NFA-based Filtering for Efficient and Scalable XML Routing

Yanlei Diao, Hao Zhang, Michael J. Franklin

Computer Science Division

University of California, Berkeley

{*diaoyl, nhz, franklin*}*@cs.berkeley.edu*

Abstract

Soon, much of the data exchanged over the Internet will be encoded in XML. This encoding can be used as the basis for sophisticated filtering and content-based routing of data using XML queries. Filtering systems such as XFilter represent XML path expressions as *Finite State Machines* and index them; In contrast, work on continuous query processing for database systems has focused on grouping common parts of relational-style queries. It is clear that for scalable and efficient XML filtering and routing both types of techniques will be needed. We propose a new approach based on *Nondeterministic Finite Automata* (NFA) that indexes path expressions and captures the overlap between queries naturally. The approach also has the potential of gracefully handling other problems in this environment such as concurrent filtering and online updates. Preliminary experimental results show that the NFA approach can dramatically improve filtering performance.

1 Introduction

The vast range of information available on the Internet has spawned tremendous interest in techniques for filtering and routing of data based on user preferences. Systems that perform this function are known as Publish/Subscribe systems. The ability to augment data with semantic and structural information using XML-based languages raises the potential for much more accurate and useful delivery of data. In an XML-based publish/subscribe system, users express their interests as queries over XML documents. Typically these queries involve path expressions. Continuously arriving streams of XML documents are filtered and routed based on these queries. For systems that support many users, filtering efficiency and system scalability are of paramount concern.

Publish/subscribe systems have long been developed using Information Retrieval techniques based on both the Boolean and the "bag-of-words" models. More recently database researchers have been developing Continuous Query systems such as *NiagaraCQ* [CDT00], *OpenCQ* [LPT99] and *TriggerMan* [HCH99], that aim to provide similar functionality using models based on relational and XML query languages. A key optimization used by some of these systems is the grouping of similar queries in order to improve performance and scalability by minimizing redundant work. At the same time, other projects have investigated query-based filtering over streams of XML encoded-data [AF00, ILW00]. This latter work has focused on the efficient evaluation of path expressions over streaming data, using some form of a *Finite State Machine* (FSM) to represent the path expressions.

FSMs are a natural and effective way to represent and process path expression queries. Each element of a path expression is mapped to a state. A transition is fired when an element is found in the document that matches that transition from the currently active state. If an "accept" state is reached, then the document is said to satisfy the query. When an XML document arrives to be filtered, it can be parsed with an event-based parser; each time a new element or the end of an element is encountered, an event is raised. These events trigger transitions in the query FSMs.

The original work on *XFilter* [AF00] focused on techniques to allow many queries (FSMs) to be processed *simultaneously*. XFilter implements a special index and related optimizations so that when an XML document arrives to be filtered, the FSMs that represent potential matching queries can be quickly identified and executed. Such techniques were indeed shown to be effective, but unlike the work on database continuous queries, no attempt was made to eliminate redundant processing by combining similar queries. For large-scale systems, however, it is likely that significant commonality among user interests will exist. Thus, we have developed an alternative approach to supporting XML-based publish/subscribe systems. Rather than representing each query as a separate *Deterministic Finite Automaton* (DFA), the approach we describe here combines multiple queries into a single *Nondeterministic Finite Automaton* (NFA). The use of an NFA inherently allows a dramatic reduction in the number of states needed to represent the set of user queries and also greatly simplifies the bookkeeping that must be done while simultaneously processing large numbers of queries. In addition, the approach also has the potential of gracefully

handling other problems in this environment such as concurrent filtering, online updates, and disk-resident index structures.

The remainder of the paper is organized as follows: In Section 2 we introduce and discuss the NFA-based model. In Section 3 we present the techniques of constructing a combined NFA, building an index, and executing the NFA. Preliminary experimental results are shown in Section 4. Section 5 concludes the paper with a discussion on future work.

2 An NFA-based Model

Following XFilter, our current work is based on XPath [CD99], which 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. The most common axes use a parent-child operator '/' or a descendent-or-self operator "//". A node test can be an element name or a wildcard operator '*', which matches any element name. Predicates can be applied to attributes of an element, to the contents of an element, or may contain references to other elements in the document. In our current NFA-based implementation, such predicates are not yet supported, so they are not discussed further in this paper.

