Commonality of XPath Queries

Jae-hoon KIM, Member, Youngsoo KIM, and Seog PARK, Nonmembers

SUMMARY Recently, for more efficient filtering of XML data, YFilter system has been suggested to exploit the prefix commonalities that exist among path expressions. Sharing the prefix commonality gives the benefit of improving filtering performance through the tremendous reduction in filtering machine size. However, exploiting the postfix commonality can also be useful for an XML filtering situation. For example, when a stream of XML messages does not have any defined schema, or users cannot remember the defined schema exactly, users often use the partial matching path queries which begins with the descendant axis (/), e.g., /science/article/title, /entertainment/article/title, and /title. If so, the registered XPath queries are most likely to have the postfix commonality. Therefore, in this paper, we introduce a bottom-up filtering approach exploiting the postfix commonality against the top-down approach of YFilter.

EXPERIMENTAL RESULTS Our method has better filtering performance when registered XPath queries mainly consist of the partial matching path queries with the postfix commonality.

key words: XML data filtering, bottom-up approach, postfix sharing, non-deterministic finite automaton

1. Introduction

In the past few years, XML data filtering systems have been studied to support the publish/subscribe scheme [1]: XFilter [2], YFilter [3], XTrie [4], AFilter [5], etc. In such systems, users subscribe their profile represented as an XPath query [6], and XML documents from various data sources are continuously inputted to the system. If there is a document which matches a user profile, the document is delivered to the user. Among the studies mentioned above, YFilter in particular has been proposed to exploit the prefix commonality that exists among path expressions. For example, for the subscribed XPath queries /news/science/article/title, /news/science/article/summary, and /news/science/article, the partial path expression /news/science/article is shared. YFilter combines all XPath queries into a single Nondeterministic Finite Automaton (NFA) to exploit such prefix commonality. This approach brings about a tremendous reduction in filtering machine size, and as a result, it gives the benefits of higher filtering performance and scalability.

However, we assume a situation where the postfix sharing is useful. For example, when a stream of XML messages does not have any defined schema, or users cannot remember the defined schema exactly, users often use the partial matching path queries which begins with the descendant axis (/), e.g., /science//title, /entertainment/title, and /title. If so, the registered XPath queries are most likely to have the postfix commonality. Thus, in this paper, we introduce a novel bottom-up filtering approach exploiting the common postfixes in the NFA-based scheme, as opposed to the top-down approach of YFilter. We name the new method PosFilter. Since our method matches the shared postfixes once with an inputted XML data stream and omits the matching evaluation for the initial descendant axis of the partial matching path query, PosFilter has better filtering performance than YFilter. The key idea of our method is that in order to exploit the postfix commonality, the NFA filtering machine is constructed according to the reverse path expression of registered XPath queries, and the state transition in the NFA execution is conducted at the end-of-element event. However, the execution of YFilter is conducted at the start-of-element event.

Similar to our purpose, AFilter [5] has been recently proposed for postfix sharing. Moreover, the approach was intended to benefit from prefix commonalities, while simultaneously leveraging postfix commonalities. However, we think that they omitted a significant explanation for the adaptable filtering. First, they did not show how to define the detailed condition for the unfolding (i.e., the switch from postfix sharing to prefix sharing). Next, they did not show how much is the overhead of calculating the condition, which is able to significantly affect the overall filtering performance. In addition, AFilter is not an automata-based scheme. It uses its own specific memory organization and a path matching algorithm. Hence, although the scheme of AFilter can give further improvement of path matching speed, we think that the substantial benefits of expressiveness and incremental maintenance provided by the NFA model outweigh the speed improvement as mentioned in YFilter [3]. We can conveniently set up and maintain a filtering system using some development tools and libraries supporting automata theory.

This paper is organized as follows. In Sect. 2, we present the basic idea of this paper. Section 3 describes the proposed PosFilter technique and in Sect. 4, we address...
some limitations of PosFilter. Section 5 presents some experimental results for the throughput comparison of PosFilter, YFilter, and AFilter, and in Sect. 6, we review other studies related to our method. Section 7 finally concludes this paper.

