High-Performance XML Filtering: An Overview of YFilter

Yanlei Diao, Michael J. Franklin

University of California, Berkeley, CA 94720
{diaoyl, franklin}@cs.berkeley.edu

Abstract

We have developed YFilter, an XML filtering system that provides fast, on-the-fly matching of XML-encoded data to large numbers of query specifications containing constraints on both structure and content. YFilter encodes path expressions using a novel NFA-based approach that enables highly-efficient, shared processing for large numbers of XPath expressions. In this paper, we provide a brief technical overview of YFilter, focusing on the NFA model, its implementation, and its performance characteristics.

1 Introduction

Today, it is widely agreed that in distributed computing scenarios such as Web Services, data and application integration, and personalized content delivery, XML is the way that data to be exchanged will be encoded. This use of XML has spawned significant interest in techniques for filtering XML data. In an XML filtering system, continuously arriving streams of XML documents are passed through a filtering engine where documents are matched to query specifications representing data interests of users or applications, and the matched documents are delivered accordingly. Queries in these systems are expressed in a language such as XPath [4], which is used to specify constraints over both structure (using path expressions) and content (using value-based predicates).

An earlier project, XFilter [1], pioneered the use of event-based parsing and *Finite State Machines* (FSMs) for fast structure-oriented XML filtering. In XFilter, XPath expressions are converted in to FSMs by mapping location steps of the the expressions to machine states. Arriving XML documents are then parsed with an event-based (SAX) parser, and the events raised during parsing are used to drive the FSMs through their various transitions. A query is determined to match a document if during parsing an accepting state for that query is reached. In XFilter, a separate FSM is created for each distinct path expression and a sophisticated indexing scheme is used during processing to locate potentially relevant machines and to execute those machines simultaneously. The indexing scheme and several optimizations provide a substantial performance improvement over a more naive approach. The drawback, however, is that by creating a separate FSM for each distinct query, XFilter fails to exploit commonality among the path expressions, and thus, may perform redundant work.

Based on this insight, we have developed YFilter, an XML filtering system aimed at providing efficient filtering for large numbers (e.g., 10's or 100's of thousands) of query specifications. The key innovation in YFilter is an Nondeterministic Finite Automaton (NFA)-based representation of path expressions which combines all

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

queries into a *single* machine. YFilter exploits commonality among path queries by merging the common prefixes of the paths so that they are processed at most once. The resulting shared processing provides tremendous improvements in structure matching performance over algorithms that do not share such processing or exploit sharing to a more limited extent. The NFA-based implementation also provides additional benefits including a relatively small number of machine states, incremental machine construction, and ease of maintenance.

An important challenge that arises due to the shared structure matching approach of YFilter is the handling of value-based predicates that address contents of elements. We have developed two alternative approaches to handling such predicates. One approach evaluates predicates as soon as the addressed elements are read from a document, while the other delays predicate evaluation until the corresponding path expression has been entirely matched. A further complication that arises in this regard is that of "nested paths". Since predicates may also reference other elements in an XML document, we employ a query decomposition scheme to take advantage of the shared path processing, and use special post-processing to return the final query evaluation results.

The remainder of this paper is organized as follows. The logical model of the NFA-based shared path processing approach is presented in Section 2. The implementation of the approach and techniques for predicate evaluation are described in some detail in Section 3. Section 4 discusses related work, and Section 5 presents conclusions and future work.

2 An NFA-based Model for Shared Path Processing

The basic path matching engine of YFilter handles query specifications that are written in a subset of XPath.[1] XPath allows parts of XML documents to be addressed according to their logical structure. A query path expression in XPath is composed of a sequence of location steps. Each location step consists of an axis, a node test and zero or more predicates. An axis specifies the hierarchical relationship between the nodes. We focus on two common axes: the parent-child operator '/', and the ancestor-descendent operator "//". We support node tests that are specified by either an element name or the wildcard operator '*' (which matches any element name). Predicates can be applied to address contents of elements or to reference other elements in the document

2.1 An NFA-based Model with an Output Function

Any single path expression written using the axes and node tests described above can be transformed into a regular expression. Thus, there exists a Finite State Machine (FSM) that accepts the language described by such a path expression [9]. In YFilter, we combine all of the path queries into a single FSM that takes a form of *Nondeterministic Finite Automaton* (NFA). All common prefixes of the paths are represented only once in the NFA.