The descendent-or-self operator "//" means the associated node test can be satisfied at any level at or below the current document level. These operators introduce non-determinism into the model. When query evaluation is at such a node, if a matching element arrives, the processing may either transition to the next state, or can remain in the current state awaiting further input. The bookkeeping mechanisms used to manage this non-determinism can also be exploited when combining multiple distinct queries into a single machine.

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

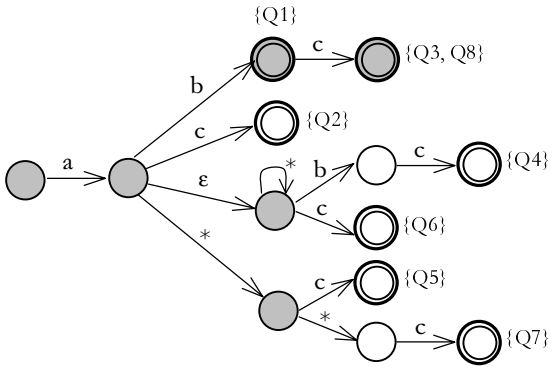

Figure 1: XPath Queries and A Corresponding NFA

Figure 1 shows an

example of an NFA that represents eight queries. A circle denotes a state, a directed edge represents a transition, and the symbol on the edge represents the input symbol that triggers the transition. Shaded circles represent states shared by queries. We use two concentric circles to represent an accept state and mark it with the IDs of the corresponding queries.

In the figure, note that the common prefixes of all the queries are shared. Note also, that the NFA contains multiple accept states, corresponding to the accept states of the individual queries. If multiple queries share the same accept state, then they are identical queries (recall that predicates are not handled in this implementation).

An arriving document is parsed and the events raised by the parser drive the transitions in the NFA. Since it is an NFA, many states can be active simultaneously. When an "end-of-element" event is raised the execution must backtrack to previous states. A run-time stack structure (described in the next section) is used to track the active and previously processed states. Finally, it is important to note that, unlike an NFA used to check a regular language, the filtering task here requires that processing continue until all possible accept states have been reached. The queries corresponding to the set of accept states ultimately reached are those that match the input document.

3 Implementation

3.1 Constructing a Combined NFA

As described in the previous section, two common relationships specified in the axis of a location step are given by the parent-child operator '/' and the descendent-or-self operator "//". The wildcard operator '*' is also supported in XPath to represent any element name in a node test. Thus, the three basic location steps are "/a", "//a", and "/*", where 'a' is an arbitrary element name.[1] Figure 2 shows the directed graphs, called *NFA fragments*, built for these basic location steps. In the NFA fragment for location step "//a", we introduce a null transition marked by the symbol 'ε' to a state with a self-loop. This null transition is

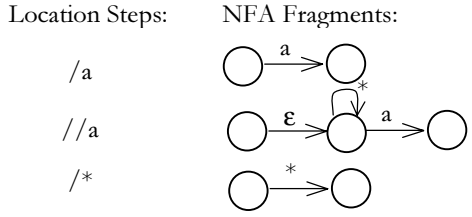

Figure 2: Three Basic Location Steps and their NFA Fragments

[1] XPath allows path expressions to begin with an element name. This is semantically equivalent to placing a "//" operator at the beginning of the path, and is treated as such by the algorithm.

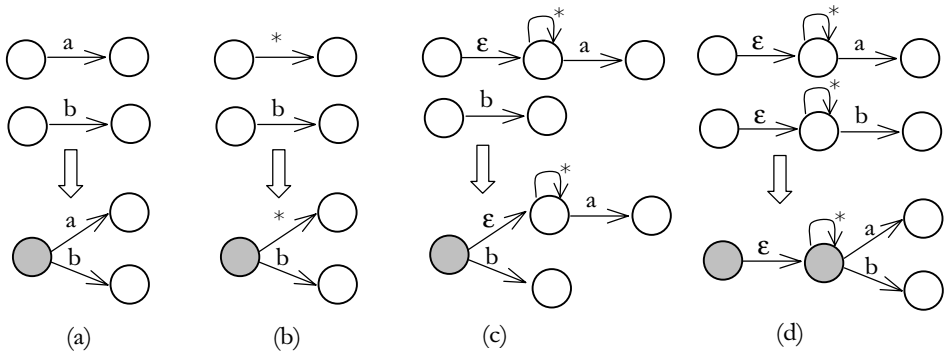

Figure 3: Combining NFAs

needed so that when combining NFA fragments representing "//" and "/" steps, the resulting NFA accurately maintains the different semantics of both steps (see the example in Figure 3(c) below).