2. Motivation

When a stream of XML messages does not have any defined schema, or users cannot remember the defined schema exactly as in two cases shown later, users often compose partial matching path queries remembering some familiar vocabularies. Here, the partial matching path query is a query type which begins with the descendant axis ("//") [11]. For example, the query '/article/title' against the sample data of Fig. 1(a) has the same query result as the following full path queries which begin with the root element ('/'): '/news/science/article/title', '/news/entertainment/article/title', and '/news/economy/article/title'.

Case 1: Let us consider a situation having to filter multiple heterogeneous, but related with a similar subject, XML data. Since in such a situation, the data stream are most likely to have similar tag names and structure as in Fig. 1(a)(b), users prefer subscribing a partial matching path query consisting of vocabularies of the common concept, e.g., '//science//summary', '//entertainment//summary', and '//summary'. Here, for the semantic interoperability of the query '//science//summary' against the XML document of Fig. 1(b), we assume that a query translation technique can be used as in OBSERVER [12] and the study [13]. That is, the statement of a subscribed query can be translated into another statement using the vocabularies of Fig. 1(b), e.g., 'science' ⇒ 'science_information' and 'summary' ⇒ 'abstract'. However, since the investigation is beyond the scope of this paper, we will not deal with the query translation problem.

Case 2: Suppose that a user is familiar with the tag names and structure of Fig. 1(a) but the user does not remember the schema completely. If so, the user should compose a partial matching path query only by remembering the previously familiar tag names and structure.

If the partial matching path queries occur often, there is a high possibility that the subscribed XPath queries have more postfix commonalities than the prefix commonalities. For example, the sample queries of Fig.1(c) share more postfixes ('title', 'article/title', and 'summary') than prefixes ('science', 'entertainment', and 'article').

Figure 2 shows the NFA of YFilter [3] exploiting the prefix commonality, which corresponds to our postfix sharing scheme of Fig. 5(a). First, we can see that the machine size of our PosFilter is smaller than that of YFilter due to the postfix sharing. Consequently, it can be expected that the number of automata states in a runtime stack should be smaller in our PosFilter method, and the throughput should increase. For example, in the NFA of YFilter, the reachable states by the <title> element are six (state 2, 5, 7, 8, 10, and 16), but in the NFA of PosFilter, only one (state 2). Next, since the NFA of PosFilter is constructed according to the reverse path expression of the registered XPath queries, the self-loop transition (state 2 in the NFA of YFilter) represent-
Fig. 2 YFilter exploiting the prefix commonality.

ing the initial descendant axis of the partial matching path query can be removed in the NFA of PosFilter. This also brings about the benefit of reducing the state explosion in the runtime stack by the initial descendant axis. In the next section, we will introduce this bottom-up filtering approach based on postfix sharing in detail.

3. PosFilter: An NFA-Based Approach Exploiting the Postfix Commonality

3.1 XPath Queries

Prior to introducing the PosFilter construction, let us consider representing each XPath query as an NFA. We focus especially on a class of XPath [14], denoted by XP(/, //, *) and defined as follows:

\[ p := E | l | * | p/p | //p, \]

where \( E \), \( l \), and \( * \) denote an empty path, a label, and a wildcard respectively and "/" and "//" stand for a parent-child axis and a descendant axis. As for the filtering of a nested path expression, we adopt the query decomposition technique suggested in YFilter [3]. For example, the query "/a[b]/c[d/e]/f" can be decomposed into a main path "/a/c/f" and two predicate paths "/a/b" and "/a/c/d/e", after which each path is individually registered into the filtering system. If the three paths are all matched with an inputted document, then the document is filtered.

Figure 3 shows the NFAs for the four location steps which can appear in XP(/, //, *). The location steps "/l" and "/*/" are represented as the state transitions by a label \( l \) and a wildcard respectively. The wildcard operator matches any one label. The location steps "///" and "///*" are represented as the \( \epsilon \)-transition and the state with a self-loop. The symbol "\( \epsilon \)" (or called empty string) is used to mark a transition that requires no input, and it has the following properties: \( u \epsilon = u = \epsilon u \) and \( u \epsilon v = uv \) for arbitrary XML elements \( u \) and \( v \). In fact, the \( \epsilon \)-transition is necessary for combining the two location steps "/l" and "///*". For example, let us consider the single NFA combining the two NFAs of Q5 and Q6 in Fig. 4.

