Efficient Filtering of XML Documents for Selective Dissemination of Information [*]

Mehmet Altınel
Department of Computer Science
University of Maryland
altinel@cs.umd.edu

Michael J. Franklin
EECS Computer Science Division
University of California at Berkeley
franklin@cs.berkeley.edu

Abstract

Information Dissemination applications are gaining increasing popularity due to dramatic improvements in communications bandwidth and ubiquity. The sheer volume of data available necessitates the use of selective approaches to dissemination in order to avoid overwhelming users with unnecessary information. Existing mechanisms for selective dissemination typically rely on simple keyword matching or "bag of words" information retrieval techniques. The advent of XML as a standard for information exchange and the development of query languages for XML data enables the development of more sophisticated filtering mechanisms that take structure information into account. We have developed several index organizations and search algorithms for performing efficient filtering of XML documents for large-scale information dissemination systems. In this paper we describe these techniques and examine their performance across a range of document, workload, and scale scenarios.

1 Introduction

The proliferation of the Internet and intranets, the development of wireless and satellite networks, and the availability of asymmetric, high-bandwidth links to home have fueled the development of a wide range of new *dissemination-based* (or *Selective Dissemination of Information (SDI)*) applications. These applications involve timely distribution of data to a large set of customers, and include stock and sports tickers, traffic information systems, electronic personalized newspapers, and entertainment delivery. The execution model for these applications is based on continuously collecting new data items from underlying data sources, filtering them against user profiles (i.e., user interests) and finally, delivering relevant data to interested users.

In order to effectively target the right information to the right people, SDI systems rely upon user profiles. Current SDI systems typically use simple keyword matching or "bag of words" Information Retrieval (IR) techniques to represent user profiles and match them against new data items. These techniques, however, often suffer from limited ability to express user interests, thereby raising the potential that the users receive irrelevant data while not receiving the information they need. Moreover, work on IR-based models has largely focused on the effectiveness of the profiles rather than the *efficiency* of filtering. In the Internet environment, where huge volumes of input data and large numbers of users are typical, efficiency and scalability are key concerns.

Recently, XML (eXtensible Markup Language) [BPS98, Cov99] has emerged as a standard information exchange mechanism on the Internet. XML allows the encoding of structural information within documents. This information can be exploited to create more focused and accurate profiles of user interests. Of course such benefits come at a cost, namely, an increase in the complexity of matching documents to profiles.

We have developed a document filtering system, named *XFilter*, that provides highly efficient matching of XML documents to large numbers of user profiles. In XFilter, user interests are represented as queries using the XPath language [CD99]. The XFilter engine uses a sophisticated index structure and a modified Finite State Machine (FSM) approach to quickly locate and examine relevant profiles. In this paper we describe these structures along with an event-based filtering algorithm and several enhancements. We then evaluate the efficiency, scalability, and adaptability of the approaches using a detailed experimental framework that allows the manipulation of several key characteristics of document and user profiles. The results indicate that XFilter performs well and is highly scalable. Thus, we believe our techniques represent a promising technology for

[*]This research has been partially supported by Rome Labs agreement number F30602-97-2-0241 under DARPA order number F078, by the NSF under grant IRI-9501353, and by Intel, Microsoft, NEC, and Draper Laboratories.

Proceedings of the 26th VLDB Conference,
Cairo, Egypt, 2000.

the deployment of Internet-scale SDI systems.

The remainder of the paper is organized as follows: In Section 2, we give an overview of an XML-based SDI system and the XPath language, which is used in our user profile model. Related work is discussed in Section 3. In Section 4, we present the profile index structures and an event-based XML filtering algorithm. Enhancements to this algorithm are provided in Section 5. We discuss the experimental results in Section 6. Section 7 concludes the paper.

2 Background

In this section we first present a high-level architecture of an XML-based information dissemination system. We then describe the XPath language, which we use to specify user profiles in XFilter.