The NFA for a path expression can be built by concatenating all the NFA fragments for its location steps. The final step of the NFA for a path expression is the accept state for that expression. Once NFAs have been constructed for all the queries, the next task is to combine them into a single NFA. This is done by inserting one query at a time. All query NFAs share the same initial state. To insert a new query, we traverse the combined NFA until either: 1) the accept state of the input query is reached, or 2) a state is reached for which there is no transition that matches the corresponding transition of the input query. In the first case, we make that final state an accept state (if it is not already one) and add the query ID to the query set associated with the accept state. In the second case we create a new branch from the last state reached in the combined NFA that consists of the mismatched transition and the remainder of the query NFA. Figure 3 shows four examples of this process.

Figure 3(a) shows the process of merging a fragment consisting of "/a" with a state in the combined NFA that represents a "/b" step. Figure 3(b) shows that this process treats the '*' symbol in the same way that it treats the other symbols in the alphabet. Figure 3(c) shows the process of merging a "//a" step with a "/b" step, while Figure 3(d) shows the merging of a "//a" step with a "//b" step. Note that in all cases, the simple rule specified above is followed.

The shared NFA shown in Figure 1 was the result of applying this process to the eight queries shown in that figure. Note that the process can also be applied incrementally, so that new queries can easily be added to an existing system.

3.2 Building an Index

After creating a combined NFA, the NFA is translated into a tree structure for efficient evaluation. In this index, a data structure is created for each state. The state data structure contains: 1) The ID of the state, 2) type information about the state (i.e., if it is an accept state or a //-child as described below), 3) a small hash table that contains all the legal transitions from that state, and 4) for accept states, a list of the corresponding queries is also maintained. The transition hash table contains [*symbol, stateID*] pairs where the *symbol*, which is the key, indicates the label of the transition, and the *stateID* is a pointer to the state that the transition leads to.

Note that in addition to element names, two special symbols can be used in the hash table. First is the wildcard symbol '*', which matches any element name. Second is a special symbol for "//" steps. Rather than creating hash table entries for the null transitions to self-loop states used to represent "//" steps in the NFA, the null transition is represented using the special "//" symbol and its hash table entry is set to point to the child (self-loop) state. As described in the next section, transitions marked with "//" are treated specially by the evaluation mechanism. This special treatment allows us to remove the self-loop from the child state. We refer to such a child state as a "//-child".

Because each state has its own hash table, in effect, the NFA is translated into a tree of hash tables. The implementation is thus a variant of a hash table based NFA, which has been shown to provide small time complexity for inserting/deleting states, inserting/deleting transitions, and actually performing the transitions [Wat97]. This is important for large NFA systems, as studies have shown that the performance bottleneck for many practical NFA systems is transition, including both the time spent in performing look ups on the transition table and the time to do the actual transition [Mao97].

3.3 Executing the NFA

When an XML document arrives, the execution of the NFA starts at the initial state. When a new element name is read from the XML document, the execution engine follows all