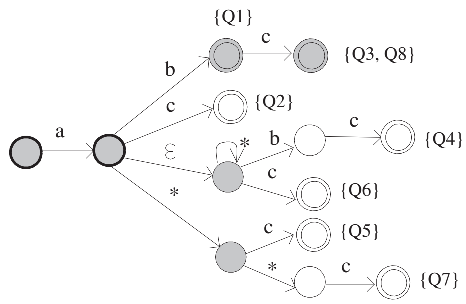
Figure 1 shows an example of such an NFA, representing eight queries (we describe the process for constructing such a machine in the following section). A circle denotes a state. Two concentric circles denote an accepting state; such states are also marked with the IDs of the queries they represent. A directed edge represents a transition. The symbol on an edge represents the input that triggers the transition. The special symbol "*" matches any element. The symbol "ϵ" is used to mark a transition that requires no input. In the figure, shaded circles represent states shared by queries. Note that the common prefixes of all the queries are shared. Also note that the NFA contains multiple accepting states. While each query in the NFA has only a single accepting state, the NFA represents multiple queries. Identical (and structurally equivalent) queries share the same accepting state (recall that at the point in the discussion, we are not considering predicates).

This NFA can be formally defined as a Moore Machine [9]. The output function of the Moore Machine here is a mapping from the set of accepting states to a partitioning of identifiers of all queries in the system, where each partition contains the identifiers of all the queries that share the accepting state.

[1] In more recent work we show how to use YFilter to handle more sophisticated queries written in XQuery [6].

Q1=/a/b
Q2=/a/c
Q3=/a/b/c
Q4=/a//b/c
Q5=/a/*/c
Q6=/a//c
Q7=/a/*/*/c
Q8=/a/b/c

(a) XPath queries



(b) A corresponding NFA

Figure 1: XPath queries and their representation in YFilter

Some Comments on Efficiency. A key benefit of using an NFA-based approach is the tremendous reduction in machine size it affords. It is reasonable to be concerned that using an NFA-based model could lead to performance problems due to (for example) the need to support multiple transitions from each state. A standard technique for avoiding such overhead is to convert the NFA into an equivalent DFA [9]. A straightforward conversion could theoretically result in severe scalability problems due to an explosion in the number states. But, as pointed out in [8], this explosion can be avoided in many cases by placing restrictions on the set of DTDs (i.e., document types) and queries supported, and lazily constructing the DFA.

Our experimental results (reported in [5], however, indicate that such concerns about NFA performance in this environment are unwarranted. In fact, in the YFilter system, path evaluation (using the NFA) is sufficiently fast, that it is in many cases not the dominant cost of filtering. Rather, other costs such as document parsing and result collection are often more expensive than the basic path matching. Thus, while it may in fact be possible to further improve path matching speed, we believe that the substantial benefits of flexibility and ease of maintenance provided by the NFA model outweigh any marginal performance improvements that remain to be gained by even faster path matching.

2.2 Constructing a Combined NFA

Having presented the basic NFA model used by YFilter, we now describe an incremental process for NFA construction and maintenance. The shared NFA shown in Figure 1 was the result of applying this process to the eight queries shown in that figure.

The four basic location steps in our subset of XPath are ”/a”, ”//a”, ”/*” and ”//*”, where ’a’ is an arbitrary symbol from the alphabet consisting of all elements defined in a DTD, and ’*’ is the wildcard operator. Figure 2(a) shows the directed graphs, called NFA fragments, that correspond to these basic location steps. Note that in the NFA fragments constructed for location steps with ”//”, we introduce an ϵ-transition moving to a state with a self-loop. This ϵ-transition is needed so that when combining NFA fragments representing ”//” and ”/” steps, the resulting NFA accurately maintains the different semantics of both steps (see the examples in Figure 2(b) below). The NFA for a path expression, denoted as NFA_p, can be built by concatenating all the NFA fragments for its location steps. The final state of this NFA_p is the (only) accepting state for the expression.