We can see that in the combined NFA, Q6 is incorrectly represented as "/article//summary". However, in Fig. 6 (a), Q6 is safely combined by the \( \epsilon \)-transition.

Note that the automata of Figs. 3, 4 and 6 are represented bottom-up so that the common postfixes of subscribed queries can be shared.

3.2 Incremental Construction of a Combined NFA

We now describe the incremental construction of our PosFilter. Let \( NFAp \) denote the non-deterministic finite automaton for a path expression, and let \( combined-NFAp \) denote a single NFA which all \( NFAps \) are combined into.

(Step 1) Create a single initial state (state 1) shared by all \( NFAps \).

(Step 2) If an XPath query is first inputted, create the initial \( combined-NFAp \) starting from state 1.

For the illustration, let us see the example of Fig. 5. When the XPath query Q1 of Fig. 1 (c) is subscribed first, the initial \( combined-NFAp \) is constructed like Fig. 5 (a).
(Step 3) Whenever a new XPath query is subscribed, add the NFAp for the query into the combined-NFAp.

- Match up the NFAp to the combined-NFAp starting from each initial state.

- If the final state of the NFAp is reached or a state is reached for which there is no transition matched, make the final state an accepting state in the combined-NFAp, add the query ID to the matching query set associated with the accepting state, or create a new branch from the last state reached in the combined-NFAp.

For example, when the query Q2 is subscribed, a new transition (wildcard operator) from the last matched state (state 2) is added like Fig. 5 (b). When the queries Q3 and Q4 are subscribed, the final states (state 2 and 3) are marked as an accepting state (denoted by two concentric circles) like Fig. 5 (c). For the duplicate query Q9, the query ID Q9 will only be added into the matching query set {Q1} like Fig. 5 (d).

When a path expression begins with the descendant axis ("//"), the last ε-transition and the last state with a self-loop can be omitted. The reason is that the last evaluation of the descendant axis does not affect the query result since the path matching is evaluated bottom-up in PosFilter. For example, let us see the automaton of Fig. 5 (d) representing the query Q3 of Fig. 1 (c) and suppose that the streaming XML data of Fig. 1 (a) are read in reverse order ("...</news>...<summary>...<title><article><science><news>"). When it reaches the accepting state of state 2, the elements <article>, <science>, <news>, do not need to be checked any more for the query Q3. This benefit is important in that it can reduce the state explosion by the initial descendant axis in the YFilter machine of Fig. 2.

3.3 Executing the PosFilter

PosFilter is executed in an event-driven fashion using SAX parser. When an arriving XML document is parsed, the events raised by the parser callback the event handlers and drive the transitions in the combined-NFAp. We define the following three events: startDocument, startElement, and endElement.

startDocument. This event is raised when an XML document is arrived. At this time, the execution of the combined-NFAp begins at the initial state. That is, the state_id of the common initial state (state 1) is pushed to the runtime stack as the active state.

For the illustration, let us see the example of Fig. 6. Figure 6 (b) shows the stack state change in the combined-NFAp of Fig. 6 (a) when the XML document of Fig. 1 (a) is inputted. First, the initial state of the runtime stack is set to state 1 at the startDocument event.

startElement. This event is raised when a new element name is read from the document. At this time, the state_id 1 is pushed onto the top of the runtime stack as the active state.

endElement. This event is raised when an end-of-element name is read from the document. PosFilter uses the end-of-element for the state transition from currently active states. The following NFA execution is performed at this time.

(1) The state_ids of all active states at the top of the runtime stack are popped off, and the state_ids of the target states of all matching transitions from the active states are added to the top of the runtime stack as new active states. At this time, if any state_id at the top of the stack corresponds to state 1 or a final state in the combined-NFAp, it is removed.

(2) Especially, for the state transition with a self-loop (this represents the descendant axis ("//")), its own state_id is always added to the top of the stack.

For example, when the elements <news>, <science>, <article>, and <title> are inputted, the state_id 1 is pushed onto the top of the stack four times.

endElement. This event is raised when an end-of-element name is read from the document. PosFilter uses the end-of-element for the state transition from currently active states. The following NFA execution is performed at this time.

