Path sharing and predicate evaluation for high-performance XML filtering

Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, Peter M. Fischer

Angaben zur Veröffentlichung / Publication details:

Diao, Yanlei, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter M. Fischer. 2003. "Path sharing and predicate evaluation for high-performance XML filtering." *ACM Transactions on Database Systems* 28 (4): 467–516. https://doi.org/10.1145/958942.958947.

Path Sharing and Predicate Evaluation for High-Performance XML Filtering

YANLEI DIAO
University of California, Berkeley
MEHMET ALTINEL
IBM Almaden Research Center
MICHAEL J. FRANKLIN, HAO ZHANG
University of California, Berkeley
PETER M. FISCHER
University of Heidelberg

XML filtering systems aim to provide fast, on-the-fly matching of XML-encoded data to large numbers of query specifications containing constraints on both structure and content. It is now well accepted that approaches using event-based parsing and Finite State Machines (FSMs) can provide the basis for highly scalable structure-oriented XML filtering systems. The XFilter system [Altinel and Franklin 2000] was the first published FSM-based XML filtering approach. XFilter used a separate FSM per path query and a novel indexing mechanism to allow all of the FSMs to be executed simultaneously during the processing of a document. Building on the insights of the XFilter work, we describe a new method, called "YFilter" that combines all of the path queries into a single Nondeterministic Finite Automaton (NFA). YFilter exploits commonality among queries by merging common prefixes of the query paths such that they are processed at most once. The resulting shared processing provides tremendous improvements in structure matching performance but complicates the handling of value-based predicates.

In this paper we first describe the XFilter and YFilter approaches and present results of a detailed performance comparison of structure matching for these algorithms as well as a hybrid approach. The results show that the path sharing employed by YFilter can provide order-of-magnitude performance benefits. We then propose two alternative techniques for extending YFilter's shared structure matching with support for value-based predicates, and compare the performance of these two techniques. The results of this latter study demonstrate some key differences between shared XML filtering and traditional database query processing. Finally, we describe how the YFilter approach is extended to handle more complicated queries containing nested path expressions.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval --- Information filtering; H.3.4 [**Information Storage and Retrieval**]: Systems and Software --- Selective dissemination and information; Performance evaluation.
General Terms: Algorithms, Performance.
Additional Key Words and Phrases: XML filtering, structure matching, path sharing, Nondeterministic Finite Automaton, content-based matching, predicate evaluation, nested path expressions.

This work has been supported in part by the National Science Foundation under the ITR grants IIS0086057 and SI0122599, and CAREER grant IRI-9501353; by Rome Labs agreement no. F30602-97-2-0241 under DARPA order number F078, and by Boeing, IBM, Intel, Microsoft, Siemens, and the UC MICRO program.
Authors' addresses: Yanlei Diao, Michael J. Franklin, Hao Zhang, EECS, University of California, Berkeley, CA, 94720, email: {diaoyl, franklin, nhz}@cs.berkeley.edu; Mehmet Altinel, IBM Almaden Research Center, San Jose, CA, 95120, email: maltinel@us.ibm.com; Peter M. Fischer, Institute of Computer Science, University of Heidelberg, Heidelberg, Germany, email: peter.fischer@ informatik.uni-heidelberg.de.

1. INTRODUCTION

The emergence of XML as a common mark-up language for data interchange has spawned significant interest in techniques for filtering and content-based routing of XML data. In an XML filtering system, continuously arriving streams of XML documents are passed through a filtering engine that matches documents to queries and routes the matched documents accordingly. Queries in these systems are expressed in a language such as XPath 1.0 [Clark and DeRose 1999], which is used to specify constraints over both structure (using path expressions) and content (using value-based predicates).

In the past few years, there have been a number of efforts to build efficient large-scale XML filtering systems. While most of these systems support both structure and value matching to some extent, they have tended to emphasize either the processing of path expressions (e.g., *XFilter* [Altinel and Franklin 2000], *CQMC* [Ozen et al. 2001], *WebFilter* [Pereira et al. 2001], *XTrie* [Chan et al. 2002], *Index-Filter* [Bruno et al. 2003], and [Green et al. 2003]), or the processing of value-based predicates (e.g., *TriggerMan* [Hanson et. al 1999], *NiagaraCQ* [Chen et al. 2000], and *Le Subscribe* [Fabret et al. 2001]). One recent project, called *MatchMaker* [Lakshmanan and Parthasarathy 2002], has addressed both issues but has focused on disk-oriented solutions with performance characteristics that differ significantly from these other systems.

The work we present in this paper is motivated by the realization that efficient and scalable matching of path expressions lays the foundation for high-performance XML filtering. In particular, we show that *shared processing* for structure matching is a key technique for high-performance XML filtering. Given a shared path matching engine, a further challenge is the processing of predicates within such an engine. Thus, a second focus of our work is on the integration of predicate processing with shared structure matching.

1.1 Structure Matching

For structure matching, a useful approach has been to adopt some form of Finite State Machine (FSM) to represent path expressions in which location steps of path expressions are mapped to machine states. Arriving XML documents[1] are then parsed with an *event-based* parser; the events raised during parsing are used to drive the FSMs through their various transitions. A query is said to match a document if during parsing, an *accepting state* for that query is reached. This approach to XML filtering was first used in the XFilter system [Altinel and Franklin 2000][2].

In the filtering context, large numbers of queries representing the interests of the user community are stored and must be checked upon the arrival of a new document. In order to process these queries efficiently, XFilter employs a dynamic index over the states of

[1] In this paper we refer to incoming XML data items generically as "documents". Such documents, however, may also be XML-encoded messages or database tuples.

[2] An FSM-based approach for processing of individual queries (as opposed to filtering for large numbers of queries) was developed independently around this same time in the *XScan* system [Ives et al. 2000].

the query FSMs and includes optimizations that reduce the number of path expressions that must be checked for a given document.

In large-scale systems there is likely to be significant commonality among user interests, which could result in redundant processing in XFilter. *CQMC* [Ozen et al. 2001] improved upon XFilter by building an FSM for a set of queries identical in structure. *XTrie* [Chan et al. 2002] further supports shared processing of certain common sub-strings of the path expressions. In this paper, we present an approach called YFilter, that exploits sharing even more aggressively by using a single combined FSM to represent all path expressions. The combined FSM naturally supports the sharing of processing for all common prefixes among path expressions.

We implement this combined FSM as a *Nondeterministic Finite Automaton* (NFA). The NFA-based implementation has several practical advantages including: 1) a relatively small number of machine states required to represent even large numbers of path expressions, 2) the ability to support complicated document types (e.g., with recursive nesting) and queries (e.g., with multiple wildcards and ancestor-descendent axes), and 3) incremental construction and maintenance. The NFA-based shared path matching approach can result in order of magnitude performance improvements over the FSM-per-query approach of XFilter.

Query specifications can specify additional structural constraints by including *nested* path expressions on elements in location steps. These nested paths reference other elements in the same document. To support nested path processing, we extend YFilter using a technique we call *decomposition* that exploits the NFA-based path sharing.

1.2 Predicate Processing

Structure matching is only one part of the XML filtering problem. Query specifications can also include value-based predicates on the elements of path expressions. Such predicates are applied to address attributes or text data of those elements.

For value-based predicates, one could extend the NFA by including predicates as labels on additional transitions between states. Unfortunately, such an approach would result in a potentially huge increase in the number of states in the NFA, and would also destroy the sharing of path expressions, the primary advantage of using an NFA.

For this reason, we have investigated several alternative approaches to combining structure-based and value-based filtering in YFilter. Similar to traditional relational query processing, the placement of predicate evaluation in relation to the other aspects of query processing can have a major impact on performance. Relational systems use the heuristic of “pushing” cheap selections as far as possible down the query plan so that they are processed early in the execution. Following this intuition, we have developed an approach called “*Inline*”, that processes value-based predicates as soon as the relevant state is reached during structure matching. Similar approaches have been reported in [Ozen et al. 2001] to support predicate evaluation in the execution of individual FSMs, or proposed in [Chan et al. 2002] as an extension to its path matching algorithm. In addition, we have developed an alternative approach, called *Selection Postponed* (*SP*), that waits

until an accepting state is reached during structure matching, and only at that point, applies all the value-based predicates for the corresponding matched queries.

1.3 Contributions and Overview

The contributions of this paper include the following:

- We provide an overview of XFilter, an initial FSM-based XML filtering approach, which has become an important point of comparison for XML filtering systems (e.g. [Pereira et al. 2001], [Chan et al. 2002], [Green et al. 2003]).
- We describe a novel path matching approach, called YFilter, which shares processing work for multiple queries by building a single NFA over all path expressions.
- We present results of a detailed performance study that investigates the impact of shared path matching on performance and scalability using XFilter, YFilter, and a hybrid approach that does more sharing than XFilter but less than YFilter.
- We propose and evaluate two alternative methods for value-based predicate processing in YFilter: Inline and Selection Postponed (SP). The performance results show that contrary to intuition from traditional database systems, the delayed predicate processing of SP outperforms the eager processing of Inline by a wide margin. This study is, to our knowledge, the first study focused on alternative approaches to combined structure and value-based filtering.
- Finally, we describe how to use YFilter to handle nested path expressions in a way that allows them to also exploit shared processing.

The remainder of the paper is organized as follows. Section 2 provides background on XML filtering and sketches the basic system design. Section 3 describes the XFilter and YFilter algorithms and outlines a hybrid approach for path matching. Section 4 presents the results of a detailed performance comparison of these techniques. Section 5 describes the Inline and SP alternatives for value-based predicates and compares their performance. Section 6 presents our solution to nested path processing and evaluates its performance. Section 7 surveys related work. Section 8 presents our conclusions.

2. BACKGROUND

In this section we first present a high-level overview of an XML filtering system. We then describe a subset of the XPath language that we use to specify user interests in this work. Finally, we describe the main components of our solutions.

2.1 Overview of a Filtering System

A filtering system delivers documents to users based on their expressed interests. Figure 1 shows the context in which a filtering system operates. There are two main sets of inputs to the system: user profiles and document streams.

User profiles describe the information preferences of individual users. These profiles may be created by the users themselves, e.g., by choosing items in a Graphical User Interface, or may be created automatically by the system using machine learning

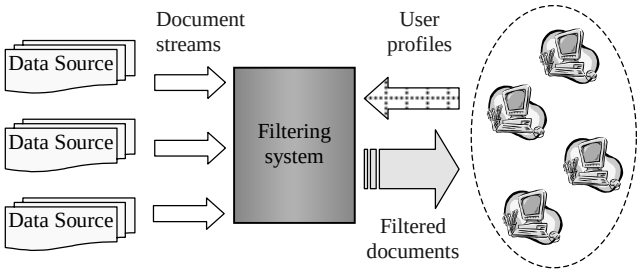


Figure 1: Overview of a filtering system

techniques. The user profiles are converted into a format that can be efficiently stored and evaluated by the filtering system. These profiles are effectively "standing queries", which are applied to all incoming documents. Hereafter, we will use profiles and queries interchangeably.

The other key inputs to a filtering system are the document streams containing continuously arriving documents from data sources. These documents are to be filtered and delivered to users or systems in a timely fashion. Filtering is performed by matching each arriving document against all of the user profiles to determine the set of interested users. The document is then delivered to this set of users. In our system, documents are processed one-at-a-time. That is, incoming documents are placed in a queue; a document is removed from the queue and processed in its entirety (i.e., matched with all relevant queries) before processing is initiated for the next document.

As filtering systems are deployed on the Internet, the number of users for such systems can easily grow into the millions. A key challenge in such an environment is to efficiently and quickly search the huge set of user profiles to find those for which the document is relevant. Our work is aimed at solving this very problem.

2.2 Data Encoding and Profile Language

We focus on filtering XML documents. XML is rapidly gaining popularity as a mechanism for sharing and delivering information among businesses, organizations, and users on the Internet. It is also achieving importance as a means for publishing commercial content such as news items and financial information.

XML provides a mechanism for tagging document content in order to better describe its organization. It allows the hierarchical organization of a document as a root element that includes sub-elements; elements can be nested to any depth. In addition to sub-elements, elements can contain data (e.g., text) and attributes. A general set of rules for a document's elements and attributes can be defined in a *Document Type Definition* (DTD). It is important to note, however, that the filtering techniques we describe in this paper do not require DTDs and do not exploit them if present.

We use a subset of the XPath 1.0 [Clark and DeRose 1999] for expressing user profiles as queries over XML documents. XPath is a language for addressing parts of an XML document that was designed for use by both the XSL Transformations (XSLT) [Clark 1999] and XPointer [DeRose et al. 1999] languages. XPath provides a flexible

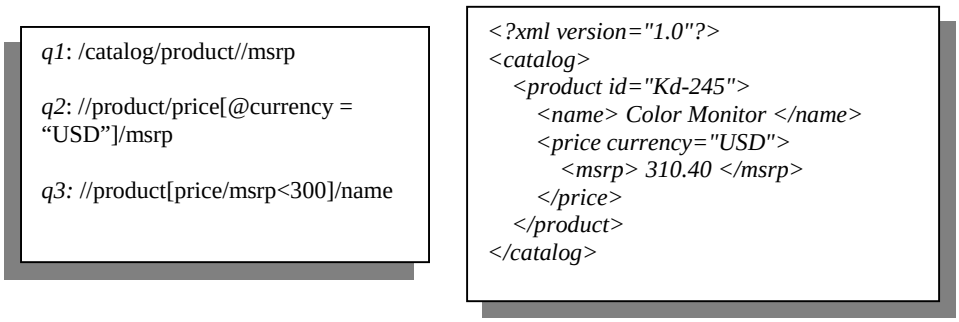


Figure 2: Three XPath queries and a sample XML document

way to specify path expressions. It treats an XML document as a tree of nodes; XPath expressions are patterns that can be matched to nodes in the XML tree. The evaluation of an XPath pattern yields an object whose type can be a node set (i.e., an unordered collection of nodes without duplicates), a boolean, a number, or a string.

Paths can be specified as absolute paths from the root of the document tree or as relative paths from a known location (i.e., the context node). A query path expression consists of a sequence of one or more *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 child operator "/" (i.e., nodes at adjacent levels), and the descendent operator "//" (i.e. nodes separated by any number of levels). In the simplest and most common form, a node test is specified by an element name. For example, query *q1* in Figure 2 addresses all *msrp* element descendants of all *product* elements that are children of a top-level *catalog* element in the document. XPath also allows the use of a wildcard operator ("*"), which matches any element name, as a name test at a location step.

Each location step can also include one or more predicates to further refine the selected set of nodes. A predicate, delimited by '[' and ']' symbols, is applied to the element addressed at a location step. All the predicates at a location step must evaluate to TRUE in order to fulfill the evaluation of the location step.