NFA_ps are combined into a single NFA as follows: There is a single initial state shared by all NFA_ps. To insert a new NFA_p, we traverse the combined NFA until either: 1) the accepting state of the NFA_p is reached, or 2) a state is reached for which there is no transition that matches the corresponding transition of the NFA_p. In the first case, we make that final state an accepting state (if it is not already one) and add the query ID to the query set associated with the accepting state. In the second case, we create a new branch from the last state reached in the combined NFA. This branch consists of the mismatched transition and the remainder of the

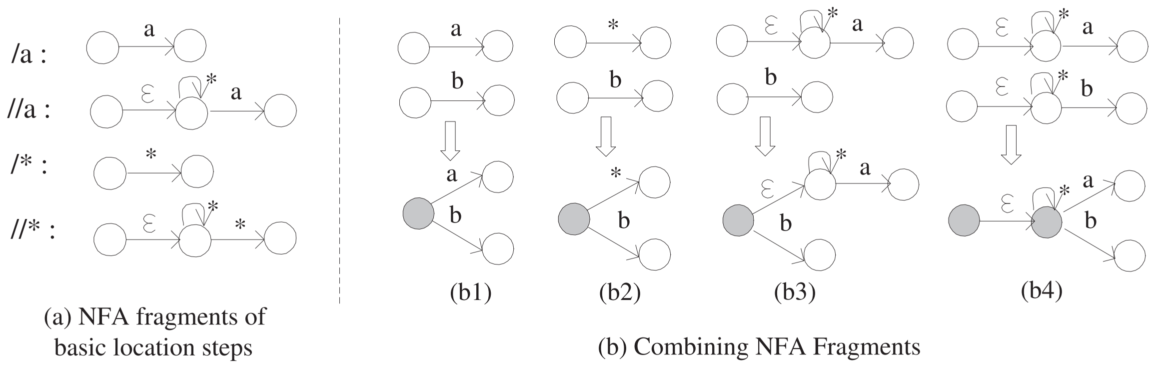


Figure 2: NFA fragments for location steps, and examples of merging NFA fragments

NFA_p. Figure 2(b) provides four examples of this process.

It is important to note that because NFA construction in YFilter is an incremental process, new queries can easily be added to an existing system. This ease of maintenance is a key benefit of the NFA-based approach.

3 Implementation of the NFA-based Path Processing

The previous section described YFilter's NFA model and its logical construction. In this section, we present the implementation of the NFA approach and describe its execution.

3.1 Implementing the NFA

For efficiency we implement the NFA using a hash table-based approach. Such approaches have been shown to have low time complexity for inserting/deleting states, inserting/deleting transitions, and actually performing the transitions [12]. In this approach, a data structure is created for each state, containing: 1) The ID of the state, 2) type information (i.e., if it is an accepting state or a //-child as described below), 3) a small hash table that contains all the legal transitions from that state, and 4) for accepting states, an ID list of the corresponding queries.

The transition hash table for each state contains [symbol, stateID] pairs where the symbol, which is the key, indicates the label of the outgoing transition (i.e., element name, '*', or 'ϵ') and the stateID identifies the child state that the transition leads to. Note that the child states of the 'ϵ' transitions are treated specially. Recall that such states have a self-loop marked with '*' (see Figure 2(a)). For such states, (called "//-child" states) we do not index the self-loop. As described in the next section, this is possible because transitions marked with 'ϵ' are treated specially by the execution mechanism.