2.1 An XML-based SDI Architecture

The process of filtering and delivering documents based on user interests is sometimes referred to as Selective Dissemination of Information (SDI). Figure 1 shows a generic architecture for an XML-based SDI system. There are two main sets of inputs to the system: user profiles and data items (i.e., documents). User profiles describe the information preferences of individual users. In most systems these profiles are created by the users, typically by clicking on items in a Graphical User Interface. In some systems, however, these profiles can be learned automatically by the system through the application of machine learning techniques to user access traces. The user profiles are converted into a format that can be efficiently stored and evaluated by the Filter Engine. These profiles are "standing queries", which are (conceptually) applied to all incoming documents.

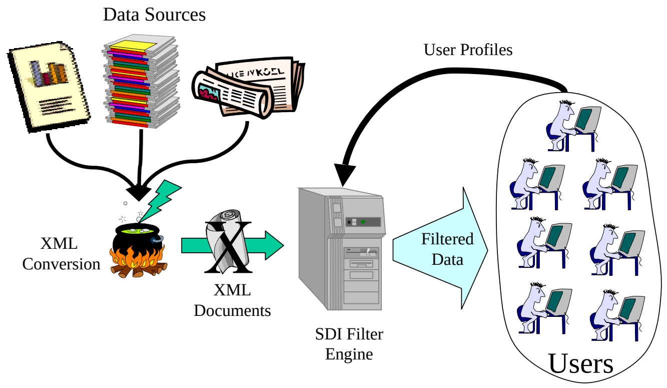

Figure 1: Architecture of an XML Based SDI System

The other key inputs to an SDI system are the documents to be filtered. Our work is focused on XML-encoded documents. XML is a natural fit for SDI because it 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 contents in order to better describe their 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). A DTD specifies the elements and attributes names and the nature of their content in the document.

In an SDI system, newly created or modified XML documents are routed to the Filter Engine. When a document arrives at the filter engine, it is matched against the user profiles to determine the set of users to whom it should be sent. As SDI 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 potentially huge set of user profiles to find those for which the document is relevant. XFilter is aimed at solving exactly this problem. Before presenting the solutions used in XFilter, however, we first describe a model for expressing user profiles as queries of XML documents.

2.2 XPath as a Profile Language

The profile model used in XFilter is based on XPath [CD99], a language for addressing parts of an XML document that was designed for use by both the XSL Transformations (XSLT) [Cla99b] and XPointer [DDM99] languages. XPath provides a flexible 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 either 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*. In the simplest and most common form, a location step specifies a node name (i.e., an element name).[1] The hierarchical relationships between the nodes are specified in the query using parent-child ("/") operators (i.e., at adjacent levels) and ancestor-descendant ("//") operators (i.e., separated by any number of levels). For example the query `/catalog/product//msrp` addresses all `msrp` element descendants of all `product` elements that are direct children of the `catalog` (root) element in the document. XPath also allows the use of a wildcard operator ("*"), which matches any element name, at a location step in a query.

Each location step can also include one or more *filters* to further refine the selected set of nodes. A filter is a predicate that is applied to the element(s) addressed at that location step. All the filters at a location step must evaluate to

[1] The full XPath specification [CD99] contains many more options. We do not list them all here due to space considerations.

TRUE in order for the evaluation to continue to the descendant location steps. Filter expressions are enclosed by "[" and "]" symbols. The filter predicates can be applied to the text of the addressed elements or the attributes of the addressed elements and may also include other path expressions. Any relative paths in a filter expression are evaluated in the context of the element nodes addressed in the location step at which they appear. For example, consider the query: `//product[price/msrp<300]/name`. This query selects the `name` elements of the XML document if the `msrp` of the product is less than 300. Here, the path expression `price/msrp` in the filter is evaluated relative to the `product` elements. This example also shows how element contents can be examined in the queries.

In XFilter, XPath is used 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.