Predicates can be applied to the text data, the attributes, or the positions of the addressed elements. In this paper, we refer to such predicates as *value-based*. Query *q2* in Figure 2 gives an example of such a predicate. In addition, predicates may also include other path expressions, which are called *nested paths*. Relative nested paths are evaluated in the context of the element node addressed in the location step where they appear. For example, query *q3* in Figure 2 selects *name* elements of *product* elements with an *msrp* of less than 300. Here, the path expression *"price/msrp"* in the predicate is evaluated relative to the *product* elements.

In this work, we use XPath to select entire documents rather than parts of documents. That is, we treat an XPath expression as a predicate applied to documents. If the XPath expression matches at least one element of a document then we say that the document satisfies the expression. For the example in Figure 2, the sample XML document shown

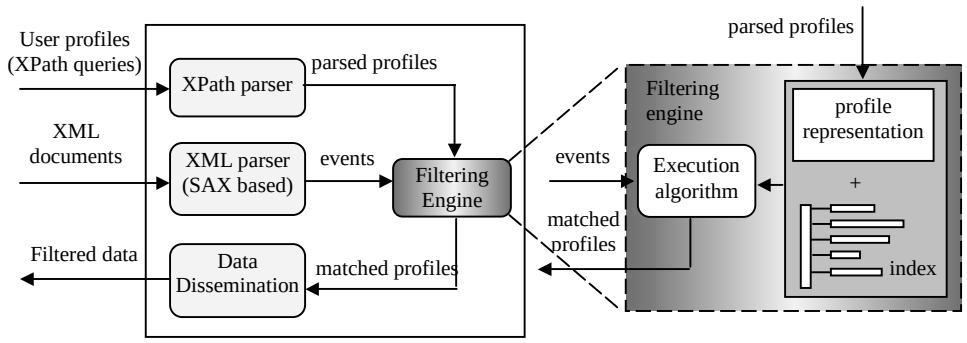


Figure 3: Architecture of a filtering system

satisfies the first two queries, but not the third. An alternative to using XPath would be to use an XML query language such as *XQuery* [Chamberlin et al. 2002]. We chose to use XPath in our work because we did not need the full functionality of such a query language for our document filtering purposes. Our ongoing work is investigating the use of YFilter for processing XQuery statements, but that work is beyond the scope of this paper.

2.3 FILTER SYSTEM COMPONENTS

The basic components of our XML Filtering systems are shown in Figure 3. They are:

XPath parser: The XPath parser takes queries written in XPath, parses them, and sends the parsed profiles to the filtering engine. New profiles can be added to a running filtering engine only when the engine is not actively engaged in processing a document.

Event-based XML parser: When an XML document arrives at the system, it is run through the XML parser. We use a parser based on the SAX interface, which is a standard interface for event-based XML parsing [Sax Project 2001]. Figure 4 shows an example of how a SAX event-based interface breaks down the structure of the sample XML document from Figure 2 into a linear sequence of events. "Start document" and "end document" events mark the beginning and the end of the parse of a document. A "start element" event carries information such as the name of the element, its attributes, etc. A "characters" event reports a string that is not inside any XML tag. An "end element" event corresponds to an earlier "start element" event by specifying the element name and marks the close of that element in the document.

To use the SAX interface, the system receiving the events must implement handlers to respond to different events. For our filtering systems, these events are used to drive the profile matching process. An additional advantage of using an event-based parser is that profile matching can start long before the parse of a document is complete. This is crucial to timely delivery if the document is long and parsing takes a fairly large amount of time.

Filtering engine: At the heart of the system, the filtering engine takes the parsed profiles from the XPath parser and converts them to an internal representation. It also receives and reacts to events from the XML parser. These events call back the

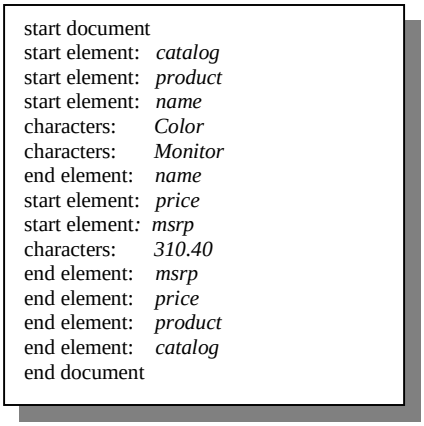


Figure 4: SAX API example

corresponding handlers implemented by the filtering engine, which perform the matching of documents against profiles.

Given the large number of profiles we wish to support, a brute force strategy of checking each profile will not scale. As has been pointed out in earlier work on information filtering (e.g., [Yan and Garcia-Molina 1994]), the key to scalability lies in the observation that filtering is the inverse problem of querying a database: In a traditional database system, a large set of data is stored persistently. Queries, coming one at a time, search the data for results. In a filtering system, a large set of queries is persistently stored. Documents, coming one at a time, drive the matching of the queries. In a traditional database, indexes enable the data to be searched without having to sequentially scan it. Likewise for filtering systems, indexing the queries can enable selective matching of incoming documents to queries.

The XFilter system extends this intuition to deal with the additional complexities introduced by hierarchically-structured documents and path-based queries over them. XFilter represents each path expression as a finite state machine (FSM). Events received from the parser naturally drive the transitions in those FSMs. XFilter uses a dynamic index structure on the machine states in order to reduce the number of FSMs (i.e., queries) that must be examined for each parsing event. Building on lessons gained from the study of XFilter, we have more recently developed a much more efficient approach, which we call YFilter. YFilter combines all of the queries into a single, nondeterministic finite automaton in order to allow the sharing of processing for query paths with common prefixes. The implementation of YFilter essentially consists of a tree of hash indexes, thus directly integrating the necessary index into the filtering structure. XFilter and YFilter are described in detail in the following section.

Dissemination component: Once the matching profiles have been identified for a document, the document must be sent to the appropriate users. Our current implementations simply use unicast delivery to send the entire document to each interested user. Ongoing work involves the integration of a variety of delivery mechanisms (e.g., those described in [Aksoy et al. 1998; Altinel et al. 1999]), and the

delivery of partial documents. These issues are beyond the scope of this current paper, so they are not addressed further here.

3. PROCESSING PATH EXPRESSIONS

In this section we describe FSM-based approaches to XML filtering, focusing on the way they achieve structure-based matching for large numbers of path expressions. We begin with XFilter, our initial approach based on indexing multiple FSMs. We then describe YFilter, which shares work for path expressions by combining them into a single NFA-based machine. At the end of this section, we outline a hybrid approach as a middle point between XFilter and YFilter with respect to the amount of sharing exploited. This hybrid approach is used in our study to help quantify the performance impact of improved shared path matching.

3.1 XFilter: Filtering Using Indexed FSMs

Filtering XML documents using a structure-oriented path language such as XPath raises several problems: (1) Checking the sequencing of the elements in a path expression; (2) Handling wildcards and descendant operators in path expressions; and (3) Evaluating nested paths that are applied to element nodes. Here we focus on the first two problems, and postpone the discussion of our solution to handling nested paths until Section 6, because it shares a common mechanism with one of our approaches to value-based predicate evaluation.

XFilter's solution to the first two problems is based on the observation that any single path expression written using the axes ("/", "//") and node tests (element name or "*") 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 [Hopcroft and Ullman 1979]. XFilter converts each XPath query to a *Finite State Machine* (FSM). The events that drive the execution of the filtering engine are generated by the XML parser. In the XFilter execution model, a profile is considered to match a document when the final state of its FSM is reached. In this section, we present an overview of XFilter. More details can be found in [Altinel and Franklin 2000].

3.1.1 Internal Profile Representation

The main structures for profile representation in XFilter are depicted in Figure 5(a). Each XPath query is decomposed into a set of *path nodes* by the XPath parser. These path nodes represent the element nodes in the query and serve as the states of the FSM for the query. Path nodes are not generated for wildcard ("*") nodes. A path node contains the following information:

QueryId: A unique identifier for the path expression to which this path node belongs (generated by the XPath Parser).

Position: A sequence number that determines the location of this path node in the order of the path nodes for the query. The first node of the path is given position 1, and the following nodes are numbered sequentially.

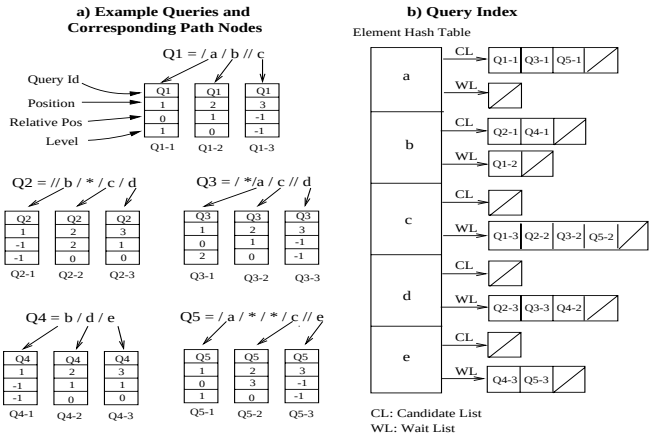


Figure 5: Path node decomposition and the corresponding *QueryIndex*

RelativePos: An integer that describes the distance in *document levels* between this path node and the previous (in terms of position) path node. This value is set to 0 for the first node if it does not contain a descendant ("//") operator. A node that is separated from the previous one by such an operator is flagged with a special RelativePos value of -1. Otherwise, the RelativePos value of a node is set to 1 plus the number of wildcard nodes between it and its predecessor node.

Level: An integer that represents the level in the XML document at which this path node should be checked. Because XML does not restrict element types from appearing at multiple levels of a document and because XPath allows queries to be specified using "relative" addressing, it is not always possible to assign this value during query parsing. Thus, unlike the previous three items, this information can be updated during the *evaluation* of the query.

The level value is initialized as follows: If the node is the first node of the query and it specifies an absolute distance from the root (i.e., it is either applied to the root node or is a fixed number of wildcard nodes away from the root node), then the level for that node is set to 1 plus its distance from the root. If the RelativePos value of the node is -1, then its level value is also initialized to -1. Otherwise, the level value is set to 0.

NextPathNodeSet: Each path node also contains a pointer to the next path node of the query to be evaluated.

3.1.2 Index Construction

To achieve high performance for structure filtering, XFilter contains an *inverted index* [Salton 89], called the *Query Index*, on all the XPath queries. It is used to efficiently match a document to individual queries. The Query Index is built over the states of the query FSMs. As shown in Figure 5(b), the Query Index is organized as a hash table based on the element names that appear in the XPath expressions. Associated with each unique element name are two lists of path nodes: the *candidate list* and *wait list*.

Candidate lists identify the path nodes corresponding to the states that the FSM execution is attempting to match at a particular moment. Wait lists contain path nodes

that are subsequent to the nodes in the candidate lists. The contents of the candidate lists constantly change as parsing events drive the execution of the FSMs.

The initial distribution of the path nodes to the lists (i.e., which node of each XPath query is initially placed on a candidate list) is an important contributor to the performance of the XFilter system. We have developed two such placement techniques. Figure 5(b) shows the most straightforward case, where the path nodes for the initial states are placed on the Candidate Lists. For many situations, however, such an approach can be inefficient, as the first elements in the queries are likely to have poorer selectivity due to the fact that they address elements at higher levels in the documents where the sets of possible element names are smaller. In the resulting Query Index, the lengths of the candidate lists would become highly skewed, with a small number of very long candidate lists that do not provide much selectivity. Such skew hurts performance as the work that is done on the long lists may not adequately reduce the number of queries that must be considered further.

Based on the above observation, we developed the *List Balance* method for choosing a path node to initially place in a candidate list for each query. This simple method attempts to balance the initial lengths of the candidate lists. When adding a new query to the index the element node of that query whose entry in the index has the shortest candidate list is chosen as the "pivot" node for the query. This pivot node is then placed on its corresponding candidate list, making it the first node to be checked for that query for any document.

This approach, in effect, modifies the FSM of the query so that its initial state is the pivot node. We accomplish this by representing the portion of the FSM that precedes the pivot node as a "prefix" that is attached to that node. When the pivot node is activated, the prefix of the query is checked as a precondition in the evaluation of the path node. If this precondition fails, the execution stops for that path node. In order to handle prefix evaluation, List Balance uses a stack that keeps track of the traversed element nodes in the document. We use this stack for fast forward execution of the portion of FSM corresponding to the prefix.

Figure 6 shows example path nodes and a modified Query Index for the List Balance algorithm. Notice that the lengths of the candidate lists are the same for each entry of the Query Index. The tradeoff of this approach is the additional work of checking prefixes for the pivot nodes when activated. As experimental results in [Altinel and Franklin 2000] show, this additional cost is far outweighed by the benefits of List Balance.

3.1.3 Execution algorithm

When a document arrives at the filtering engine, it is run through an XML Parser that reports parsing events to the application through callbacks, and the events are used to drive the profile matching process. For XFilter, we implemented the following two callback functions for the parsing events. Both of the handlers are passed the name and document level of the element for (or in) which the parsing event occurred. Additional handler-specific information is also passed as described below.

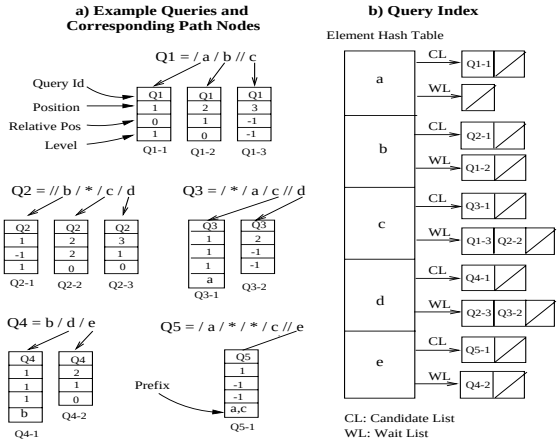


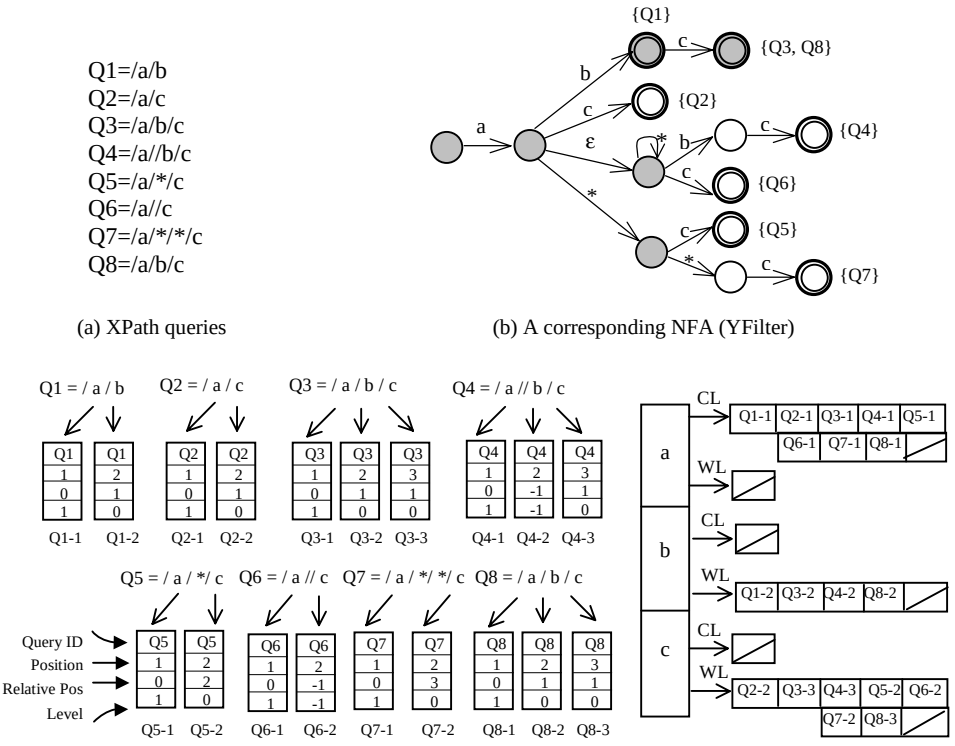
Figure 6: Path nodes and *QueryIndex* in List Balance

Start Element Handler: An start element event calls this handler, passing in the name and level of the element encountered as well as any XML attributes and values that appear in the element tag. The handler looks up the element name in the Query Index and examines all the nodes in the candidate list for that entry. For each node, it performs a level check. The purpose of the level check is to make sure that the element appears in the document at a level that matches the level expected by the query. If the path node contains a non-negative level value, then the two levels must be identical in order for the check to succeed. Otherwise, the level for the node is unrestricted, so the check succeeds regardless of the element level.