3.2 Executing the NFA

Having walked through the logical construction and physical implementation we can now describe the execution of the machine. Following the XFilter approach, we chose to execute the NFA in an event-driven fashion. As an arriving document is parsed, the events raised by the parser drive the transitions in the NFA. The nesting of XML elements requires that when an "end-of-element" event is raised, NFA execution must backtrack to the states it was in when the corresponding "start-of-element" was raised. A stack mechanism is used to enable the backtracking. Since many states can be active simultaneously in an NFA, the run-time stack mechanism must be capable of tracking multiple active paths. Details are described in the following.

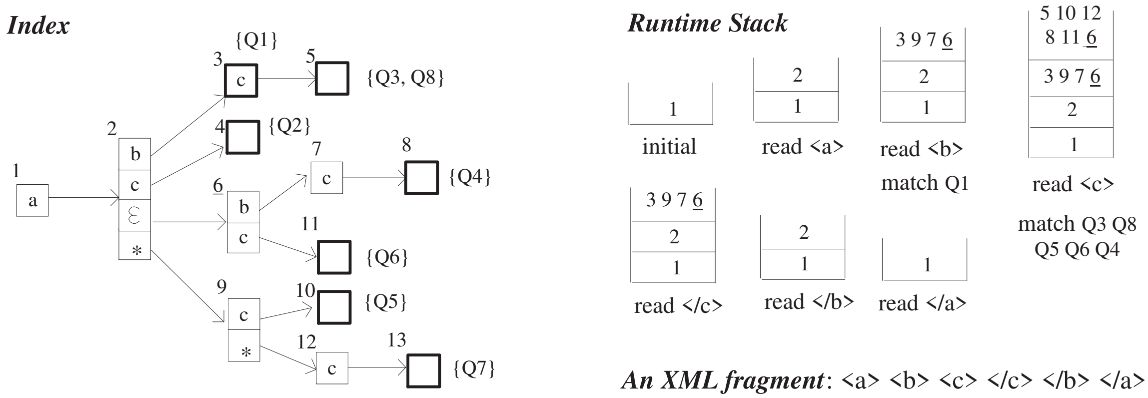


Figure 3: An example of the NFA execution

Start of Document. When an XML document arrives to be parsed, the execution of the NFA begins at the initial state. That is, the common initial state is pushed to the runtime stack as the active state.

Start of Element. When a new element name is read from the document, the NFA execution follows all matching transitions from all currently active states, as follows. For each active state, four checks are performed: 1) The incoming element name is looked up in the state's hash table. If it is present, the corresponding stateID is added to a set of "target states". 2) The '*' symbol is also looked up in the hash table. If it exists, its stateID is also added to the set of target states. Since the '*' symbol matches any element name, a transition marked by it is always performed. 3) Then, the type information of the state is checked. If the state itself is a "//-child" state, then its own stateID is added to the set, which effectively implements a self-loop marked by the '*' symbol in the NFA structure. 4) Finally, to perform an ϵ-transition, the hash table is checked for the 'ϵ' symbol, and if one is present, the //-child state indicated by the corresponding stateID is processed recursively, according to steps 1-3 above.

After all the currently active states have been checked in this manner, the set of "target states" is pushed onto the top of the run-time stack. They then become the "active" states for the next event. If a state in the target set is an accepting state, which means it has just been reached during reading the last element, the identifiers of all queries associated with the state are collected and added to an output data structure.

End of Element. When an end-of-element is encountered, backtracking is performed by simply popping the top set of states off the stack.

Finally, it is important to note that, unlike a traditional NFA, whose goal is to find one accepting state for an input, our NFA execution must continue until all potential accepting states have been reached. This is because we must find all queries that match the input document.

An example of this execution model is shown in Figure 3. On the left of the figure is the index created for the NFA of Figure 1. The number on the top-left of each hash table is a state ID and hash tables with a bold border represent accepting states. The right of the figure shows the evolution of the contents of the runtime stack as an example XML fragment is parsed. In the stack, each state is represented by its ID. An underlined ID indicates that the state is a //-child.