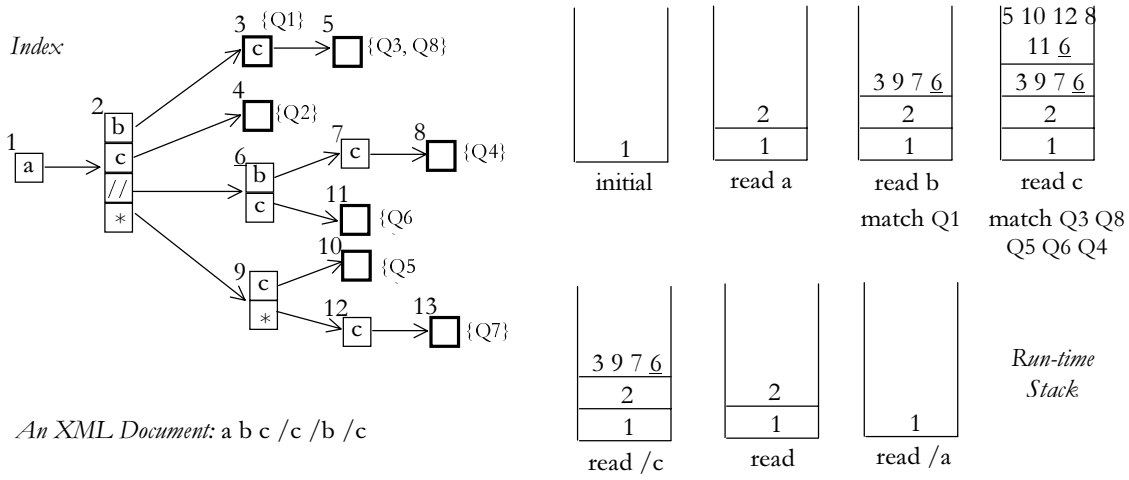

Figure 4: An Example of the NFA Execution. In the index, the number on the top-left of each hash table is a state ID and hash tables with a bold border represent accept states. In

matching transitions from all currently active states. The transitions are performed as follows. For each active state, four checks are performed. First, the incoming element name is looked up in the state's hash table. If it is present, the corresponding *stateID* is added to the list of target states. Second, the hash table is checked for a '*' symbol, and if one is present, its corresponding *stateID* is added to the list. Third, the type information of the state is checked, and if the state itself is a "//-child" state, then its own *stateID* is added to the list. Finally, the hash table is checked for the "//" symbol, and if one is present, the //-child state indicated by the corresponding *stateID* is processed immediately, according to these same rules.[2] Note that the processing done by the last two checks effectively implements a "self-loop" from the combined NFA; as stated in the previous section, such self-loops are removed when translating the combined NFA into its tree representation.

After all the currently active states have been checked in this manner, the list of target states is pushed onto the top of the run-time stack. The target states then become the "active" states for the next event. When an end of the element is encountered during the XML document parse, the algorithm needs to backtrack to previously active states. In our implementation, this backtracking is implemented by popping the run-time stack. When an

[2] Note that we collapse any instances of multiple adjacent "//" operators in to a single "//" operator, which is semantically equivalent. In this way, we ensure that the process does not traverse more than one additional level in the tree, since //-child nodes do not themselves contain a "//" symbol.

accept state is reached during this processing, all of the queries associated with the state are added to the list of matching queries.

An example of this execution model is shown in Figure 4. On the left of the figure is the index created for the NFA of the eight queries shown in Figure 1. The right of the figure shows the evolution of the contents of the runtime stack as an example XML document is parsed.

4 Initial Performance Results

In order to examine the potential performance benefits of the NFA-based approach, we implemented it and compared it to an implementation of the XFilter approach. Both of the implementations were written in Java. For the experiments, we followed the experimental set-up that was used in the original XFilter study [AF00]. Due to space considerations, we only summarize the key aspects of that set-up here. The reader is referred to [AF00] for additional details. User queries and XML documents were created based on the NITF (News Industry Text Format) DTD [Cov99]. XML documents were generated using IBM's XML Generator tool [IBM99]. We re-wrote the query generator of XFilter in Java that takes a DTD as input and creates a set of XPath queries based on input parameters. We developed an event-based parser using the Xerces toolkit [Apa99] that contains a validating XML parser supporting SAX 1.0, a standard interface for *event-based* XML parsing [Meg98]. The experiments were performed on a Sun Ultra-5 workstation with 128MB memory running JVM v1.1.6. All data structures were kept in memory in the experiments.

We studied the performance of both approaches under different workloads. Workloads were generated by changing the parameters of XML documents and user queries. Two workload parameters used were the number of queries (Q) and the maximum depth of XML documents and XPath queries (D). D took a value of {2, 4, 6, 8, 10}. We generated a set of XML documents for each value of D in advance. At the beginning of each set of experiments, we set D to a particular value and varied Q. For each Q value, we generated a set of queries with the fixed D and Q as input parameters, loaded them into either an XFilter query index or an NFA structure. In these experiments (as in [AF00]) the structures were kept in memory. Then a set of documents generated using the same maximum depth value (D) were read and matched against the queries. We measured "filter time" as the time to find all queries that match one XML document – the cost of generating documents and queries