An alternative to using XPath would be to use one of the query languages that have been proposed for semi-structured data such as UnQl [BDHS96], Lorel [AQM+97] or XML-QL [DFF+98]. We chose to use XPath in our work for two reasons: First, we did not need the full functionality of such query languages for our document filtering purposes. In particular, XFilter examines one document at a time, so only path expressions over individual documents are needed. Secondly, the XPath specification is a World Wide Web Consortium (W3C) recommendation, which means W3C considers it appropriate for widespread deployment. In contrast, the standardization process for XML Query Languages is still in progress. Be that as it may, the techniques described in this paper are largely applicable to path expressions in general, and thus, we believe that they can be adapted to suit other languages as the need arises.

3 Related Work

User profile modeling and matching have been extensively investigated by the Information Retrieval community in the context of Information-Filtering (IF) and SDI research (e.g., [AAB+98, FZ98, FD92]). IR-style user profiles are intended for unstructured text-based systems and typically use sets of keywords to represent user interests.[2] 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 the profiles and documents are assigned a similarity value. A document whose similarity to a profile exceeds a certain threshold is said to match the profile. The Vector Space Model [CFG00, Sal89] and statistical approaches (e.g., [BC92]) are examples of similarity-based techniques. The Stanford Information Filtering Tool (SIFT) [YM94, YM95] is a text filtering system for Internet News articles that is based on keywords. SIFT originally used Boolean profiles but was later changed to use a Vector Space approach.

[2] Several recent text retrieval methods aim to take structure information into account for text databases [BN96]. These methods, however, are not as mature as the ones described above and they have not been studied in the context of SDI.

Our profile model differs from the IR-based work in SDI in the following ways:

- The application domain of IR-based SDI systems involves only text documents, whereas our system can work for any application domain in which data is tagged using XML.
- Our profile language takes advantage of embedded schema information in the XML documents, providing more precise filtering than is available using only keywords.
- With the notable exception of the SIFT project, work on IR-based models has largely focused on the effectiveness of the profiles rather than the efficiency of filtering [VH98]. For an Internet-scale filtering system, efficiency and scalability are of paramount importance.

Query-based profile models have also been studied by the database community in the context of Continuous Queries (CQ), which are standing queries that allow users to get new results whenever an update of interest occurs in a database. Early work on CQ for relational databases was done by Terry et al. [TGNO92]. More recently, OpenCQ [LPT99] and NiagaraCQ [CDTW00] have been proposed for information delivery on the Internet. The scalability of these systems is fundamentally limited because they apply all standing queries (or at least all non-equivalent ones) to delta increments or to the updated database for each update that arrives at the system. For Internet-scale systems with potentially millions of users, such approaches are simply not feasible. NiagaraCQ provides some measure of scalability by grouping exactly equivalent queries, but does little to improve the efficiency of matching XML documents to profiles beyond that optimization.

The key insight to building high-performance, scalable SDI systems is that in such systems, the roles of queries and data are reversed [YM94]. In a database system large numbers of data items are indexed and stored, and queries are individually applied. In contrast, in SDI systems, large numbers of queries are stored, and the documents are individually matched to the queries. Thus, in an SDI system, it is necessary to *index the queries*. XFilter is unique in that it combines the scalable SDI approach of indexing queries with the ability to reference document structure (i.e., schema information) leading to scalable but precise filtering of documents for Internet-scale systems.

Triggers [SJGP90, WF89, MD89] in traditional database systems are similar to CQ. However, triggers are a more general mechanism which 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 [HCH+99], however, this work has not addressed the XML-related issues that XFilter handles.

Finally, the C3 project [CAW98] is related to XFilter in that a query subscription service is provided to subscribe to changes in semi-structured information sources. To date this work has focused on developing semantics and basic mechanisms for querying the changes in semi-structured data, rather than on the efficient evaluation of new data items against large numbers of user profiles.

4 XFilter Implementation

In this section, we describe the basic data structures and algorithms used to implement XFilter. We begin by presenting an overview of the XFilter architecture. This architecture, which is depicted in Figure 2, follows the basic SDI architecture presented earlier. The major components include: 1) an event-based parser for incoming XML-encoded documents; 2) an XPath parser for user profiles; 3) the filter engine, which performs the matching of documents and profiles; and 4) the dissemination component, which sends the filtered data to the appropriate users.