(1) The state_ids of all active states at the top of the runtime stack are popped off, and the state_ids of the target states of all matching transitions from the active states are added to the top of the runtime stack as new active states. At this time, if any state_id at the top of the stack corresponds to state 1 or a final state in the combined-NFAp, it is removed.

(2) Especially, for the state transition with a self-loop (this represents the descendant axis ("//")), its own state_id is always added to the top of the stack.
state.

The processing for the remaining elements follows the same procedure. When the start-of-element <summary> is inputted, the state_id 1 is pushed onto the top of the stack. Then when the end-of-element </summary> is inputted, the state_id 1 is popped off, and the state_id 7 is added to the top of the stack, i.e., 2, 7, because the state transition of 'summary' from state 1 exists. At this time, the query result of Q7 is returned. When the end-of-element <article> is inputted, it is checked whether or not the state transition of 'article' from state 2 or 7 exists. Since the transitions to the target states 3, 5, 8, and 10 are matched, the state ids are added to the top of the stack. The query Q4 and Q6 are matched. For the end-of-element of </science>, the state_id 4, 6, and 8 are added to the top of the stack and the query Q1, Q2 and Q9 are matched. In the stack, the state_id 4 and 6 will be removed because the states are a final state in the combined-NFAp. The state_id 8 is added again according to the above rule (2).

4. Limitation of PosFilter

It is important to understand that our PosFilter execution follows the state transition at the endElement event to share the common postfixes of subscribed XPath queries. This bottom-up filtering approach restricts the streaming XML document to being bounded and not huge size because surely, the endless stream of XML data cannot be buffered until an end-of-element is reached. Although this restriction makes PosFilter inapplicable to an environment having an unbounded XML data stream, we believe that our method is reasonable. This is because there are many application domains with the bounded XML data streams such as online news, online auction, online stock sites, etc. In addition, although the unbounded XML data stream cannot be buffered wholly, we can consider a partial buffering of a repeated XML subtree fragment. This is because in the most of unbounded XML data streams, the data pattern is a repetition of a specific bounded XML data fragment. For example, Figure 7 shows an unbounded XML data stream in a Location-Based Services (LBS) system where the bounded XML data fragment below the <sensor> element is repeated. Supposing all of the registered XPath queries filter only the data below the <sensor> element, the partially buffered data can be removed continuously whenever the end-of-element </sensor> is read.

PosFilter system is disadvantageous for prefixshared queries. As an adaptable filtering mechanism, we consider the duplicate system operation of PosFilter and YFilter. The following evaluation is performed at a regular interval (e.g., an hour, a day, etc.). When the current subscribed queries are mainly prefix-shared, YFilter system is launched, whereas PosFilter system is launched for the current postfix-shared queries. Since PosFilter is advantageous for the partial matching path query with postfix sharing as mentioned in Sect. 2, we use the following switching criteria for both systems:

\[
\text{if} \left( \frac{\text{#(partial matching path queries)}}{\text{#(total queries)}} > \alpha \wedge \frac{\text{sizeOf(PosFilter NFA)}}{\text{sizeOf(YFilter NFA)}} < \beta \right) \rightarrow \text{XML filtering by PosFilter}
\]

\[
\text{else} \rightarrow \text{XML filtering by YFilter}
\]

where #() is the number of queries, sizeOf() is the size of an NFA, \(\alpha\) and \(\beta\) is the real number between 0 and 1. Here, since the frequent postfix sharing makes the size of the NFA of PosFilter less than that of YFilter, \(\beta\) should be smaller than one. Especially, we observed that the value of \(\alpha\) can be lowered to even 0.2 through the experiments which will be introduced in the next section.

5. Performance Evaluation

In this section, we will present the results of some experiments to analyze the performance of the proposed PosFilter system. We compared PosFilter with YFilter and partially, with AFilter with respect to the postfix sharing.

5.1 Experimental Setup

We implemented PosFilter in Java (JDK 1.4). For the comparison of YFilter, we used the implementation provided at...
the Web site [15], and we could measure the filtering time of both methods. All experiments were performed on a Windows XP computer with 512 MB of memory and 1.7 GHz Pentium IV CPU.

For the test data, we used the auction data generated by the xmlgen of XMark [16] and for the test queries, we used the queries generated by the query generator of YFilter [15]. The test data size is approximately 100 MB. Table 1 shows the workload parameters for query and data generation. In order to get the partial matching path queries, we first generate a set of XPath queries by configuring the DS parameter in the query generator of YFilter. Then we can select a subset of partial matching path queries according to the parameter $P$.