If the check succeeds, then the node passes. If this is the final path node of the query (i.e., its final state) then the document is deemed to match the query. Otherwise, if it is not the final node, then the query is moved into its next state. This is done by *copying* the next node for the query from its wait list to its corresponding candidate list (note that a copy of the promoted node remains in the wait list). If the *RelativePos* value of the copied node is not -1, its level value is also updated using the current level and its *RelativePos* values to do future level checks correctly.

Note that in the most basic case, there is only one copy of a path node in its candidate list during the evaluation of a query. However, when the same element name appears in a nested manner at different levels of the input document and a path node related to this element name corresponds to a "//" location step, matching of the nested elements with this path node will cause multiple promotions of its subsequent path node. In such cases, multiple copies of the subsequent path node can exist in its corresponding candidate list to reflect different document levels where it can be matched.

End Element Handler: When an end element tag is encountered, the path nodes promoted when the corresponding start element tag was encountered are deleted from the candidate lists in order to restore those lists to the state before reading this element. This "backtracking" is necessary to handle the case where multiple elements with the same name appear at different levels in the document.



(a) XPath queries

(b) A corresponding NFA (YFilter)

(c) Path nodes of the queries and the Index (XFilter)

Figure 7: XPath queries and their representation in YFilter and XFilter

3.2 YFilter: An NFA-based Approach Exploiting Path Sharing

XFilter demonstrates how by using an index on profiles, one scan of a document can be used to drive the simultaneous execution of all candidate profiles. As stated previously, however, XFilter fails to exploit commonality that exists among the path expressions (recall that XFilter builds an FSM for each query). For large-scale filtering of XML data, exploiting such commonality can greatly reduce redundant processing. Thus, we propose a new filtering approach that follows the event-driven philosophy of XFilter, but in addition, shares processing among path expressions to eliminate redundant work.

3.2.1 Profile Representation: A Combined NFA with an Output Function

In the new model, rather than representing each query as an FSM individually, we combine all queries into a single *Nondeterministic Finite Automaton* (NFA). The labels of the transitions of this machine form a *trie* over the location steps of the path expressions. As such, the common prefixes of the paths are represented only once in the structure. In addition, the machine employs a stack-based mechanism to cope with non-determinism and support backtracking in the structure.

Figure 7(b) shows an example of such an NFA representing the eight queries shown in Figure 7(a) (for comparison purposes, Figure 7(c) shows the eight FSMs and the query index that would be used in basic XFilter). 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 this point in the discussion, we are not considering predicates).

This NFA can be formally defined as a *Moore Machine* [Hopcroft and Ullman 1979]. 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.

Some Comments on Efficiency. A key benefit of using an NFA-based implementation of the combined FSM is the tremendous reduction in machine size it affords. Of course, it is reasonable to be concerned that using an NFA 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 [Hopcroft and Ullman 1979]. A straightforward conversion could theoretically result in severe scalability problems due to an explosion in the number of states. But, as pointed out in [Green et al. 2003], this explosion can be avoided in many cases by placing restrictions on the types of documents (i.e., DTDs) and queries supported, and lazily constructing the DFA.

Our experimental results (described in Section 4), 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 typically not the dominant cost of filtering. Rather, other costs such as document parsing are in many cases more expensive than the basic path matching, particularly for systems with large numbers of similar queries. Thus, while it may in fact be possible to further improve path matching speed, we believe that the substantial benefits of expressiveness and incremental maintenance provided by the NFA model outweigh any marginal performance improvements that remain to be gained by even faster path matching.

3.2.2 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 7 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 8 shows the directed graphs, called *NFA fragments*, that correspond to these basic location steps.

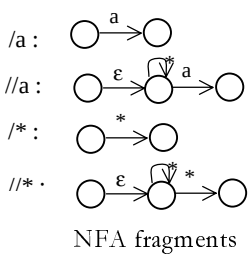


Figure 8: NFA fragments of basic location steps

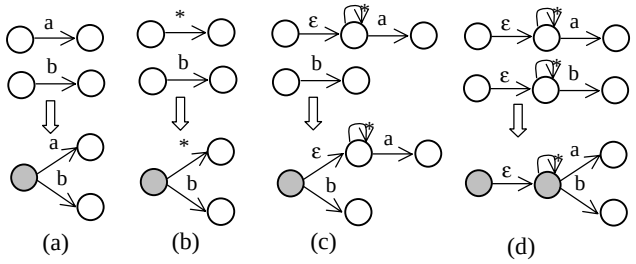


Figure 9: Combining NFA Fragments

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 9 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 NFA_p. Figure 9 provides four examples of this process.

Figure 9(a) shows the process of merging a fragment for location step "/a" with a state in the combined NFA that represents a "/b" step. We do not combine the edge marked by "a" and the edge marked by "b" into one marked by "a,b" as in a standard NFA, because the states after edge 'a' and edge 'b' differ in their outputs, so they cannot be combined. For the same reason, this process treats the "*" symbol in the way that it treats the other symbols in the alphabet, as shown in Figure 9(b).

Figure 9(c) shows the process of merging a "//a" step with a "/b" step, while Figure 9(d) shows the merging of a "//a" step with a "//b" step. Here we see why we need the ε-transition in the NFA fragment for "//a". Without it, when we combine the fragment with the NFA fragment for "/b", the latter would be semantically changed to "//b". The merging process for "//*" with other fragments (not shown) is analogous to that for "//a".

The "*" and "//" operators introduce *Non-determinism* into the model. "*" requires two edges, one marked by the input symbol and the other by "*", to be followed. The descendent operator "//" means the associated node test can be satisfied at any level at or below the current document level. In the corresponding NFA model, if a matching

symbol is read at the state with a self-loop, the processing must both transition to the next state, and remain in the current state awaiting further input.

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.

3.2.3 Implementing the NFA Structure

The previous section described the logical construction of the NFA model. For efficient execution we implement the NFA using a "hash table-based" approach, which has been shown to have low time complexity for inserting/deleting states, inserting/deleting transitions, and actually performing the transitions [Watson 1997].

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 8). 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.

3.2.4 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 callback the handlers and 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 handlers.

Start Document Handler: 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 Element Handler: 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) First, 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".

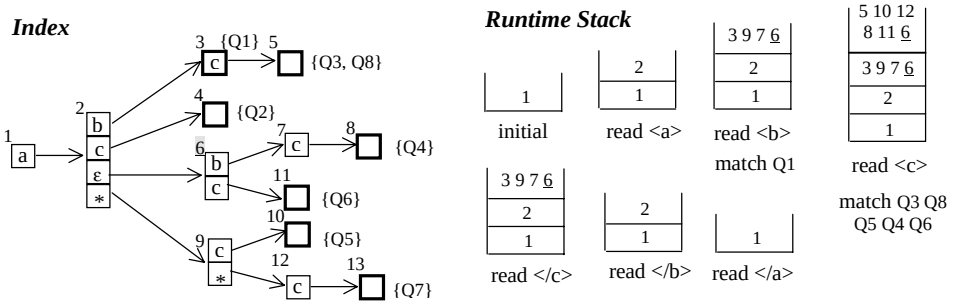


Figure 10: An example of NFA execution

2) Second, the "*" symbol is 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 the three rules above.[3]

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 the identifiers of all queries associated with the state are collected and added to an output data structure.[4]

End Element Handler: 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 find all matching queries. Thus, even after an accepting state has been reached for a document, the execution must continue until the document has been completely processed.

An example of this execution model is shown in Figure 10. On the left of the figure is the index created for the NFA of Figure 7. 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 A Hybrid Approach

[3] Note that this process traverses at most one additional level, since //-child nodes cannot themselves contain an "ε" symbol.
[4] If predicate processing is not needed, we can also mark the accepting state as "visited" to avoid processing matched queries more than once.

We have developed a third approach, called “Hybrid”, as a compromise between the XFilter and YFilter approaches. Hybrid is an improved version of XFilter that exploits some path sharing, but not as much as YFilter. In Hybrid, queries are decomposed into substrings containing only “/” operators (i.e., they are split at “*” and “//” operators). The processing of these substrings is shared, but the processing of the operators between these substrings is done individually for each query.

Hybrid works as follows. First, each query is parsed into a list containing one node for each substring of the query. Each node contains five data items (QueryId, NodePosition, RelativePos, Level, and NextNodePointer), as path nodes in XFilter. The difference is that RelativePos here specifies the distance in document levels from the end of the previous substring to the end of this substring. Then, the substrings of all of the queries are inserted into a single *Trie* index. Inside the index, a candidate list is allocated in each index node that represents the end (i.e., the last element) of a substring. Similar to XFilter, a candidate list here contains nodes representing those substrings that the current execution attempts to match. Initially, candidate lists only contain the nodes for the first substrings of queries.

During the execution, input elements drive the navigation in the *trie* index as in YFilter, but without any concern for “*” and “//” operators. Each input element initiates a search from the root of the *trie* and also continues searches from index nodes that the navigation reached on the previous input element. As in YFilter, a runtime stack is used for maintaining the list of index nodes representing the current state and for backtracking. When an index node with a non-empty candidate list is encountered, all substring nodes in the list undergo a document level check. For each of those substrings that pass the level check, the expected level of the end of the next substring in the query is updated in the node for the next substring, and that substring node is copied to its corresponding candidate list. In this way, the matching of a substring in the *trie* index is shared by all queries containing this substring, but the transitions between two substrings are done on a query-by-query basis using document level checking as in XFilter.

Independently of our work, Chan et al. developed several algorithms for XML filtering under the name “XTrie” [Chan et al. 2002]. XTrie uses a “minimal decomposition” of queries that is identical to the decomposition we use in Hybrid. Furthermore, Hybrid’s execution model is similar in spirit to the “eager TRIE” version of XTrie in that matching of substrings is shared among queries and transitions between substrings are handled on an individual query basis. It is worth noting that “eager TRIE” is not the best performing approach studied by Chan et al. Other optimizations, orthogonal to the issue of sharing, have been developed in that work.

4. PERFORMANCE OF STRUCTURE MATCHING

In this section, we examine the performance of path matching in the absence of predicate evaluation. Recall that our development of YFilter was motivated by the desire to share processing during path evaluation. As such, the focus of this performance study is on the impact of such shared processing.

4.1 Algorithms

We compare the performance of XFilter, YFilter, and Hybrid. In the experiments that follow we use the variant of XFilter with list balance, which was shown to provide better performance overall than the basic XFilter approach [Altinel and Franklin 2000]. Also, for both XFilter and Hybrid, we use a simple optimization that is important in some of our workloads, namely, that *identical* queries are represented in the system only once. We did this by pre-processing the queries and collecting the IDs of identical queries in an auxiliary data structure. This structure is the same as that used by YFilter to manage query IDs in accepting states. YFilter, of course, does not require such an optimization as it naturally shares processing of identical queries.

Despite the similarity between Hybrid and XTrie [Chan et al. 2002], we do not claim to do a direct comparison with that work. However, Chan et al. did compare their approaches to XFilter with list balance, so as discussed in the next section, it is possible to gain some insight into the relative performance of our techniques and the variants of XTrie.

4.2 Experimental Set-up

We implemented the three algorithms (YFilter, XFilter with list balance, and Hybrid) using Java. All of the experiments reported here were performed on a Pentium III 850 Mhz processor with 384MB memory running JVM 1.3.0 in server mode on Linux 2.4. We set the JVM maximum allocation pool to 250MB, so that virtual memory and other I/O-activity had no influence on the results. This was also verified using the Linux *vmstat*() command.

4.2.1 Workload Generation

While, as stated previously, the three matching algorithms do not require or exploit DTD information, we do use DTDs to generate the workloads for our experiments. In this section, we focus on workloads generated using the NITF (News Industry Text Format) DTD [Cover 1999] used in previous studies [Altinel and Franklin 2000; Chan et al. 2002]. We also ran experiments using two other DTDs: The Xmark-Auction DTD [Busse et al. 2001] from the Xmark benchmark, and the DBLP [Ley 2001] bibliography DTD. Some characteristics of these DTDs are shown in Table 1. Note that all of the DTDs allow an infinite level of nesting due to loops involving multiple elements.

	NITF	Auction	DBLP
number of elements names	123	77	36
number of attributes in total	510	16	14
maximum level of nesting allowed	infinite	infinite	infinite

Table 1: Characteristics of three DTDs

Given a DTD, the tools used to run an experiment include a *DTD parser*, a *query generator*, an *XML generator*, and an *event-based XML parser* supporting the SAX interface [Sax Project 2001]. The DTD parser which was developed using a WUTKA DTD parser [Wutka 2000] outputs parent-child relationships between elements, and statistical information for each element including the probability of an attribute occurring

in an element (randomly chosen between 0 and 1) and the maximum number of values an element or an attribute can take (randomly chosen between 1 and 20). The output of the DTD parser is used by the query generator and the document generator.