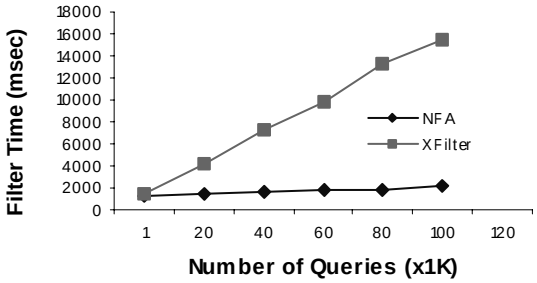

Figure 5: Filter Time with Varying Q

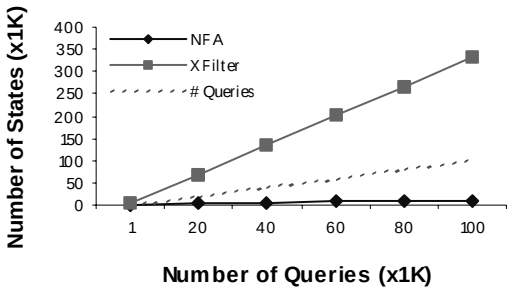

Figure 6: Number of States with Varying Q

were not included in this metric. The average filter time per document over a set of 20 XML documents were used for the graphs shown below.

Similar results were found for sets of experiments with different values of D. In the interest of space, we only show the results with D set to 6. Figure 5 shows the increase of filter time as the number of queries is increased. It can be observed that the NFA approach runs much faster than XFilter in this case. When the number of queries is large, the NFA runs 5 to 8 times faster. The performance gain comes partly from grouping queries and partly from fast transitions in the execution by using the new index.

To investigate the grouping benefits of the NFA approach, we also examined the number of states required in the XFilter and NFA approaches to store the query set. Figure 6 shows the results of these measurements for the experiment just described (i.e., D=6). The number of queries is also included in it for comparison. It can be observed that as expected, the number of states in NFA increases at a much slower rate than the number of states in XFilter because the latter does not group queries. Thus, we would expect the performance of the NFA approach to scale much better the XFilter in large-scale applications.

5 Conclusions

The vast range of information available on the Internet and the expected popularity of XML for data transfer has raised interest in techniques for large-scale filtering and routing of data based on user preferences. In this paper, we proposed an NFA-based approach for representing queries in an XML-based publish/subscribe system. The advantages of this approach are that it effectively indexes queries while also grouping them to avoid redundant storage and processing. We described the implementation of the index and a corresponding

execution algorithm. Preliminary experimental results show using this approach greatly improves filtering performance.

In a publish and subscribe system, there are many other issues to be solved. We believe that the NFA-based approach has the potential of gracefully solving most problems. One is concurrent filtering. Since our index does not change when the combined NFA is executed and all the filtering processes are read-based processes, they can be performed concurrently. Another problem is online updates. A locking mechanism can be applied to the tree-structured index. A good schedule of updates based on the privilege of the user and the urgency of the updates will provide better concurrency. Yet another problem is that when the number of user queries is large, the index does not fit in memory any more. Parallel computing based on hash table partitioning is naturally supported by our NFA-based approach.

References

[Apa99] Apache XML project. Xerces Java Parser 1.2.3 Release. http://xml.apache.org/xerces-j/index.html, 1999.

[AF00] M. Altinel, M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th International Conference on Very Large Data Bases*, Sep. 2000.

[CDT00] J. Chen, D. Dewitt, F. Tian and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of 2000 ACM SIGMOD International Conference on Management of Data*, May 2000.

[Cov99] R. Cover. The SGML/XML Web Page. http://www.w3.org/TR/xslt, Nov. 1999.

[HCH99] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park and A. Vernon. Scalable Trigger Processing. In *Proceedings of IEEE International Conference on Data Engineering*, Mar. 1999.

[IBM99] A. L. Diaz, D. Lovell. XML Generator. http://www.alphaworks.ibm.com/tech/xmlgenerator, Sep., 1999.

[ILW00] Z. Ives, A. Levy, D. Weld. Efficient Evaluation of Regular Path Expressions on Streaming XML Data. Technical Report, University of Washington, 2000.

[LPT99] L. Liu, C. Pu, W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *Special Issue on Web Technologies*, IEEE TKDE, Jan. 1999.

[Mao97] V. L. Maout. Tools to Implement Automata, a First Step: ASTL. In *Proceedings of the 2nd International Workshop on Implementing Automata*, Sep. 1997.

[Meg98] Megginson Technologies. SAX 1.0: A free API for event-based XML parsing. http://www.megginson.com/SAX/index.html, May, 1998.

[Wat97] B. W. Watson. Practical Optimization for Automata. In *Proceedings of the 2nd International Workshop on Implementing Automata*, Sep. 1997.