5.2 Experiment 1: Comparison with YFilter

First, to see whether or not the partial matching path query with postfix sharing affects the filtering performance of PosFilter and YFilter, we performed the throughput comparison of both methods. The test query set consisted of only partial matching path queries ($P = 1$). Table 2 shows the throughput of both methods when the parameter $N$ is 100, 500, and 1,000. Here, the throughput is defined as the number of filtered elements per one second. We can see that PosFilter has a higher throughput than YFilter in all the cases. As indicated in Sect. 2, this result reflects that our PosFilter approach is advantageous to the partial matching path query with postfix sharing. The graph of Fig. 8 shows having traced the number of state transition in each NFA filtering machine for an arbitrary data section. The number is to count all the matching transitions from the current active states when an XML element is inputted. We can see that the number of state transition in PosFilter is lower than that in YFilter; the numbers in PosFilter are almost zero. This also reflects that our PosFilter machine has the less state explosion than the YFilter machine by exploiting the postfix sharing.

Next, to analyze how deep is the impact of the partial matching path query, we performed the throughput comparison of PosFilter and YFilter according to the occupying probability of partial matching path queries. In the experiment, the parameter $N$ is set to 100 and the parameter $P$ is varied from 0% to 100%. The graph of Fig. 9 shows that the throughput of YFilter worsens as the probability increases while PosFilter has the steady throughput. We can also see that even at 20%, YFilter has worse throughput than PosFilter.

5.3 Experiment 2: Comparison with AFilter

As mentioned in Sect. 1, AFilter is another study on exploiting the postfix sharing similar to our method. However, unlike state machine-based schemes such as PosFilter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$</td>
<td>5, 6</td>
<td>Depth of XML documents</td>
</tr>
<tr>
<td>$d$</td>
<td>1 to 5</td>
<td>Depth of XPath queries</td>
</tr>
<tr>
<td>$N$</td>
<td>100, 500, 1000</td>
<td>Number of queries</td>
</tr>
<tr>
<td>$P$</td>
<td>0 to 1</td>
<td>Probability of the partial matching path queries occupying in the total queries</td>
</tr>
</tbody>
</table>

Table 2: Throughput (elements/sec) of PosFilter and YFilter when $N = 100, 500$ and $1,000$ and $P = 1$.

<table>
<thead>
<tr>
<th></th>
<th>100</th>
<th>500</th>
<th>1,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>YFilter</td>
<td>18,588</td>
<td>15,795</td>
<td>14,479</td>
</tr>
<tr>
<td>PosFilter</td>
<td>22,786</td>
<td>20,441</td>
<td>18,051</td>
</tr>
</tbody>
</table>

Fig. 9 Throughput of PosFilter and YFilter when $N = 100$ and $P = 0$–100%.

Fig. 8 Trace of the number of state transition in each NFA machine (at the end-of-element in PosFilter and the start-of-element event in YFilter).
and YFilter, it uses its own specific memory organization and path matching algorithm. In this subsection, we present an experimental result for the filtering performance of both methods.

Since we could not obtain the optimized implementation of AFilter, we performed the performance analysis through measuring the number of states pushed into the runtime stack of each filtering machine. The number of states pushed into the runtime stack should be associated closely with the throughput. The reason is that pushing one state into the stack means that there are many related operations such as the stack operation of push and pop, the automata operation of state transition, and memory-related operations. Therefore, we can roughly compare the filtering performance of both methods by measuring the number of pushed states.

Especially, we compare PosFilter approach with the suffix-clustering approach of AFilter with respect to postfix sharing. In addition, since AFilter requires an additional operation of the pointer traversal for the backtracking path matching algorithm, we counted the number of pointer traversal in addition to the number of pushed states. Such traversal cost is no less significant than the cost of state transition. In the graph of Fig. 10, the measurement for AFilter indicates summing up the number of pointer traversal and the number of pushed states.