We wrote a query generator that creates a set of XPath queries based on the workload parameters listed in Table 2. The query generator generates random query strings according to the input DTD and these parameters. In order to remove some semantic redundancy that may be introduced by this random approach, it performs a simple rewriting step in which the following rules are applied in the presented order: 1) For each occurrence of "//*" in a query, turn it into "/*//"; 2) If a query contains multiple consecutive "/*//" substrings, only keep the first one; and 3) If "/*//" occurs at the end of a query, remove "//".

Parameter	***Range***	***Description***
Q	1000 to 500000	Number of queries
D	6 to 10	Maximum depth of XML documents and XPath queries.
W	0 to 1	Probability of a wildcard "*" occurring at a location step
DS	0 to 1	Probability of "//" being the operator at a location step
Distinct	True or False	Query strings required to be unique?
P	0 to 20	Number of predicates per query
NP	0 to 3	Number of nested paths per query
RP	2, 3, 5	Max. no. of repeats of an element under a single parent

Table 2: Workload parameters for query and document generation

The query generator can be set to create workloads with or without duplicate queries. We refer to this later mode as the *distinct* mode. If duplicates are allowed, the generator is simply invoked Q times. Otherwise, in distinct mode the query generator is invoked repeatedly until Q syntactically unique queries are produced. Of course, in such a *distinct* workload there may be significant overlap in the query strings but no two strings will be identical. Note that in most of the experiments reported in this study, the query generator is used in the distinct mode.

For document generation, we used IBM's XML Generator [Diaz and Lovell 1999] to generate the document structure. Two parameters are passed to the generator: maximum depth D, and RP, which specifies the maximum number of times that an element can be repeated under a single parent. As a default, RP is limited to 3. Then attributes of elements were generated according to their probabilities of occurring. The value of an element or an occurring attribute was randomly chosen between 1 and the maximum number of values it can take.

For each DTD we generated a set of 200 XML documents. All reported experimental results are averaged over this set. For each experiment, a set of queries was generated according to the workload setting. For each algorithm run in an experiment, queries were preprocessed, if necessary, and then bulk loaded to build the index and other data structures. Then XML documents were read from disk one after another. The execution for a document returned a bit set, each bit of which indicates whether or not the corresponding query has been satisfied. We began a new process for each experiment run

of an algorithm (i.e., 200 documents), to avoid complications from Java's garbage collector.

4.2.2 Metrics

Previous work [Altinel and Franklin 2000; Chan et al. 2002] used "filtering time" as the primary performance metric, which is the total time to process a document including parsing and outputting results. Noting that Java parsers have varying parsing costs, we instead report on a slightly different performance metric we call "multi-query processing time (MQPT)". MQPT captures all costs attributable to the filtering algorithms themselves. It is simply the filtering time minus the document parsing time. That is:

Multi-query processing time (***MQPT***) = Filtering time – Document parsing time

Filtering time = Wall clock time from the start of document parsing to the end of output

MQPT for path matching consists of two components: path navigation and result collection. The former captures the cost of state transitions driven by received events. The latter is the cost to collect the identifiers of queries from the auxiliary data structures and to mark them in the result bit set. Note that when only distinct queries are used in experiments, the cost of result collection is negligible.

Where appropriate, we also report on other metrics such as the number of transitions followed, the size of the various machines, and the costs associated with maintenance, etc.

4.3 Efficiency and Scalability

Having described our experimental environment, we begin our discussion of experimental results by presenting the MQPT results for the three alternatives as the number of queries in the system is increased.

4.3.1 Experiment 1: NITF

In this experiment we generated 200 XML documents using the NITF DTD under the workload (D = 6, RP = 3). The average length of generated documents is 77 in terms of start-end element pairs. The average level of nesting of elements is 5.45.

We first examine the MQPT for the three algorithms as the number of *distinct* queries in the system is increased from 1000 to 150,000 with the probability of "*" and "//" operators each set to 0.2. With this setting, each query contains approximately one "*" operator and one "//" operator. Recall that in this section we are studying structure matching only, so there are no predicates on the elements. Predicate processing is studied subsequently, in Section 5.

The results are shown in Figure 11. As can be seen in the figure, YFilter provides the significantly better performance than the other two across the entire range of query populations. XFilter is the slowest here by far, and not surprisingly, Hybrid's performance lies between the two.

As the number of queries increases, YFilter exhibits a slight cost increase and levels off around 30ms when Q is larger than 50,000. In contrast, the processing cost of XFilter increases dramatically, to 732ms at 100,000 and runs out of memory after this point, while Hybrid takes 344ms at this point. Thus YFilter exhibits an order of magnitude improvement for path matching over these other schemes.[5]

The performance benefits of YFilter come from two factors. The first is the benefit of shared work obtained by the NFA approach. YFilter is the most effective of the three at exploiting commonality among similar, but not exactly identical queries, as it can share all common prefixes of the paths. The second factor is the relatively low cost of state transitions in YFilter compared to the others, which results from the hash-based implementation described in Section 3.2. We verified this by comparing the improvement ratio of YFilter over XFilter in terms of path navigation time with that in terms of the number of transitions. For example, when Q is 100,000, XFilter makes 7.4 times more transitions but takes 25.2 times longer to navigate.

The experiment just described, like other XML filtering studies [Altinel and Franklin 2000; Chan et al. 2002; Lakshmanan and Parthasarathy 2002; Green et al. 2003] did not address the effect of duplicate path queries on the query processing time. Duplicate paths, however, are likely to exist in a large filtering system. For this reason, we re-ran the previous experiment with the query generator set to not remove duplicates. Figure 12 shows the MQPT of three algorithms as the number queries in the system is varied from 1,000 to 500,000.

Compared to Figure 11, YFilter still achieves a significant performance improvement over Hybrid and XFilter, but the differences among the algorithms are not as great. In particular, XFilter and Hybird seem to scale better, and the cost of YFilter increases.

We measured the number of distinct queries among random queries and report them in Table 3. It shows the relatively slow increase in the number of distinct queries. Since all three algorithms represent identical queries only once, they all benefit from the slow increase, which explains the improved MQPT of Hybrid and XFilter.

Number of random queries (x1000)	1	100	200	300	400	500
Number of distinct queries (x1000)	0.53	15.7	24.2	30.5	35.6	40.0

Table 3: Number of distinct queries out of randomly generated queries
(NITF, D=6, W=0.2, DS=0.2)

We further decompose the MQPT into two component costs: path navigation and result collection. Results are shown in Figure 13. For each data point, the bars represent from left to right: YFilter, Hybrid, and XFilter. The cost of path navigation at each data point is consistent with that for the same number of distinct queries in Figure 11. The cost of result collection, however, becomes significant. Even though we coded result collection carefully, e.g. using unsynchronized data structures and avoiding ID instance

[5] Note that the performance of XTrie was also compared with that of XFilter [Chan et al. 2002] for a similar workload. The fastest algorithm studied there, called Lazy Trie, was shown to have only about a 4x improvement over XFilter.

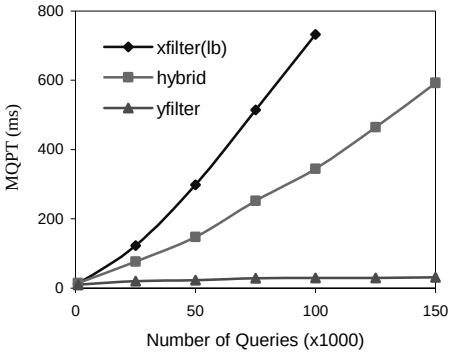


Figure 11: Varying number of distinct queries (NITF, D=6, W=0.2, DS=0.2)

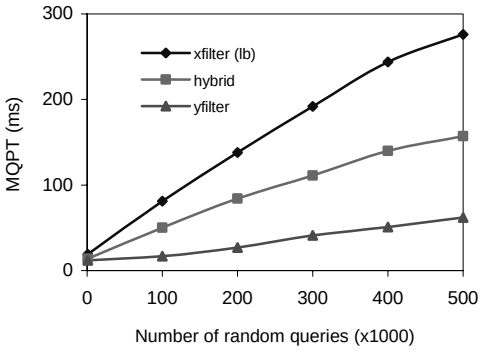
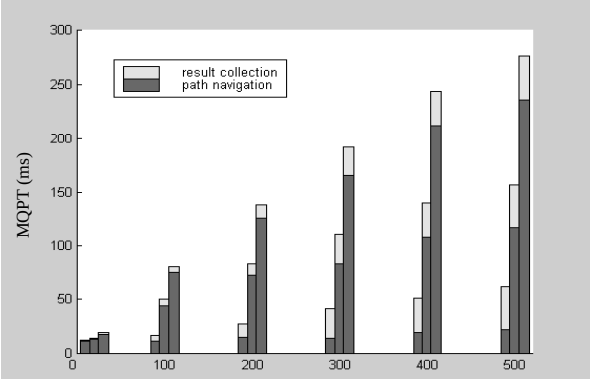


Figure 12: Varying number of queries (with duplicates) (NITF, D=6, W=0.2, DS=0.2)



No. of random queries (x1000) (bars left to right: yfilter, hybrid, xfilter(lb))

Figure 13: Component costs for processing queries containing duplicates (NITF, D=6, W=0.2, DS=0.2)

copies, its cost is still high in this experiment, because a high percentage of path expressions match each document (34% here for each value of Q compared to less than 10% for most values of Q in the previous experiment using distinct queries). Note that in Figure 13, the MQPT of YFilter is dominated by the cost of result collection starting from the point of Q=300,000. At this point, the number of query IDs collected is 9.3 times the number of state transitions YFilter makes.

The above results for duplicate path queries indicate that experiments using distinct paths may tend to magnify the differences among filtering algorithms in scenarios where duplicate queries are likely. To exhibit a significant performance improvement in practical workloads containing duplicate queries, a filtering algorithm needs to outperform others by a wide margin, as YFilter outperforms Hybrid and XFilter.

Similarly, for both the distinct and random workloads, document parsing is another fixed overhead that contributes to overall filtering time (recall that parsing is not included in MQPT). For example, the Xerces [Apache 1999] parser we used, set in a non-validating mode, took 168ms on the average to parse a document, completely dominating the NFA-based execution in both cases. We also tried other publicly available java

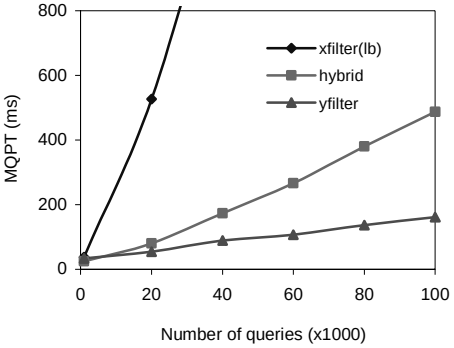


Figure 14: Varying number of distinct queries (Auction, D=8, W=0.2, DS=0.2)

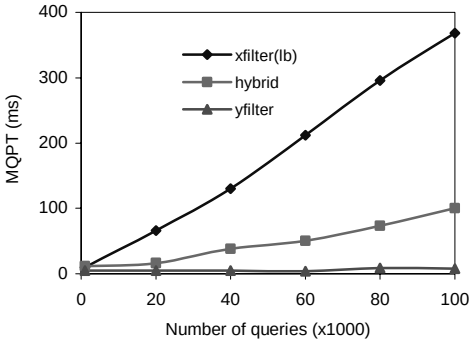


Figure 15: Varying number of distinct queries (DBLP, D=8, W=0.2, DS=0.2)

parsers including Java XML Pack [Sun Microsystems 2001] and Saxon XSLT processor [Kay 2001] supporting SAX 2.0. Saxon gave the best performance at 81 ms, still substantially more than the NFA navigation cost.[6] Thus, while we do not claim that YFilter is the fastest possible path matching approach, it is clear that its performance for both these workloads is sufficiently fast that any further improvements in path navigation time will have at best, a minor impact on overall performance.

4.3.2 Experiment 2: Other DTDs

We also ran experiments using two other DTDs: DBLP and Xmark-Auction. For these two DTDs, we set the maximum depth D to 8 in order to generate a relatively large set of distinct queries. The setting of W and DS is the same as the previous experiments and we report results only for the *distinct* query workload. Due to their DTD structures, DBLP tends to generate very short documents, while Xmark-auction tends to produce very long ones. We adjusted the RP parameter to control the document lengths for our experiments. For DBLP, RP was set to 5 and the generated documents contain on average, 16 start and end elements pairs. For Auction, we set RP to 2, obtaining an average document length of 175. The results of these experiments are similar to those obtained using the NITF workload. Space precludes us from describing these results in detail, so we summarize them here.

Figure 14 shows the MQPT results for the Xmark-Auction workload as Q is varied from 1,000 to 100,000. As can be seen in the figure, the trends observed using NITF are also seen here: YFilter performs substantially better than XFilter and Hybrid is in between the others. Since documents here are 2.3 times as long as those of NITF, all algorithms take longer to filter the documents. XFilter, however, is particularly sensitive to the length of documents because its FSM representation and execution algorithm result in significant memory management overhead, which in turn invokes garbage collection much more frequently.

[6] We have also experimented with C++ parsers, which are much faster, but even with these parsers we would expect parsing time to be at best similar to the cost of path navigation with YFilter, particularly if YFilter were also implemented in C++!

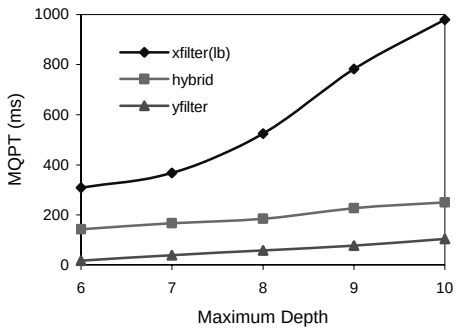


Figure 16: Varying maximum depth (NITF, Q=50,000, W=0.2, DS=0.2)

When the DBLP DTD is used, all algorithms run much faster, as shown in Figure 15. However, even though the documents used here are very short, YFilter still achieves substantial performance improvement over XFilter (e.g., 46 times at Q=100,000).

4.4 Experiment 3: Varying the maximum depth

In this experiment, we examine the impact of document depth on the performance of the three algorithms. Of particular concern is the performance of YFilter, since deep documents could theoretically cause an exponential blow-up in the number of active states for NFA execution. We used the NITF DTD in all the following experiments. The maximum depth was increased from 6 to 10.[7] For each D value, we generated 50,000 distinct queries.

As can be seen in Figure 16, the MQPT for all algorithms increases with the document depth, but YFilter remains the fastest. More importantly, the increase for YFilter is linear. To provide a better understanding these behaviours, we report the statistics on documents and queries used in this experiment in Table 4. Note that the average document depth (i.e., the average depth of all paths in each document) and query depths do not increase as quickly as D. This is because the DTD dictates that many paths cannot reach a very deep level. As the maximum repeat RP was fixed to 3 in this experiment, a larger value of D also caused longer documents (i.e., more start/end element pairs) to be generated.