3.3 Predicate Evaluation

The discussion so far has focused on the structure matching aspects of YFilter. In an XPath expression, however, predicates can be applied to address properties of elements, such as their text data, their attributes and their position. We refer to these as *value-based* predicates. In addition, predicates may also include other path expressions, which are called nested paths. Any number of such predicates can be attached to a location step in a path expression. In this section, we briefly describe the techniques used in YFilter to support the evaluation of these predicates.

Given the NFA-based model for path-matching, an intuitive approach to supporting value-based predicates would be to extend the NFA by including additional transitions to states that represent the successful evaluation of the predicates. Unfortunately, such an approach could result in an explosion of the number of states in the NFA, and would destroy the sharing of path expressions, the primary advantage of using an NFA. Instead, in YFilter, we use a separate selection operator that evaluates value-based predicates by interacting with the NFA-based processing of path expressions. Traditional relational query processing uses the heuristic of pushing selections down in the query plan so that they are processed early in the evaluation. Following this intuition, we developed an approach called *Inline*, that processes value-based predicates as soon as the elements in path expressions that those predicates address are matched during structure matching. We also developed an alternative approach, called *Selection Postponed* (SP), that waits until an entire path expression is matched during structural matching, and at that point applies all the value-based predicates for the matched path.

Nested paths are handled by first decomposing queries into their constituent paths and then inserting all of these paths into the path matching engine. These paths are matched using the shared path matching approach described above. Then, we use a separate collection operator to process the matches of constituent paths and return the final query results. A more detailed description of these techniques is provided in the full paper on YFilter [5].

3.4 Overview of the Performance Results

We have performed a detailed performance study of our YFilter implementation [5]. In the study, we compared the performance of the NFA-based path matching approach in YFilter, the FSM-per-query approach used by XFilter, and a hybrid approach that exploits a reduced degree of shared path processing. We also investigated the tradeoffs between the Inline and SP approaches to value-based predicates. The results of the studies can be summarized as follows:

1. YFilter can provide order of magnitude performance improvements over both XFilter and the hybrid approach. In fact, as discussed earlier, path processing using YFilter is sufficiently fast that in many cases it is outweighed by other costs for XML filtering such as document parsing and result collection.

2. The NFA-based approach is robust and efficient under query workloads with varying proportions of "//" operators and '*' operators. This is important because it is these operators that introduce non-determinism into the path matching process. The NFA-based approach was also shown to perform well using a number of DTDs with different characteristics.

3. The maintenance cost (i.e., as queries are added and removed) of the NFA structure is small, due to the incremental construction that an nondeterministic version of a FSM enables and due to the sharing of structure inherent in the NFA approach.

4. For value-based predicates, the SP approach was found to perform much better than the Inline approach. The Inline approach suffers because early predicate evaluation cannot eliminate future work of structure matching or predicate evaluation, due to the shared nature of path matching and the effect of recursive elements in the presence of "//" operators in path queries. In contrast, SP uses path matching to prune the set of queries for which predicate evaluation needs to be considered, thus achieving a significant performance gain.

4 Related Work

A number of XML filtering systems have been developed to efficiently match XPath queries with streaming documents. XFilter [1] builds a Finite State Machine (FSM) for each path query and employs a query index on all the FSMs to process all queries simultaneously. XTrie [3] indexes sub-strings of path expressions that only contain parent-child operators, and shares the processing of the common sub-strings among queries using the index. In [8], all path expressions are combined into a single DFA, resulting in good performance but with significant limitations on the flexibility of the approach. YFilter and Index-Filter are compared through a detailed performance study in [2]. MatchMaker [10] is the only published work reporting its performance on shared tree pattern matching. Using disk-resident indexes on pattern nodes and path operators, it labels document nodes with all matching queries. I/O invocations limited its matching efficiency. Other related work includes publish/subscribe systems, such as Xlyeme [7] and Le Subscribe [11]. A common feature of these systems is the use of restricted profile languages, e.g. a set of attribute value pairs, and data structures tailored to them for high system throughput.

