

YFilter: efficient and scalable filtering of XML documents

Yanlei Diao, Peter M. Fischer, Michael J. Franklin, Raymond To

Angaben zur Veröffentlichung / Publication details:

Diao, Yanlei, Peter M. Fischer, Michael J. Franklin, and Raymond To. 2003. "YFilter: efficient and scalable filtering of XML documents." In Proceedings: 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA, USA, edited by Rakesh Agrawal, Klaus Dittrich, and Anne H. H. Ngu, 341-42. Piscataway, NJ: IEEE. <https://doi.org/10.1109/icde.2002.994748>.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under the following conditions:

Deutsches Urheberrecht

Weitere Informationen finden Sie unter: / For more information see:

<https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publizieren>



YFilter: Efficient and Scalable Filtering of XML Documents

Yanlei Diao, Peter Fischer, Michael J. Franklin, Raymond To
Computer Science Division, EECS
University of California, Berkeley
{diaoyl, fischerp, franklin}@cs.berkeley.edu, raygto@uclink4.berkeley.edu

Abstract

Soon, much of the data exchanged over the Internet will be encoded in XML, allowing for sophisticated filtering and content-based routing. We have built a filtering engine called YFilter, which filters streaming XML documents according to XQuery or XPath queries that involve both path expressions and predicates. Unlike previous work, YFilter uses a novel NFA-based execution model. In this demonstration, we present the structures and algorithms underlying YFilter, and show its efficiency and scalability under various workloads.

1 Overview

Recently, there has been growing interest in the filtering and routing of data based on user preferences. In an XML filtering system, continuously arriving XML documents are routed to users according to subscriptions specified as queries. XML allows the encoding of semantic and structural information that can improve delivery accuracy. For large systems, filtering efficiency and scalability are of paramount concern.

Filtering systems have traditionally 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* [3] that provide similar functionality using relational and XML queries. A key optimization used in *NiagaraCQ* is the grouping of similar queries to minimize redundant work. At the same time, another project, *XFilter* [1], has focused on the efficient evaluation of path expressions over streaming XML data, using indexed *Finite State Machines* (FSM) to allow many structure-based queries to be processed *simultaneously*. *XFilter*, however, makes no attempt to eliminate redundant processing for similar queries.

FSMs are a natural and effective way to represent and process path expressions. Elements of a path expression are mapped to states. A transition from an active state is fired when an element is found in the document that matches that transition. If an accepting state is reached, then the document is said to satisfy the query. For large-scale systems, it is likely that significant commonality

among user interests will exist. Thus, we have developed an alternative approach we call *YFilter*, which combines multiple queries into a single *Nondeterministic Finite Automaton* (NFA). The use of a combined NFA allows a dramatic reduction in the number of states needed to represent the set of user queries and greatly improves filtering performance by sharing execution work. *YFilter* also extends this NFA model to efficiently handle predicates within path expressions.

2 NFA-based Path Navigation

As in *XFilter*, path expressions in our system are written in XPath. Such path expressions are 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. A node test is typically a name test, which can be an element name or a wildcard operator “*” that matches any element name. Predicates will be discussed shortly.

We focus on the two common axes, parent-child (“/”) and descendent-or-self (“//”) with name tests. Path expressions written using this subset of XPath can be transformed into regular expressions for which there exists an FSM that accepts the language described by the expression. In order to handle many queries, we construct a combined machine that satisfies the following: 1) It identifies the exact language defined by all path expressions together. 2) When an accepting state is reached, it outputs all queries that are accepted at this state. 3) Common prefixes of the path expressions are represented by a single FSM.

Fig. 1 shows an example of such a *Non-deterministic Finite Automaton* (NFA) representing eight queries. A circle denotes a state. Two concentric circles denote an accepting state, marked by the IDs of accepted queries. A directed edge represents a transition. The symbol on an edge represents the input that triggers the transition. A special symbol * matches any element. An edge marked by ϵ represents an ϵ -move (i.e., an empty input transition). Shaded circles represent states shared by queries. In the figure, note that the common prefixes of all the queries are shared. Note also that the NFA contains multiple accepting states, corresponding to the accepting states of individual queries.