Maximum Depth D	6	7	8	9	10
Avg. Document depth	5.45	6.06	6.68	7.28	7.69
Avg. Query depth	5.05	5.70	6.09	6.35	6.53
Avg. Document length	77	107	154	221	271

Table 4: Characteristics of documents and queries as maximum depth varies

[7] Note that we stopped increasing D at 10, because we expect that in large scale XML filtering scenarios, documents even that deep will be quite rare. In other scenarios such as general XML query processing in large databases, some researchers expect that documents may be more deeply nested. While such scenarios are beyond the scope of this paper, the interested reader is referred to [Bruno et al. 2003] for a discussion of the performance of NFA-based solutions in such settings.

Given these statistics, the increase in MQPT of the filtering algorithms can be explained by two factors: the increased document length and the increased document depth. In the case of YFilter, the number of state transitions made increases 5.9 times as D is increased from 6 to 10. Much of the increase comes from the simple fact that there are 3.5 times more start/end element pairs in the documents when D = 10 compared to when D = 6. Although the increased document depth could theoretically cause exponential increase in the number of transitions, we did not observe it in this experiment, because in the NFA execution, most input elements can trigger transitions only from a limited subset of the active states.

Note that the NITF DTD we use is one of the few complicated DTDs published online in terms of the number of elements allowed to be recursive (26 out of 123 elements). For this reason, we anticipate YFilter's performance shown in this experiment to serve as a good indicator of its sensitivity to the maximum level of element nesting in most other practical workloads.

4.5 Experiment 4: Varying Non-determinism

In the previous experiments, we kept W and DS (the probability of "*" and "//" operators, respectively) fixed at 0.20. Wildcards and "//" operators, however, are the sources of non-determinism in our NFA-based model. Thus, in this set of experiments we investigate their impact on filtering performance. In order to separate the effects of these two parameters, we fixed one at 0 while varying the other. Note also that we use a large D value (10) in order to allow a reasonable number of distinct queries to be generated for all measured values of W and DS.

Varying W and DS can dramatically impact the properties of the query sets produced by the query generator. Thus, for these experiments we modified our query generation technique. We first generated a large set of distinct queries using the setting (D=10, W=0, DS=0). Then to experiment with different W values, for each query in this set, elements were replaced with wildcards with probability = W; if due to this process, a query became identical to an existing one in the query set, the duplicate query was not added to the set. Query sets for the cases with varying DS were generated similarly.

4.5.1 Varying W

Figure 17 shows the MQPT results when W is varied from 0 to 0.8 with Q = 50,000.[8] As can be seen in the figure, YFilter again significantly outperforms the others. Note also that it is much less sensitive to this parameter than the other two algorithms. The reason for YFilter's low sensitivity to W is explained as follows. As W increases, the size of the NFA changes slowly, due to the prefix sharing among path expressions. As W is increased from 0, the NFA grows somewhat because the addition of wildcards adds new paths to the NFA. As W is further increased, the NFA size actually begins to decrease, as the queries become more similar to each other. In this experiment, the NFA begins with

[8] Note that at W=1 very few distinct queries can be generated, so that case is not shown here.

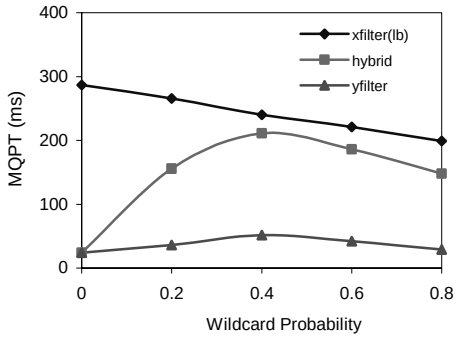


Figure 17: Varying wildcard probability
(NITF, Q=50,000, D=10, DS=0)

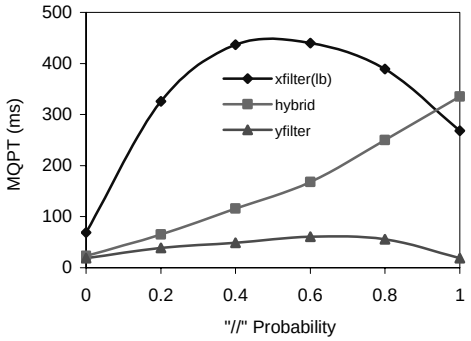


Figure 18: Varying "//" probability
(NITF, Q=10,000, D=10, W=0)

approximately 82,000 states (when W = 0), and increases to a high of approximately 112,000 when W = 0.4.

In contrast, XFilter's performance improves with increasing W. Since XFilter does not store nodes for wildcards, the number of transitions it makes is reduced as wildcards are added.

In this experiment, the performance of Hybrid demonstrates that it does in fact share common attributes with both XFilter and YFilter. When W and DS are both set to 0, Hybrid is similar to YFilter as there is no decomposition of queries. As W (or DS) increases, Hybrid moves more towards XFilter due to increased query decomposition. Beyond a certain point (W = 0.4, here), the benefit of not processing wildcards becomes dominant, and Hybrid's performance improves along with XFilter's.

4.5.2 Varying DS

Figure 18 shows the effect of varying DS (the probability of "//" operators) from 0 to 1 with Q = 10,000 (a smaller number of queries was used here because XFilter was unable to complete for the mid-range values of DS with more queries). As in the previous experiment, YFilter has the best performance overall and is less sensitive to the parameter setting than the other two.

The performance of YFilter is again largely explained by the change in the machine size. As DS is increased from 0 to 1, the machine size first increases because of the diversity of axes in location steps in queries, and then decreases, as queries become more similar to each other. The turning point here occurs at DS = 0.6, where the machine size is 2.8 times that at DS = 0, resulting in a 3.2 times increase in MQPT. The performance degradation is kept small due to the shared representation and processing of "//" operators among multiple queries.

In contrast, XFilter does pay a large performance penalty as DS is increased. This penalty is due to overhead it incurs when processing "//" operators in the presence of recursive elements. Recall that (as described in Section 3.1.3) in XFilter, if a location step "//a" can be matched by recursive "a" elements, the path node of the subsequent location step will be promoted to its candidate list each time "//a" is matched. In XFilter's implementation, if the subsequent location step contains a "//" operator (e.g. "//b"), its

path node is simply added to the candidate list multiple times. However, if the next location step contains a '/' operator instead (e.g. "/b"), different instances of this path node are first created and then added to the candidate list to remember all the possible levels where this location step could be matched. Note that the probability of patterns such as "//a/b" first increases with DS and then decreases. The behavior of XFilter in this experiment is determined by multiple promotions of path nodes in general and the overhead of handling these particular patterns.

In this experiment, Hybrid again exhibits characteristics of the other two algorithms. When DS = 0, Hybrid is similar to YFilter, and as DS is increased, it becomes more like XFilter. Hybrid, however, does not exhibit the bell shape, because it uses a single runtime stack to keep track of the active states as in YFilter, rather than promoting path nodes multiple times to remember different document levels as in XFilter. At DS = 1, every query is decomposed into single elements and the performance of Hybrid is very close to XFilter. XFilter actually outperforms Hybrid a little as a benefit of using list balancing.

The experiments on non-determinism have shown that compared to the other two algorithms, YFilter shows relatively little sensitivity to the W and DS parameters. Due to prefix sharing, increasing the probabilities has only a modest effect on the size of the NFA. As a result, the filtering cost of YFilter is relatively low and robust to changes in these parameters.

4.6 Experiment 5: Maintenance cost

The last set of experiments we report on in this section deal with the efficiency of maintaining the YFilter structure, which is expected to be one of the primary benefits of the approach. Updates to the NFA in YFilter are handled as follows: To insert a query, we merge its NFA representation with the combined NFA as described in Section 3.2.2, and append the identifier of this query to the end of the query ID list at its accepting state. To delete a query, the accepting state of the query is located and the query's identifier is deleted from the list of queries at this state. If the list becomes empty and the state does not have a child state in the NFA, the state is deleted by removing the link from its parent. The deletion of this state can be propagated to its predecessors. An update to a query is treated as a delete of the old query followed by the insertion of the new one.

Deletion is the dual problem of insertion except that modification of the list at the accepting state can be more expensive than appending an identifier to the list. As demonstrated in the previous sections, YFilter's performance is fairly robust with respect to the number of queries in the system. Thus, instead of deleting queries immediately, we adopt a lazy approach where a list of deleted queries is maintained. This list is used to filter out such queries before results are returned. The actual deletions can then be done asynchronously. Thus, in this section we focus on the performance of inserting new queries.

We measured the cost of inserting 1000 queries with varying numbers of queries already in the index (which can contain duplicate queries). The insert costs are shown in Table 5. With Q = 2000 (i.e., 2000 queries already in the NFA), it takes 77 ms to insert

the 1000 new queries. At this point, the chance of a query being new is high, requiring new states to be created and transition functions to be expanded by adding more hash entries to the states. However, the cost drops dramatically as more queries are present in the system. Beyond Q=50,000, the insertion cost stabilizes around 5 ms. This is because most path expressions are already present in the index, so inserting a new query can typically be added by simply traversing down a single path to an existing accepting state and appending the query ID to the list at that state.

Q (x1000)	2	4	6	8	10	10 ~ 50	60 ~ 500
1000 Insertions (ms)	77	57	30	24	9	6	≈ 5

Table 5: Cost of inserting 1000 queries (ms) (NITF, D=6, W=0.2, DS=0.2)

5. VALUE-BASED PREDICATE EVALUATION IN YFILTER

The previous section demonstrated the substantial performance improvements that can be gained by sharing structure matching through the use of an NFA. This sharing, however, would be greatly reduced if value-based predicates (i.e., predicates on individual elements) were encoded directly into the NFA. In this section, we explore two alternative techniques for handling such predicates in the YFilter framework.

Value-based predicates in XPath address properties of elements, such as their content, their position, and their attributes. Examples include:

- The value of an attribute in an element, e.g., //product/price[@currency = “USD”];
- The text data of an element, e.g., //product/price [text() <= 300].
- The position of an element, e.g., /catalog/product[position() = 2], which means “select the second *product* child element of the *catalog* element”.

Any number of such predicates can be attached to a location step in a query.

In this section, we focus on the processing of predicates on attributes or element position but not on the data. Predicates on element data require additional bookkeeping because the data (if present) is delivered by the parser in separate “characters” events that may arrive at any time between the "start element" event and its corresponding “end element” event.

We have developed two alternative approaches to implement value-based selections. The first approach, called *Inline*, applies selection during the execution of the NFA. The second, called *Selection Postponed* (*SP*), simply runs the NFA as described previously, and then applies the selection predicates in a post-processing phase. Below, we discuss these two alternatives in more detail, and compare their performance experimentally.

5.1 The Inline Approach

For the Inline approach, we extend the information stored in each state of the NFA to include any predicates that are associated with that state. These predicates are stored in a table, as shown in Figure 19. Since multiple path expressions may share a state, this table can include predicates from different queries, so we identify predicates using (*Query Id*, *Predicate Id*) pairs.

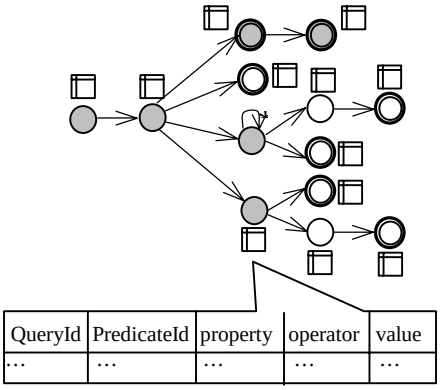


Figure 19: Predicate Storage for Inline

Inline works as follows: When a start-of-element event is received, the NFA transitions as described in Section 3.2.4. For each state reached, the predicates stored there are checked. For each query, bookkeeping information is maintained, indicating which predicates of that query have been satisfied. When an accepting state is reached, the bookkeeping information for the queries of that state is checked, and those queries for which all predicates have been satisfied are returned as matches.

While such an approach sounds conceptually simple, there are several issues to consider. The first is the potential benefit of checking predicates early. The failure of a predicate at a state does not necessarily stop processing along that path because there may be other queries sharing the state that did not fail. Furthermore, if a query contains a "//" prior to a predicate, then even if the predicate fails, the query effectively remains active due to the non-determinism introduced by that axis. For these reasons, the common query optimization heuristic of "pushing selects" to earlier in the evaluation process is not as likely to be effective in this environment.

A second issue is that, due to the nested structure of XML documents, it is possible that backtracking will occur during the NFA processing. Such backtracking further complicates the task of tracking the predicates that have been satisfied. For example, consider query *q4*= "//a[@a_1=v_1][@a_2=v_2]" that contains a location step with two predicates (on two different attributes a_1 and a_2 of "a" elements). If care is not taken during backtracking, a fragment such as "<a a_1=v_1> <a a_2=v_2> " could erroneously be determined to match *q4* even though the attributes are associated with different "a" elements. This problem can be solved by "undoing" changes made to the predicate bookkeeping information for a state when backtracking from that state.

Unfortunately, the above solution does not solve a similar problem that exists for *recursively nested* elements. Consider *q4* when applied to a fragment with nested "a" elements: "<a a_1=v_1> <a a_2=v_2> ". In order to distinguish between the two "a"s additional bookkeeping information must be kept. This additional information identifies the particular element that caused each predicate to be set to true. During the final evaluation for a query at its accepting state, the query is considered to be satisfied only if

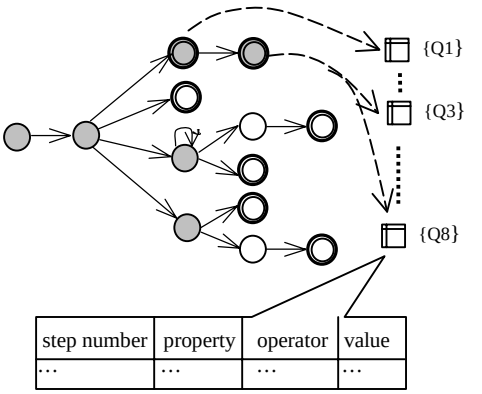


Figure 20: Predicate Storage for SP

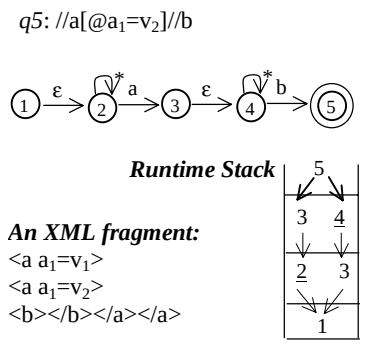


Figure 21: A sample query, its NFA, and the NFA execution

all predicates attached to the same location step are satisfied by the same element. The Inline approach is described in more detail in Appendix A.