5 Conclusions

In this paper we have presented a technical overview of the basic structure and value-based matching approaches of YFilter with a focus on its novel NFA-based approach to shared processing of path expressions. Our work has shown that the NFA-based approach can provide high-performance XML filtering for large numbers of queries that contain both structure-based and value-based constraints.

More recently we have been extending YFilter in two important directions. First, we have investigated the use of YFilter in a more general XML Message Brokering scenario [6]. XML filtering represents the lowest level of functionality required for XML-based data exchange in a distributed infrastructure. In many emerging applications in this environment, however, XML data must also be transformed on a query-by-query basis, in order to provide customized data delivery and to enable cooperation among disparate, loosely coupled services and applications. To support such transformations, we take the NFA-based path matching engine as the basis, and develop alternative techniques that push the work of processing path expressions into the engine and perform efficient post-processing of the remaining portions of queries to generalize customized results. Second, as XML filtering systems are to be deployed in a distributed wide-area environment, our current efforts are aimed at studying the deployment of such systems as the foundation of an overlay network that supports content-based routing of documents and queries and intelligent delivery that allows shared transmission of query results.

6 Acknowledgments

We would like to thank Hao Zhang for the early discussion on the NFA-based path matching algorithm, Peter Fischer and Raymond To for helping us develop YFilter, and Philip Hwang for helping provide insight into XML parsing. We also owe thanks to Joseph M. Hellerstein, Samuel R. Madden, Sirish Chandrasekaran, Sailesh Krishnamurthy, and Ryan Huebsch for their valuable comments and feedback.

This work has been supported in part by the National Science Foundation under the ITR grants IIS0086057 and SI0122599 and by Boeing, IBM, Intel, Microsoft, Siemens, and the UC MICRO program.

References

[1] ALTINEL, M., AND FRANKLIN, M. J. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of VLDB Conference* (2000).

[2] BRUNO, N., GRAVANO, L., KOUDAS, N., AND SRIVASTRAVA, D. Navigation- vs. index-based XML multi-query processing. In *Proceedings of IEEE Conference on Data Engineering* (2003).

[3] CHAN, C. Y., FELBER, P., GAROFALAKIS, M. N., AND RASTOGI, R. Efficient filtering of XML documents with XPath expressions. In *Proceedings of IEEE Conference on Data Engineering* (2002).

[4] CLARK, J., AND DEROSE, S. XML path language XPath - version 1.0. `http://www.w3.org/TR/xpath`, November 1999.

[5] DIAO, Y., ALTINEL, M., FRANKLIN, M. J., ZHANG, H., AND FISCHER, P. Path sharing and predicate evaluation for high-performance XML filtering. Submitted for publication. Available at `http://www.cs.berkeley.edu/~diaoyl/publications/yfilter-public.pdf`, 2002.

[6] DIAO, Y., AND FRANKLIN, M. J. Query processing for high-volume XML message brokering. Tech. rep., University of California, Berkeley, 2003.

[7] FABRET, F., JACOBSEN, H.-A., LLIRBAT, F., PEREIRA, J., ROSS, K. A., AND SHASHA, D. Filtering algorithms and implementation for very fast publish/subscribe. In *Proceedings of ACM SIGMOD Conference* (2001).

[8] GREEN, T. J., MIKLAU, G., ONIZUKA, M., AND SUCIU, D. Processing XML streams with deterministic automata. In *Proceedings of IEEE Conference on Database Theory* (2003).

[9] HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages and Computation*. Addition-Wesley Pub. Co, 1979.

[10] LAKSHMANAN, L. V., AND PARTHASARATHY, S. On efficient matching of streaming XML documents and queries. In *Proceedings of International Conference on Extending Database Technology* (2002).

[11] NGUYEN, B., ABITEBOUL, S., COBENA, G., AND PREDA, M. Monitoring XML data on the web. In *Proceedings of ACM SIGMOD Conference* (2001).

[12] WATSON, B. W. Practical optimization for automata. In *Proceedings of International Workshop on Implementing Automata* (1997).