In the graph of Fig. 10, we can see that the throughput of PosFilter is similar to that of AFilter at $P = 20\%$, 40\%, 60\%, and 80\%. And PosFilter has a higher throughput than AFilter at $P = 100\%$; note that the larger the number of pushed states, the longer the filtering time. However, considered as a whole, we can say that the performances of two methods are similar. Through some other experimental results which are not presented in this paper, we also observed that there is no significant difference in the number of pushed states and pointer traversals of two methods. However, as mentioned at the end of Sect. 1, although PosFilter has just a similar filtering performance as in our experiments or AFilter can give further improvement, PosFilter following the NFA scheme should give more substantial benefits in terms of the expressiveness and the incremental maintenance.

Some comments on the pointer traversal of AFilter. Figs. 11 and 12 show the stack operation in the NFA of YFilter of Fig. 2 and in the StackBranch of AFilter[5] respectively. We can see that the number of the run-time states of AFilter is smaller than the number of the run-time states of YFilter. When $<\text{title}>$ is read in Fig. 11, there exist 13 states ($=\{1\}, \{2\}, \{2, 3\}, \{2, 4, 6, 9\}, \{2, 5, 7, 8, 10\}$) in the runtime stack. However, in Fig. 12, there exist 7 states ($=\{n_1, s_1, a_1, t_1, s_1^*, a_1^*, t_1^*\}$). Therefore, we can think that the StackBranch is compacter than the NFA. However, there are a number of branches in AFilter: $e_1$, $e_2$, $e_5$, $e_6$, $e_9$, $e_{10}$, and $e_{11}(\times 2)$. We should consider the additional cost of the pointer traversal over the branches besides the stack operation. See the Fig. 6 (b). Compared with YFilter, PosFilter also reduces the number of run-time states by using the proposed bottom-up filtering approach exploiting the common postfixes.

6. Related Work

XFilter system[2] may be an early study on an automata-based XML filtering. The research point is to use Finite State Machine (FSM) and an inverted index on all the subscribed XPath queries. XFilter converts each XPath query to a FSM and an inverted index (called Query Index) is built over the states of all FSMs. When a start-of-element event is triggered, the related SAX (Simple API for XML) event
handler looks up the element name in the Query Index, and it is checked whether or not there are matched queries. Using the Query Index gives the benefit of achieving high performance filtering. The features of XFilter system are its simple construction and maintenance and the high performance filtering and scalability.

However, XFilter does not consider exploiting the commonality that exists among path expressions. Exploiting the commonality can bring about the effect of reducing redundant processing. Thus, to eliminate such redundancy, YFilter system [3] has been proposed. YFilter combines all queries into an NFA. As such, the common prefixes of queries are represented only once in the NFA. For example, for the registered XPath queries '/a/b/c' and '/a/b/d', the state transitions labeled by 'a' and 'b' are represented only once. Although using an NFA can lead to performance degradation due to the multiple transitions from each state against Deterministic Finite Automaton (DFA), YFilter showed that applying the NFA to the XML filtering system is reasonable. The reason is that the cost of path evaluation using the NFA is not the dominant cost of filtering, and rather, other costs such as document parsing are more expensive. Therefore, YFilter is valuable in that it provides the substantial benefits of expressiveness and incremental maintenance provided by the NFA model along with assuring the reasonable speed of path matching. However, in this paper, we have investigated an NFA approach based on the postfix sharing which YFilter does not exploit.

Recently, AFilter system [5] has been suggested to leverage both prefix and postfix commonalities for further improving the filtering performance. Their basic idea is to exploit the postfix sharing similar to our research work, and furthermore, they introduce a novel technique for the adaptive processing of the prefix and the postfix based filtering. However, AFilter's method is not an automata-based scheme. They contrived their own specific memory organization (named AxisView, PRLabel-tree, SFLabel-tree, and StackBranch) and a specialized path matching algorithm. In addition, although they suggested the late unfolding approach of postfix clusters for the adaptive processing, they did not show how to define the detailed condition for the unfolding (i.e., the switch from postfix sharing to prefix sharing) and how much is the overhead of calculating the condition, which is able to significantly affect the overall filtering performance. Authors only mention that the unfolding condition is in the following cases: (1) the postfix clusters are small or (2) the prefix cache hit rate is low.

XTrie [4] utilizes trie structure to support the simple and effective filtering for complex XPath expressions. Each path expression is represented as a string, and XTrie indexes the strings into a trie structure. Whenever the string sequence of an XML document is inputted, the trie is used for detecting the occurrence of matching substring.