5.2 Selection Postponed (SP)

Effort spent evaluating predicates with Inline will be wasted if ultimately, the structure-based aspects of a query are not satisfied. The Selection Postponed (SP) approach avoids this problem by delaying predicate processing until after the structure matching has been completed. SP has several other potential advantages. First, since the predicates on different elements in a query are treated as conjunctions, a short-cut evaluation method is possible; when a predicate of a query fails, the evaluation of the remaining predicates of that query can be avoided.[9] Second there is no need to extend the NFA backtracking logic as for Inline.

In SP, the predicates are stored with each query, as shown in Figure 20. We index the predicates for a query by the "step number" field. When an accepting state is reached in the NFA, selections are performed in bulk for each query associated with the state. If all predicates of a query evaluate to true, then the query is satisfied.

In order to delay selection, however, the NFA must be extended to retain some additional history about the states visited during structure matching. The reason for this is demonstrated by the following example. Consider query *q5* and an XML document fragment as shown in Figure 21. When element 'b' of the document is parsed, the NFA execution arrives at the accepting state of this query in the NFA (also shown in Figure 21). When selection processing is performed for *q5*, we need to decide on which of the two 'a' elements encountered during parsing to apply the predicate.

A naïve method would be to simply check all of the 'a' elements encountered. Unfortunately with more "//" operators in a query or more recursive elements in the document, searching for matching elements for predicate evaluation could become as expensive as running the NFA again for this query. Instead, we extend the NFA to output

[9] Note however, that with predicate evaluation it becomes possible to visit a given accepting state multiple times, due to predicate failure. Such short-cut predicate evaluation only saves work for a single visit.

not only *query ID*s, but a list of path matches. Each path match provides a list of document elements that should participate in predicate evaluation.

For example, at the accepting state for *q5*, the NFA would report the two path matches "$a_{\_1}$ b" and "$a_{\_2}$ b", where $a_{\_1}$ represents the first 'a' element and $a_{\_2}$ represents the second (nested) 'a' element. Since predicates are indexed by "step number", it is easy for the selection operator to determine which elements need to be tested. For the XML fragment shown in Figure 21, the first path match does not satisfy *q5* because $a_{\_1}$ does not satisfy the predicate, but the second path match does.

The NFA is extended to output these path matches by linking the states in the runtime stack backwards towards the root. That is, for each target state reached from an active state, we add a predecessor pointer for the target state and set the pointer to the active state. Then the target state with the pointer is later pushed onto the runtime stack. An example is shown in Figure 21, which includes the content of the stack for the accepting state of the sample query after the XML fragment was read.

For each state that is an accepting state, we can traverse backwards to find the sequence of state visits that lead to the accepting state. Note that elements that trigger transitions to "//-child" states (along self-loops) can be ignored in this process, as they do not participate in predicate evaluation. Returning to the example in Figure 21, there are two sequences of state visits, namely "2 3 5" and "3 4 5" that the NFA took when elements $a_{\_1}$, $a_{\_2}$ and b were read. After eliminating the elements that trigger transitions to "//-child" states for each state sequence, the two sequences of matching elements, "$a_{\_2}$ b" and "$a_{\_1}$ b", can be generated for predicate evaluation.

A note is that our technique of linking states in the runtime stack using predecessor pointers is similar in spirit to "backward chaining" used in *PathStack* and *TwigStack* [Bruno et. al. 2002]. The idea in both is to use backward pointers to store partial or complete matches of path expressions. The difference is that we use a single runtime stack with backward pointers to store matches for all path expressions, while *PathStack* requires a stack for each query node.

The evaluation data structures and pseudo-code for predicate evaluation using SP are presented in Appendix B. Note that SP requires no bookkeeping information and that the evaluation code is simple and straightforward.

Finally, as mentioned above, predicates on element data cannot be evaluated with other value-based predicates in a query, because the element data is not returned when the "start element" event is encountered. The fact that selection in SP is decoupled from the event-based processing enables us to treat selections involving such predicates simply as blocking operators. To collect information for such selection operators, we extend the elements carried by the path matches to include a data field called "text". When a "characters" event is received, the data returned by this event is appended to the "text" field in the corresponding element. This field is known to be complete when the corresponding "end element" event is encountered. At that moment, selection operators blocked on this field will be signaled to become unblocked.

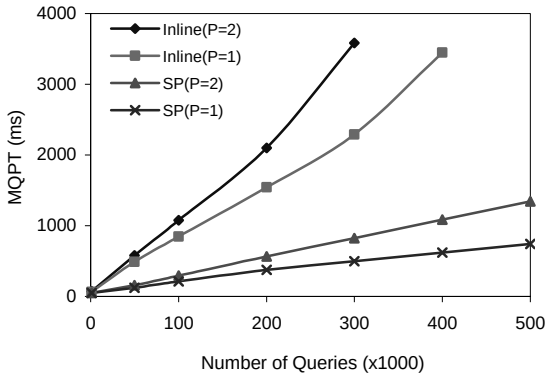


Figure 22: Varying number of queries (D=6, W=0.2, DS=0.2)

5.3 Performance of Value-based Predicate Evaluation

Having described the Inline and SP approaches to value-based selection, we now present results from an experimental study comparing their performance. The NITF DTD was used for all experiments presented in this section. For query generation, the parameter P (see Table 2) was used to determine the number of predicates that appear in each query. Such predicates are distributed over the location steps uniformly at random. Distinct queries are used in all of the experiments.

In the first experiment we examine the relative performance of Inline and SP as Q is varied from 1,000 to 500,000. Figure 22 shows the MQPT of the two approaches for the cases P=1 and P=2.

As can be seen in the figure, SP outperforms Inline by a wide margin. When P=1, for example, SP takes 375 ms to process 200,000 queries, while Inline takes 1170 ms more. To understand these results, recall the three major differences between Inline and SP.

1) **Structure matching and value matching**: Inline performs early predicate evaluation before knowing if the structure is matched, and this early predicate evaluation does not prune future work. In contrast, SP performs structure matching to prune the set of queries for which predicate evaluation needs to be considered.
2) **Conjunctive predicates in a query**: In Inline, evaluation of predicates in the same query happens independently at different states, while in SP, the failure of one predicate in a query stops the evaluation of the rest of predicates immediately.
3) **Bookkeeping**: Inline requires bookkeeping information for the final evaluation of a query. The maintenance cost includes setting the information and undoing it during backtracking. Note that in addition to reduced MQPT, bookkeeping overhead causes Inline to run out of memory, for Q above 400,000.

When the number of predicates per query is doubled (P=2, also shown in Figure 22) both approaches suffer an increase in MQPT. The differences between the approaches, however, are more pronounced. For example, for 200,000 queries containing two predicates each, Inline takes 1534 ms more than SP. Inline also experiences a tremendous

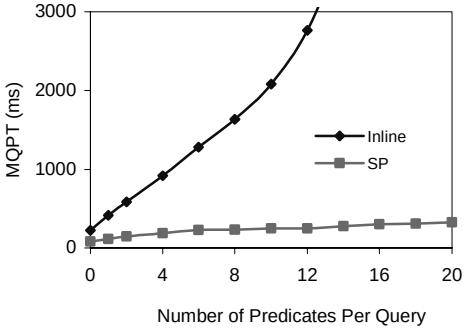


Figure 23: Varying number of predicates
(D=6, Q=50000, W=0.2, DS=0.2)

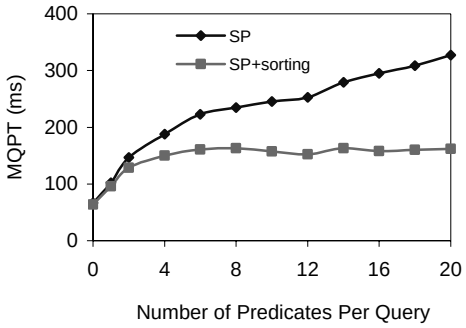


Figure 24: Effect of predicate sorting
(D=6, Q=50000, W=0.2, DS=0.2)

increase in the bookkeeping overhead, and runs out of memory with 100,000 queries earlier than P=1.

Figure 23 shows the MQPT of the two approaches as the number of predicates per query is varied from 0 to 20 for a relatively small number of queries (Q =50,000). As can be seen in the figure, a large number of predicates compounds the poor performance of Inline. In contrast, SP is much less sensitive to the number of predicates per query. As P increases, the increased cost in SP results from a larger number of invocations of predicate evaluation and longer evaluation periods. Luckily, the negative impact is limited by the short-cut evaluation strategy.

The previous experiment demonstrated the benefits of delaying content-based matching in YFilter. One of the major benefits was seen to be the ability to "short-cut" the evaluation process for a query when one predicate fails. This observation raises the potential to further improve the chances of such short-cut evaluation by evaluating highly-selective predicates first, as is done by most relational query optimizers.

If statistics on documents are kept, then the selectivity of predicates on attributes can be estimated from the probability of an attribute occurring in an element and the number of values this attribute can take. Examples of equality predicates on attributes of element "a" are given as follows:

selectivity ([***@attr***]) = probability of the attribute occurring in element '*a*'.

selectivity ([***@attr='v***']) = *Selectivity*[*@attr*] / max. # of values attribute "*attr*" can take.

The selectivity estimates for predicates involving other comparison operators can be derived in a similar way. If predicates are attached to a wildcard in a location step, we make simple assumptions about their selectivity. Other formulas are omitted here due to space constraints.

We performed a simple experiment to examine the potential performance benefits of predicate reordering in the SP approach. Figure 24 shows the MQPT for SP with and without sorting, as P is varied from 0 to 20 for Q=50,000. The results indicate that as expected, additional benefits can indeed be gained by predicate sorting, particularly for cases with large numbers of predicates.

6. NESTED PATH EXPRESSIONS

In the previous section, we described two approaches for value-based predicate evaluation in YFilter. As our experimental results show, SP outperforms Inline by a wide margin. Recall that in SP is that there is clear separation between path matching and predicate evaluation. Our technique for handling nested path expressions in YFilter leverages this post-processing of path matches.

6.1 Preliminaries

We begin by more clearly specifying the interface between the NFA path matching engine and the post-processing operators. This interface is based on *path match* structures that identify the document elements that caused the NFA to reach an accepting state. During parsing document elements are given unique identifiers. Each time an accepting state is reached, the NFA outputs a path match structure for each query associated with that state. At an accepting state that represents a path expression of *n* location steps, each structure generated is simply a list of identifiers of the *n* elements that matched the path expression.

The elements that path matches reference are stored in memory resident data structures created in document parsing. These data structures hold attributes and text data of the corresponding elements which could be used by any operators in post-processing.

6.2 Query Decomposition

The original work on XFilter proposed using *query decomposition* to handle nested path expressions. In this approach, the nested paths are extracted from the main path expressions and processed individually. A post-processing phase is used to link matched paths back together to determine if an entire query expression has been matched. The advantage of such an approach is that the path matching engine remains untouched. We follow a similar approach in YFilter, using the NFA/post-processing interface described above. In YFilter, however, this approach has the significant additional benefit that it naturally allows shared path matching to be exploited for nested path expressions.

We describe the approach by addressing: first, how the nested paths are represented, and second, how they are evaluated. We then present results from a performance study of our implementation.

6.2.1 Query representation

For ease of exposition, we initially describe our solution assuming only one level of path nesting in queries. In other words, a nested path does not itself contain any nested paths. We then relax this assumption in Section 6.2.3. For such queries we can define three terms: A *main path* is the remaining structure of a query after all the nested paths are removed. An *anchor step* of a nested path is a location step in the main path where that nested path is attached to the main path. An *extended nested path* is a nested path prepended with the prefix of the main path up to its anchor step.

In our approach, when a query containing nested paths is parsed, it is decomposed into a list of absolute paths: the main path and any extended nested paths. For example,

consider the query *q6*="/a[d]//b[e/f]/c". It contains two nested paths "d" and "e/f". Query decomposition produces a main path, "/a//b/c", and two extended nested paths, "/a/d" and "/a//b/e/f". We assign these paths identifiers consisting of pairs (*QueryId*, *PathId*), where the main path has PathId 0 and the nested paths are numbered sequentially. All of the paths are then inserted individually into the path matching engine with these identifiers. We slightly extend the interface described above so that the engine returns path matches to queries using the Query Ids with the Path Ids attached to the matches for the use inside those queries.

Post-processing is implemented using operators called *Nested Path Filters* (NP-Filter). Each NP-Filter is associated with a single query. Under the assumption of a single level of nesting, only one NP-Filter is required per query. The NP-Filter contains information for each path of its associated query. For each nested path, it stores the *position* of its anchor step in the main path. This position will identify the last shared element between the extended nested path and the main path. The NP-Filter also contains for each path (main and nested) a *store* to keep the path matches corresponding to that path.[10]

6.2.2 Query evaluation

As previously stated, queries containing nested paths are processed in two phases, path matching and post-processing of the path matching results. The first phase is completely done by the path matching engine. Thus, the processing of the common prefixes is shared among all the paths, e.g., between main paths and extended nested paths and among the extended nested paths themselves. Upon obtaining a new path match, the engine delivers it to the queries containing the path, together with the PathId of this path in each of those queries. The recipient queries hold this path match in one of its stores identified using the attached PathId.

Post-processing is performed inside each NP-Filter at the end of document processing. This processing consists of the following steps:

1) *Store check*: If any of the stores of the constituent paths of the query is empty, then return *False*.
2) *Filter construction*: Otherwise, a filter is constructed for each nested path from its corresponding store by extracting the set (no duplicates) of element ids that appear at the anchor step position of the nested path.
3) *Match filtering*: The path match structures of the main path are then pipelined through all the nested path filters. For each main path match, a nested path filter is applied to the element identifier at the corresponding anchor step position. If the filter does not contain this element identifier, the main path match is evicted. If a main path match passes all the filters, the query is evaluated to *True* and the NP-Filter stops.

[10] In the implementation, a path match store is allocated for each unique path expression and shared among all queries containing this path, so an NP-Filter only contains pointers to these shared stores. Due to this sharing, the stores contain path matches in their entirety, even though any one query may not need all of the elements.