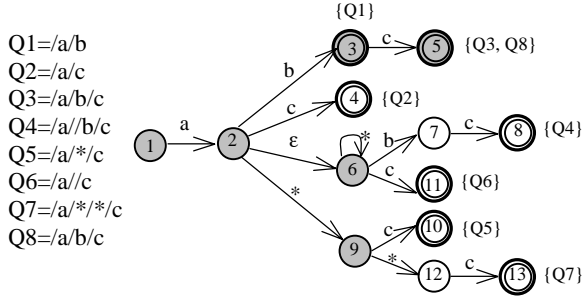


Fig. 1: XPath queries and a corresponding NFA

When an XML document arrives to be filtered, it is parsed with an event-based parser; each time a new element or the end of an element is encountered, an event is raised. The start-of-element events trigger transitions in the query FSMs. Since the machine is an NFA, many states can be active simultaneously. Unlike an NFA used to identify a regular language, the filtering of XML data requires that processing continue until all possible accepting states have been reached. So when an “end-of-element” event is raised the execution must backtrack to previous states. A run-time stack structure is used to track the active and previously processed states.

Fig. 2 shows the evolution of the contents of the stack as an example XML document is parsed. Each state in the stack is represented by its state ID, as shown in Fig. 1. On receiving a start-of-element event, the execution engine follows all matching transitions from all currently active states. For each active state, four checks are performed. First, if a transition marked by the incoming element name is present, the next state is added to the set of new active states. A transition marked by the “*” symbol is checked in the same way. Then, the state itself is added to the set if it has a self-loop, which is represented by an underlined state ID in Fig 2. Finally, if an ϵ -move is present, the state after the ϵ -move is processed immediately according to these same rules. The interested reader is referred to [4] for the details of the NFA-based processing of path expressions.

3 Selection

In an XPath expression, predicates can be applied to an element’s attributes, position or data. Predicates that do not reference other elements can be evaluated immediately when their related element is read from the document. Rather than modifying the NFA model to support these predicates, we use a special *Selection* operator that interacts with the NFA-based processing of path expressions in one of two ways. First, *Selection* can be performed for all predicates applied to an element right after the NFA execution makes transitions that are driven by the reading of this element from the document. This approach requires predicates to be stored with their corresponding states. Alternatively, selection can be

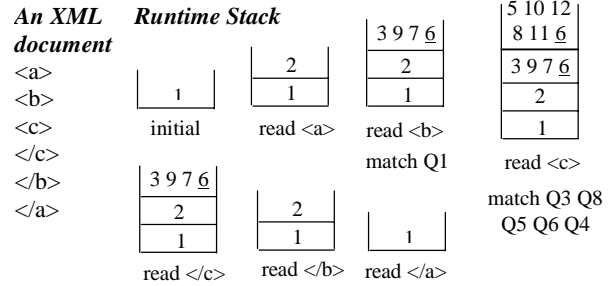


Fig. 2: An example of the NFA execution

performed for all predicates on a path expression when the expression is satisfied (i.e., when the NFA execution has reached an accepting state). For the latter approach predicates can be stored on a query-by-query basis.

4 Implementation and Demonstration

YFilter has been implemented in Java. It uses an event-based parser provided in the Xerces toolkit [2] supporting the SAX 1.0 interface for *event-based* XML parsing. The system works as follows: Queries are bulk loaded to build the NFA index and in-memory predicate tables. XML documents are then continuously generated and placed into a document queue representing streaming data. The filtering engine takes a document at a time from the queue, runs all queries on it, and reports all queries satisfied by the document. Between the processing of any two documents, the NFA index can be updated with new or deleted queries.

In the demonstration we will show the system in operation, with monitors depicting the system workload in terms of the number of queries, the number of predicates etc. as well as the performance of the system in terms of filtering time and the cost of online updates. We will also describe in more detail the structures and algorithms that are key to the efficiency and scalability of *YFilter*.

Acknowledgements

This work has been supported in part by the National Science Foundation under the ITR grant IIS00-86057, by DARPA under contract #N66001-99-2-8913, and by IBM, Microsoft, Siemens, and the UC MICRO program.

References

- [1] M. Altinel, M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *VLDB Conf.*, Sep. 2000.
- [2] Apache XML project. Xerces Java Parser 1.2.3 Release. <http://xml.apache.org/xerces-j/index.html>, 1999.
- [3] J. Chen et al. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *ACM SIGMOD Conf.*, May 2000.
- [4] Y. Diao, M. J. Franklin. NFA-based Filtering for Efficient and Scalable XML Routing. *Technical Report*, USB/CSD-1-1159, Oct. 2001.