So far, we have reviewed the main studies for XML filtering from the viewpoint of the path sharing, but there have been many XML filtering studies from the other viewpoints. In order to avoid performance degradation by the multiple transition of the NFA, a technique changing NFA to DFA such as XMLTK [7] was suggested. XPush [8] suggested an efficient technique to translate a massive collection of filter statements into a single deterministic pushdown automaton. TwigM [9] suggested a polynomial time algorithm to evaluate XPath queries with predicates and ancestor-descendant axis.

In addition, in the paper [10], we already introduced the concept of the bottom-up filtering approach. In this paper, we have revised some contents in the previous article, added some experimental results, and particularly we have explained the combining approach of PosFilter and YFilter in Sect. 4.

7. Conclusions

In this paper, we have introduced a bottom-up filtering approach for further improving filtering performance. The suggested PosFilter system exploits the postfix commonalities shared across subscribed XPath queries. As indicated in Sect. 2, if an XML data stream does not have any defined schema, or users cannot remember the defined schema exactly, there is a high possibility that the partial matching path query with postfix sharing are frequently subscribed. Our PosFilter approach exploiting the postfix sharing should be useful for such an XML filtering situation. As compared to AFilter, which is a recent study on postfix sharing, some experimental results have shown that there is no significant difference between the two methods. Furthermore, since our PosFilter approach basically follows the NFA model, we believe that it should provide the benefits of better scalability and simpler maintenance along with assuring the high performance filtering.

Acknowledgments

We would like to thank the anonymous referees for their valuable comments on earlier draft of this paper. We also would like to thank Hyunjoo Kim, Jinyoung Kim, and Hanyoung Ko who participated in the implementation and experiments for PosFilter.

This study is supported in part by the Second Stage of BK21. In addition, this work was supported by the Korea Science and Engineering Foundation(KOSEF) grant funded by the Korea government(MOST) (No. R01-2006-000-10609-0).

References

Appendix: Test Queries in Experiment 2

- Some Partial Matching Path Queries with Postfix Sharing.

  //item/name, //asia/*/name,
  //europe/*/name, //regions/*/item/name,
  //africa/*/item/name, //samerica/*/item/name,
  //open_auctions/*/annotation/description,
  //closed_auctions/*/annotation/description,
  //person/name, //name,
  //description, //people/*/name,
  //regions/*/item/name,
  //category/description,
  //open_auction/description,
  //item/description, //asia/*/description,
  //africa/*/description, //mail,
  //item/mail, //categories/category/description,
  //asia/*/name, //europe/*/name,
  //regions/*/item/name, //people/*/name,
  //category/description, //itemref,
  //closed_auctions/itemref,
  //closed_auctions/*/itemref,
  //open_auctions/itemref,
  //open_auctions/*/itemref,
  //regions/asia/*/name,
  //regions/afirc/*/name,
  //asia/*/name, //africa/*/name,
  //asia/*/*, //africa/*/*
Jaehoon Kim received the B.S. and M.S. degrees in computer science from Konkuk University in 1997 and 1999, respectively and the Ph.D. degree in computer science from Sogang University in 2005. He worked as a senior researcher at Telecommunication R&D center in Samsung Electronics Company from 2005 to 2006. From 2006, he is a research professor of computer science at Sogang University. His major research areas are Semantic Web, Web database, and database security.

Youngsoo Kim received the B.S. and M.S. degrees in computer science from Sogang University in 2003 and 2005, respectively. From 2005, she is a researcher at Telecommunication R&D center in Samsung Electronics Company. Her major research areas are XML, Web database, and database security.

Seog Park received the B.S. degree in computer science from Seoul National University, Korea, in 1978, the M.S. and the Ph.D. degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 1980 and 1983, respectively. He is a professor of computer science at Sogang University, Seoul, Korea. Since 1983, he has been working in the Department of Computer Science of the College of Engineering, Sogang University. His major research areas are database security, real-time systems, data warehouse, digital library and Web database. Dr. Park is a member of the IEEE Computer Society, ACM, and the Korea Information Science Society. Also, he has been a member of Database Systems for Advanced Applications (DASFAA) steering committee since 1999.