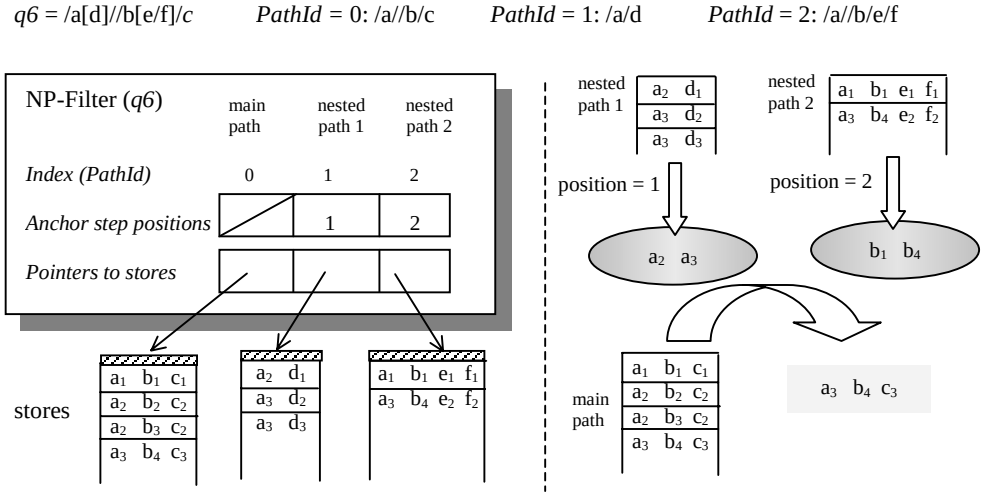


Figure 25: An example NP-Filter operator and its match filtering process

Figure 25 shows the three constituent paths of *q6* and an NP-Filter operator for it, and illustrates the post-processing performed for this query. On the left of the figure, data structures maintained in the NP-Filter are shown in the upper box. In the list of anchor step positions, the list element at index 1 corresponds to the first nested path (i.e., *PathId* 1), indicating that the anchor step of this nested path is at position 1 in the main path. The list element at index 2 keeps the position information for the second nested path. In the store list, three pointers link to the stores that contain path matches for the three constituent paths.

The right part of the figure illustrates the execution of the NP-Filter. The arrows drawn top down depict filter construction for the two nested paths. Anchor step positions are used to extract element ids for each filter. The arrow below the filters illustrates pipelining the main path matches through these two filters. The first main path match is eliminated by the first filter, because the identifier of the 'a' element in this match is not contained in the filter. The next two matches are removed by the second filter. Finally the last main path match passes both filters, and the query is evaluated to *True*.

6.2.3 Support of Multiple Levels of Path Nesting

In the above description, we assumed that nested path expressions do not themselves contain nested paths. The approach, however, can be extended to support an arbitrary number of levels of path nesting. We first modify NP-Filter operators so that they can be configured to output one or all matches retained from the nested path filters. The rest of the extension is outlined as follows.

For each query involving multiple levels of path nesting, an NP-Filter is assigned to each path expression (absolute or nested) that contains nested paths in its predicates. If additional NP-Filters are assigned to the nested paths of this path expression, the NP-Filter of this path expression treats them as child operators. In this way, a hierarchy of NP-Filters is formed in correspondence to the hierarchy of path nesting.

During post-processing, the hierarchy of NP-Filters is executed bottom up. NP-Filters at the bottom level of the hierarchy access path match stores and perform match filtering as described above. They output all main path matches that are retained from their nested paths filters. After NP-Filters at the next level receive the path matches from their child operators, they start the execution and output in the same matter. This process continues until the top level NP-Filter finds any main path match or exhausts all the input matches. In the first case, the query is evaluated to *True*.

6.4 Evaluation of Nested Path Expressions

In this subsection, we evaluate the performance of our approach to processing nested path expressions. Recall that in this approach, path matching is shared among all queries, and post-processing is performed on a per-query basis. Our experimental study provides some understanding of the component costs as well as total processing cost in MQPT.

The parameter NP (see Table 2) was used to generate a number of nested path expressions in each query. Such nested paths are distributed over the location steps uniformly at random. The depth of a nested path is determined by the difference between maximum depth D and the actual depth of the location step where this nested path is attached. The setting of parameters W and DS, is also applied to the nested paths. All queries used in the experiments contain only one level of path nesting.

6.4.1 Varying Q and varying NP

In this experiment, we varied the number of distinct queries from 1000 to 200,000 for three values of NP, 1, 2 and 3. Figure 26 shows YFilter's performance in terms of MQPT.

An important trend is observed from this figure. As the number of queries grows, there is a fair amount of increase in MQPT to process the first nested paths in queries (see the case of NP=1). Processing additional nested paths in queries (in the cases of NP>1), however, costs only a little more than processing the first nested paths. Consequently, the cases of larger NP values exhibit efficiency and scalability very close to that in the case of NP=1.

For a better understanding of this result, we implemented a profiler that reports the costs of the path matching engine and NP–Filter operators, and also provides statistics that help explain the observed execution costs. We re-ran the above experiment with the profiler turned on. Due to the overhead of running the profiler, the costs reported in this manner are higher than the costs observed while running the actual experiment. As a sample of the content of the report, we show in Table 6 the total cost of the path matching engine, the total cost of all NP-Filters, and some statistics at Q=50,000.

As Table 6 shows, when NP=1, the path matching engine costs much more than NP-Filters. We have demonstrated in Section 4 that the NFA execution is very efficient. Here, the engine cost is dominated by generation and delivery of multiple path matches during each of the 5988 visits to accepting states. In contrast, NP-Filters have a relatively low cost, due to the use of the store check as the first processing step. In this experiment, most queries cannot have both constituent paths satisfied by a document, so their NP-Filters only need to perform the inexpensive store check.

Q=50,000, NP =	1	2	3
Engine cost (ms)	152	160	171
NP-Filter cost (ms)	33	30	28
…	…	…	…
# of States in the NFA	42198	48523	57468
# of accepting states hit	5988	6193	6701
# of matched queries	3226	1837	770
…	…	…	…

Table 6: Profile on nested path processing (Q=50,000, D=6, W=0.2, DS=0.2)

When examining cases of NP=2 and NP=3 in Table 6, we observe that the effects of adding more nested paths are two-fold. First, it increases the cost of the engine, e.g. from 152 ms when NP=1 to 171 ms when NP=3. Our analysis of this cost increase is the following. After the additional nested paths are added to the engine, they increase the size of the NFA as shown in Table 6 (by 36% from NP=1 to NP=3). This increase, however, is much less than that of the total number of nested paths, due to the path sharing exploited by the engine. The increased machine size causes some more visits to accepting states during document processing, e.g., 12% more from NP=1 to NP=3, which in turn results in a slightly higher engine cost. The small increase indicates that after paying the cost for the first nested paths, queries can obtain matches to most of their additional nested paths at no extra cost. In other words, the cost of processing the initial nested paths can be amortized by additional nested paths in queries.

The second effect of adding more nested paths is the slight reduction of the cost of the NP-Filter operators. The additional nested paths increase query selectivity, as evidenced by the reduced number of query matches shown in Table 6. Due to this increased query selectivity, more NP-Filters can terminate due to store checks, thus improving the overall cost of NP-Filters slightly.

The combination of these two effects determines the small increase in MQPT from processing single nested paths in queries to multiple ones in them.

6.4.2 Performance for queries with mixed predicates

We further investigated the performance of YFilter when queries contain both value-based predicates and nested path expressions. We integrate predicate evaluation into the NP-Filters by applying the predicates to the paths immediately after the store check. Thus, predicate evaluation is performed only if all constituent paths in the query are satisfied. Similarly, the later steps of NP-Filter execution, namely, filter construction and match filtering, are executed only when all paths also pass the selection evaluation.

To examine the performance of mixed predicates, we took the query sets from the case of NP=1 of the previous experiment, and added a single value-based predicate to the main path of each query. Then we ran the experiment by varying the number of queries from 1000 to 200,000 for the two cases, (NP=1, P=0) and (NP=1, P=1). Figure 27 shows the MQPT results.

We see that adding a value-based predicate to queries containing nested paths incurs only a very modest increase in MQPT. This phenomenon can be explained by two

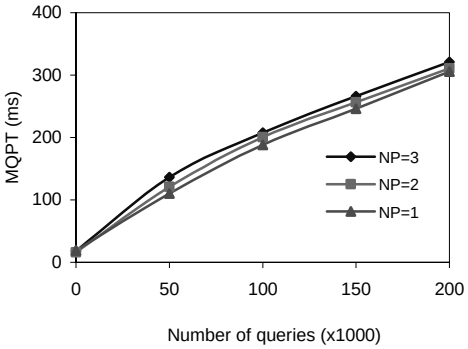


Figure 26: Varying number of queries (D=6, W=0.2 DS=0.2, P=0)

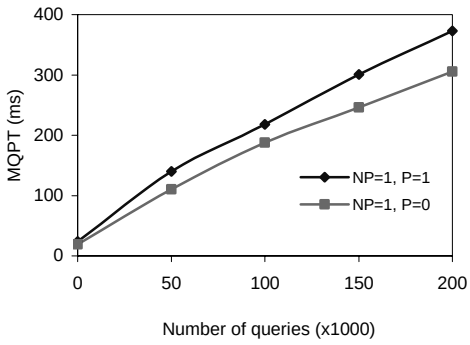


Figure 27: Varying number of queries (D=6, W=0.2 DS=0.2, NP=1)

factors. First, selection operators (using the SP approach) and NP-Filter operators completely share the overhead of path matching engine, e.g. the NFA-based path navigation and the more expensive operations to generate and deliver path matches. Second, due to the way that we combine selection evaluation with NP-Filter execution, much of the predicate evaluation is avoided by the store check performed at the beginning of NP-Filter execution.

Experimental results on nested path processing in YFilter can be summarized as follows: 1) There is a fair amount of increase in MQPT to process the first nested paths in queries. The cost is dominated by the overhead of supporting the interface of returning path matches for post-processing. 2) The cost increase can be amortized through path sharing when processing additional nested paths in queries, which results in good efficiency and scalability in the cases of multiple nested paths per query. 3) This cost increase can also be recovered when processing value-based predicates.

7. RELATED WORK

User profile modeling and matching have been extensively investigated in the context of *Information Filtering* and *Selective Dissemination of Information* research, e.g., [Foltz and Dumais 1992]. IR-style user profiles are intended for unstructured text-based systems and typically use sets of keywords to represent user interests. In general, IR profile models can be classified as either *Boolean* or *Similarity-based*. The former use an exact match semantics over queries consisting of keywords connected with Boolean operators. The latter use a fuzzy match semantics, in which a similarity value is assigned to every (document, profile) pair. A document with similarity to a profile over a certain threshold is said to match the profile [Salton 1989; Belkin and Croft 1992; Cetintemel et al. 2000]. The *Stanford Information Filtering Tool* (SIFT) [Yan and Garcia-Molina 1994; Yan and Garcia-Molina 1999] is a keyword-based text filtering system for Internet News articles that supports both profile models. Our work differs from IR-based filtering in that it is targeted at application domains in which data is encoded in XML and user profiles take advantage of the rich semantic and structural information embedded in the data for more precise filtering.

Query-based profile models have also been studied in the context of *Continuous Queries* (CQ), which are standing queries that allow users to get new results whenever an update of interest occurs. *Tapestry* [Terry et al. 1992] was early work on CQ in relational append-only databases. More recently, *OpenCQ* [Liu et al. 1999] and *NiagaraCQ* [Chen et al. 2000; Chen et al. 02] have been proposed for information delivery on the Internet, using the relational model and its techniques. *OpenCQ* uses grouped triggers for CQ condition checking and query processing with cached views for incremental result delivery. *NiagaraCQ* incrementally groups query plans of continuous queries using common *expression signatures*. *CACQ* [Madden et al. 2002] further combines adaptivity and grouping for CQ. It also breaks the abstraction of shared relational algebra expressions and shares physical operators among tuples by attaching states to them individually. These systems focus on relational techniques and do not address matching constraints over the structure of the data.

Triggers [Stonebraker 1990; Widom and Finklestein 1990; Schreier et al. 1991] in traditional database systems are similar to CQ. However, triggers are a more general mechanism that can involve predicates over many data items and can initiate updates to other data items. Thus, trigger solutions are typically not optimized for fast matching of individual items to vast numbers of relatively simple queries. Some recent work has addressed the issue of scalability for simple triggers by grouping predicates into equivalence classes and using a selection predicate indexing technique [Hanson et al 1999]. However, this work has not addressed the XML-related issues that XFilter and YFilter handle.

A number of XML filtering techniques have been proposed. Our first filtering system, XFilter, is among the earliest work that addresses filtering XML documents for a large number of profiles written in an XML query language. In the context of XFilter, *CQMC* [Ozen et al. 2001] builds a FSM for all the queries that have identical structure requirements. XTrie [Chan et al. 2002] 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 addition, both of those systems support broader functionality, e.g. ordered matching, formatting, or predicate evaluation in a simple way. Compared to our work in YFilter, however, they exploit less path sharing. Furthermore, none of this prior work has studied the integration of predicate evaluation and structure matching. A recent study by Bruno et al. [Bruno el al. 2003] shows that compared to an index-based approach to scan streaming documents for multiple queries, YFilter's approach is particularly effective for short documents and large numbers of queries. *MatchMaker* [Lakshmanan and Parthasarathy 2002] was among the first attempts to efficiently match multiple tree patterns. It constructs disk-resident requirement indexes on pattern nodes and path operators, and then labels document nodes with matching queries using the indexes. Due to the I/O invocations in the query processing, it reports filtering performance orders of magnitude slower than the other memory-based filtering algorithms.

The evaluation of path expressions on streaming data was studied in [Ives et al. 2000], where queries that include resolving IDREFs are expressed by several individual FSMs that are generated on the fly. [Green et al. 2003] proposes a structure matching approach that combines all path expressions into a single DFA, resulting in good performance, but with significant limitations. As shown in our experiments, when using an NFA-based algorithm such as YFilter, structure matching is no longer the dominant cost of filtering. As a result we do not believe that trading flexibility for any further improvements in structure matching speed is worthwhile.

A number of event-based publish/subscribe systems have been developed. Among them, *Xlyeme* [Nguyen et al. 2001] tackles the problem of finding the complex events associated with monitoring queries that are satisfied by incoming documents. It uses a hierarchy of hash tables to index sets of atomic events that compose more complex events. *Le Subscribe* [Fabret et al. 2001] proposes a predicate model to specify subscriptions. To match incoming events (i.e. a set of attribute value pairs) with predicates efficiently, all predicates are indexed, and all subscriptions are clustered by their common conjunctive predicates. A cost model and algorithms are developed to find good clustering structure and to dynamically optimize it. A common feature of these systems is the use of restricted profile languages and data structures tailored to the complexity of the languages in order to achieve high system throughput. *WebFilter* [Pereira et al. 2001] demonstrates a system that processes XML encoded events, by translating XML data in a parsing step to events that consist of set of attribute value pairs and then using techniques developed in *Le Subscribe*. Details on structure matching are not provided in this paper.

Finally, *DataGuides* [Nestorov et al. 1999; Goldman and Widom 1997] are structural summaries of an XML source database that can be used to browse database structure, formulate queries, and enable query optimization. Creating a DataGuide from a source database has been proved to be equivalent to converting an NFA to a DFA. Our NFA-based work differs in that it is intended to represent path expressions rather than data and that it must faithfully encode all of the expressions in their entirety, rather than just summarizing them. As a result, the implementations of the YFilter NFA and DataGuides differ significantly.

8. CONCLUSIONS

In this paper, we studied integrated approaches to handling both structure-based and content-based filtering of XML documents. We first described XFilter, an FSM-based approach using event-based parsing and a dynamic index, which represents what is to our knowledge, the first such approach in the literature. Next, we described YFilter, an NFA-based structure matching engine that provides flexibility and excellent performance by exploiting overlap of path expressions. Using YFilter, path matching is no longer the dominant cost for XML filtering.

We then investigated two alternative techniques for integrating value-based predicate matching with the NFA. Experimental results comparing these techniques provide a key

insight arising from our study, namely, that structure-based matching and content-based matching cannot be considered in isolation when designing a high-performance XML filtering system. In particular, our experiments demonstrated that contrary to traditional database intuition, pushing even simple selections down through the combined query plan may not be effective, and in fact, can be quite detrimental to performance due to the way that sharing is exploited in the NFA, and due to certain properties of XML documents and XPath queries. Finally, we discussed how YFilter has been extended to support nested path processing and demonstrated that our solution built on shared path sharing is efficient even for large numbers of queries containing multiple nested paths each.

The results presented here, as well as other efforts cited in the related work, have demonstrated that XML Filtering is a rich source of research issues. Furthermore, as XML continues to gain acceptance in technologies such as Web Services, Event-based Processing, and Application Integration, this work will be of increasing commercial importance. As such, there are many important problems to be addressed in future work. These include, the integration of filtering with dissemination, incorporation of more expressive query languages such as XQuery, and ultimately, the extension of the filtering concept into a more general notion of XML Routing in a wide-area distributed environment. Research on all of these issues is currently underway.

ACKNOWLEDGEMENTS

We would like to thank Raymond To for helping us develop YFilter, and Philip Hwang for helping provide insight into XML parsing. We would also like to thank Sirish Chandrasekaran, Ryan Huebsch, and Sailesh Krishnamurthy for valuable comments on early drafts of this paper.

REFERENCES

AKSOY, D., ALTINEL, M., BOSE, R., CETINTEMEL, U., FRANKLIN, M.J., WANG, J., AND ZDONIK, S.B. 1998. Research in data broadcast and dissemination. In *Proc. of the 1st International. Conference on Advanced Multimedia Content Processing*. Springer, Berlin, Germany, 194-207.

ALTINEL, M., AKSOY, D., BABY, T., FRANKLIN, M.J., SHAPIRO, W., AND ZDONIK, S.B. 1999. DBIS-Toolkit: Adaptable middleware for large scale data delivery. In *Proc. of SIGMOD 1999*. ACM Press, New York, NY, USA, 544-546.

ALTINEL, M., AND FRANKLIN, M.J. 2000. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of VLDB 2000*. Morgan Kaufmann, San Francisco, CA, USA, 53-64.

Apache XML project. 1999. Xerces Java parser 1.2.3 Release. http://xml.apache.org/xerces-j/index.html.

BELKIN, N.J., AND CROFT, B.W. 1992. Information filtering and information retrieval: Two sides of the same coin? *Communications of the ACM*, 35, 12, 29-38.

BRUNO, N., GRAVANO, L., KOUDAS, N., AND SRIVASTAVA, D. 2003. Navigation- vs. index-based XML multi-query processing. In *Proc. of ICDE 2003*. IEEE Computer Society, Los Alamitos, CA, USA.

BRUNO, N., SRIVASTAVA, D., AND KOUDAS, N. 2002. Holistic twig joins: Optimal XML pattern matching. In *Proc of SIGMOD 2002*. ACM Press, New York, NY, USA, 310-321.

BUSSE, R., CAREY, M., FLORESCU, D., KERSTEN, M., MANOLESCU, I., SCHMIDT, A., AND WAAS, F. 2001. Xmark: An XML benchmark project. http://monetdb.cwi.nl/xml/index.html.

CETINTEMEL, U., FRANKLIN, M.J., AND GILES, C.L. 2000. Self-adaptive user profiles for large scale data delivery. In *Proc. of ICDE 2000*. IEEE Computer Society, Los Alamitos, CA, USA, 622-633.

CHAMBERLIN, D., FANKHAUSER, P., FLORESCU, D., MARCHIORI, M., AND ROBIE, J. 2002. XML query use cases. W3C Working Draft. http://www.w3.org/TR/2002/WD-xmlquery-use-cases-20020430.

CHAN, C., FELBER, P., GAROFALAKIS, M., and RASTOGI, R. 2002. Efficient filtering of XML documents with XPath expressions. In *Proc. of ICDE 2002*. IEEE Computer Society, Los Alamitos, CA, USA, 235-244.

CHEN, J., DEWITT, D.J., AND NAUGHTON, J.F. 2002. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *Proc. of ICDE 2002*. IEEE Computer Society, Los Alamitos, CA, USA, 345-356.

CHEN, J., DEWITT, D.J., TIAN, F., AND WANG, Y. 2000. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proc. of SIGMOD 2000*. ACM Press, New York, NY, USA, 379-390.

CLARK, J. 1999. XSL transformations (XSLT) - version 1.0. http://www.w3.org/TR/xslt.

CLARK, J., AND DEROSE, S. 1999. XML path language (XPath) - version 1.0. http://www.w3.org/TR/xpath.

COVER, R. 1999. The SGML/XML Web Page. http://www.w3.org/TR/xslt.

DEROSE, S., DANIEL Jr., R., AND MALER, E. 1999. XML pointer language (XPointer). http://www.w3.org/TR/WD-xptr.

DEUTSH, A., FERNANDEZ, M., FLORESCU, D., LEVY, A., AND SUCIU, D. 1998. XML-QL: A query language for XML. http://www.w3.org/TR/NOTE-xml-ql.

DIAO, Y., FISCHER, P., FRANKLIN, M.J., AND TO, R. 2002. YFilter: Efficient and scalable filtering of XML documents. In *Proc. of ICDE 2002*. IEEE Computer Society, Los Alamitos, CA, USA, 341.

DIAZ, A.L., AND LOVELL, D. 1999. XML generator. http://www.alphaworks.ibm.com/tech/xmlgenerator.

FABRET, F., JACOBSEN, H-A., LLIRBAT, F., PEREIRA, J., ROSS, K.A., AND SHASHA, D. 2001. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proc. of SIGMOD 2001*. ACM Press, New York, NY, USA, 115-126.

FOLTZ, P.W., AND DUMAIS, S.T. 1992. Personalized information delivery: An analysis of information filtering methods. *Communications of the ACM*, 35, 12, 51-60.

FRANKLIN, M.J., AND ZDONIK, S. 1998. "Data in your face": Push technology in perspective. In *Proc. of SIGMOD 1998*. ACM Press, New York, NY, USA, 156-159.

GOLDMAN, R., AND WIDOM, J. 1997. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. of VLDB 1997*. Morgan Kaufmann, San Francisco, CA, USA, 436-445.

GREEN, T.J., MIKLAU, G., ONIZUKA, M., AND SUCIU, D. 2003. Processing XML streams with deterministic automata. In *Proc. of ICDT 2003*. Springer, Berlin, Germany, 173-189.

HANSON, E.N., CARNES, C., HUANG, L., KONYALA, M., NORONHA, L., PARTHASARATHY, S., PARK, J.B., AND VERNON, A. 1999. Scalable trigger processing. In *Proc. of ICDE 1999*. IEEE Computer Society, Los Alamitos, CA, USA, 266-275.

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

HORS, A.L., HICOL, G., WOOD, L., CHAMPION, M., AND BYRNE, S. 2001. Document object model core (level 2). http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html

IVES, Z., LEVY, A., AND WELD, D. 2000. Efficient evaluation of regular path expressions on streaming XML data. Technical Report, University of Washington, Seattle, WA.

KAY, M.. 2001. Saxon: the XSLT processor. http://users.iclway.co.uk/mhkay/saxon/.

LAKSHMANAN, L.V.S., AND Parthasarathy, S. 2002. On efficient matching of streaming XML documents and queries. In *Proc. of EDBT 2002*. Springer, Berlin, Germany, 142-160.

LEY, M. 2001. DBLP DTD. http://www.acm.org/sigmod/dblp/db/about/dblp.dtd.

LIU, L., PU, C., AND TANG, W. 1999. Continual queries for Internet scale event-driven information delivery. *Special Issue on Web Technologies*, *IEEE TKDE*, 11, 4, 610-628.

MADDEN, S., SHAH, M., HELLERSTEIN, J.M., AND RAMAN, V. 2002. Continuously adaptive continuous queries over streams. In *Proc. of SIGMOD 2002*. ACM Press, New York, NY, USA, 49-60.

NESTOROV, S., ULLMAN, J.D., WIENER, J.L., AND CHAWATHE, S.S. 1997. Representative objects: Concise representations of semistrctured hierarchical data. In *Proc. of ICDE 1997*. IEEE Computer Society, Los Alamitos, CA, USA, 79-90.

NGUYEN, B., ABITEBOUL, S., COBENA, G., AND PREDA, M. 2001. Monitoring XML data on the Web. In *Proc. of SIGMOD* 2001. ACM Press, New York, NY, USA, 437-448.

OZEN, B., KILIC, O., ALTINEL, M., AND DOGAC, A. 2001. Highly personalized information delivery to mobile clients. In *Proc. of the 2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access*. ACM Press, New York, NY, USA, 35-41.

PEREIRA, J., FABRET, F., LLIRBAT, F., AND JACOBSEN, H-A. 2001. WebFilter: A high-throughput XML-based publish and subscribe System. In *Prof. of VLDB 2001*. Morgan Kaufmann, San Francisco, CA, USA, 723-724.

SALTON, G. 1989. *Automatic Text Processing*. Addison-Wesley Co., Boston, MA.

Sax Project Organization. 2001. SAX: Simple API for XML. http://www. saxproject.org.

SCHREIER, U., PIRAHESH, H., AGRAWAL, R., AND MOHAN, C. 1991. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proc. of VLDB 1991*. Morgan Kaufmann, San Francisco, CA, USA, 469-478.

STONEBRAKER, M., JHINGRAN, A., GOH, J., AND POTAMIANOS, S. 1990. On rules, procedures, caching and views in data base systems. In *Proc. of SIGMOD 1990*. ACM Press, New York, NY, USA, 281-290.

Sun Microsystems, Inc. 2001. Java XML pack. Winter 01 update release. http://java.sun.com/xml/downloads/javaxmlpack.html.

TERRY, D.B., GOLDBERG, D., NICHOLS, D.A., AND OKI, B.M. 1992. Continuous queries over append-only databases. In *Proc. of SIGMO 1992*. ACM Press, New York, NY, USA, 321-330.
WATSON, B.W. 1997. Practical optimization for automata. In *Proc. of the 2nd International Workshop on Implementing Automata*. Springer, Berlin, Germany, 232-240.
WIDOM, J., AND FINKLESTEIN, S.J. 1990. Set-oriented production rules in relational database systems. In *Proc. of SIGMOD 1990*. ACM Press, New York, NY, USA, 259-270.
Wutka. 2000. DTD parser. http://www.wutka.com/dtdparser.html.
YAN, T.W., AND GARCIA-MOLINA, H. 1994. Index structures for selective dissemination of information under boolean model. *ACM TODS*, 19, 2, 332-364.
YAN, T.W., AND GARCIA-MOLINA, H. 1999. The SIFT information dissemination system. *ACM TODS*, 24, 4, 529-565.

APPENDIX A: DATA STRUCTURES AND PSEUDO-CODE FOR INLINE

A.1 Data Structures for Bookkeeping

```
QueryEvaluation[ ]      queryEvalList;
class QueryEvaluation {
        boolean   isMatched;
        PredicateEvaluation[ ]      predEvalList;
}
class PredicateEvaluation {
        int       stepNumber;
        Set       elementIdentifiers;
}
```

A.2 Pseudo-code

```
QueryEvaluation[ ] queryEvalList;
Stack elementIDStack, truePredicateStatck;
Start document handler:
  if queryEvalList has not been allocated
     allocate queryEvalList;
  else
     clear all data structures in queryEvalList;
Start element handler:
  assign an element identifier elementID to this element Element;
  for each active state
    apply rule (1) to (4) to find target states (see section 3.2.4);
  endfor
  List truePredicates;
  for each target state
(1)  for each predicate P in the local predicate table of the state
       retrieve the P.QueryID-th element queryEval from queryEvalList;
       evaluate P using Element only if queryEval.isMatched is false;
       if P is evaluated to true
         retrieve the P.PredicateID-th element predEval from queryEval;
         add elementID to the set elementIdentifiers in predEval;
```

add the pair (*P.QueryID, P.PredicateID*) to *truePredicates*;

endif

endfor

if this target state is an accepting state

(2) ***for*** each query *Q* whose identifier is in the ID list at the state

if all predicates contained in *Q* have been satisfied

intersect element identifier sets of all predicates that have the same step number;

if the intersection is non-empty for every level

queryEval.isMatched = true;

endif

endif

endfor

endif

endfor

push *elementID* to *elementIDStack*;

push *truePredicates* to *truePredicateStack*;

End element handler:

pop the top element *truePredicates* from *truePredicatStack*;

pop the top element *elementID* from *elementIDStack*;

for each pair (*P.QueryID, P.PredicateID*) of predicate *P* in the list *truePredicates*

(3) retrieve the *P.QueryID*th element *queryEval* from *queryEvalList*;

if *queryEval.isMatched* is false

retrieve the *P.PredicateID*th element *predEval* from *queryEval*;

remove *elementID* from the set *elementIdentifiers* in *predEval*;

endif

endfor

Note: (1) evaluation of a predicate; (2) final evaluation of a query; (3) undo for a predicate

APPENDEX B: DATA STRUCTURES AND PSEUDO-CODE FOR SP

A.1 Data Structures for Bookkeeping

Boolean[ ] *queryEvalList*;

A.2 Pseudo-code

Boolean[ ] *queryEvalList*;

Start document handler:

if *queryEvalList* has not been allocated

allocate *queryEvalList*;

else

clear all data structures in *queryEvalList*;

endif

Start element handler:

assign an element identifier *elementID* to this element *Element*;

for each active state
apply rule (1) to (4) to find target states (see section 3.2.5);
retrieve sequences of elements for the active state by following pointers from the state in the run time stack;
append *Element* to the end of the sequences to obtain new sequences for all target states;
for each target state that is an accepting state
for each sequence of elements
for each query *Q* whose identifier is in the ID list at the state
(1) retrieve the *Q.QueryID*th element of *queryEvalList*;
if *Q* is not matched
for each predicate *P* of *Q*
retrieve an element from the sequence using *P*'s step number and evaluate *P*;
if evaluation fails
break;
endif
endfor
if all pedicates are satisfied
set the *Q.QueryID*th element of *queryEvalList* to true;
endif
endif
endfor
endfor
endfor
Note: (1) selection performed by SP.

Received July 2002; revised April 2003; accepted August 2003.