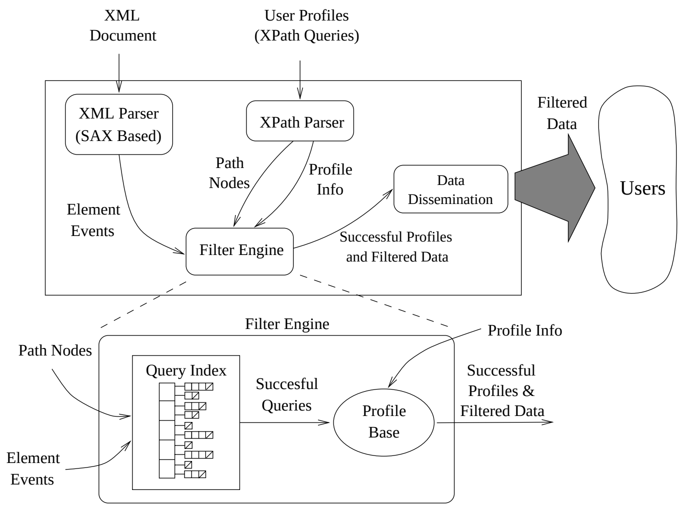

Figure 2: Architecture of XFilter

The heart of the system is the Filter Engine, which uses a sophisticated index structure and a modified Finite State Machine (FSM) approach to quickly locate and check relevant profiles. We first describe the Filter Engine and how it stores the profile information it receives from the XPath parser. The process of checking profiles is driven by an *event-based* XML parser. When an XML document arrives at the system, it is run through the parser, which sends "events" that are responded to by handlers in the filter engine. This process is described in Section 4.2.

Once the matching profiles have been identified for a document, the document must be sent to the appropriate users. The current implementation of XFilter simply uses unicast delivery and sends the entire document to each interested user. Our future work involves the integration of a variety of delivery mechanisms as investigated in our previous work [AAB+98, AAB+99], and the delivery of partial documents. These issues are beyond the scope of this current paper and so are not addressed further here.

4.1 Filter Engine

An XML-based profile model needs efficient algorithms for structure and data filtering to achieve high performance in a large-scale environment such as the Internet. As a result, profile grouping and indexing are crucial for large-scale XML document filtering. For this purpose, similar to traditional SDI systems, the Filter Engine component of XFilter contains an *inverted index* [Sal89], called the *Query Index* (See Figure 2). The Query Index is used to match documents to individual XPath queries. Our implementation also allows for user profiles to be expressed as boolean combinations of XPath queries rather than being restricted to a single XPath query. Such composite profiles are handled by post-processing the matching results at the *Profile Base* to check the boolean conditions. In this paper, due to space limitations, we focus on profiles consisting of a single XPath query.

Filtering XML documents using a structure-oriented path language such as XPath (as opposed to keyword matching) introduces several new problems that must be addressed:

1. Checking the order of the elements in the profiles.
2. Handling wildcards and descendant operators in path expressions.
3. Evaluating filters that are applied to element nodes.

In order to handle these problems efficiently, XFilter converts each XPath query to a *Finite State Machine* (FSM). The events that drive the execution of the Filter Engine are generated by the XML Parser (as described in the following section). In the XFilter execution model, a profile is considered to match a document when the final state of its FSM is reached. The Query Index is built over the states of the XPath queries.

For ease of exposition, we initially describe a solution that addresses the first two problems above (i.e., order checking and wildcard/descendant operators), but only partially addresses the third problem (element node filters). As described in Section 2.2, node filters can themselves contain path expressions, resulting queries with nested path expressions. The handling of element node filters that contain path expressions incurs significant additional complexity. Thus, we first describe a solution that works for filters that do not contain path expressions (e.g., predicates on attribute values or node contents) and postpone the discussion of our solution to handle nested path expressions until Section 4.3.

The main structures used in the Filter Engine are depicted in Figure 3. Each XPath query is decomposed into a

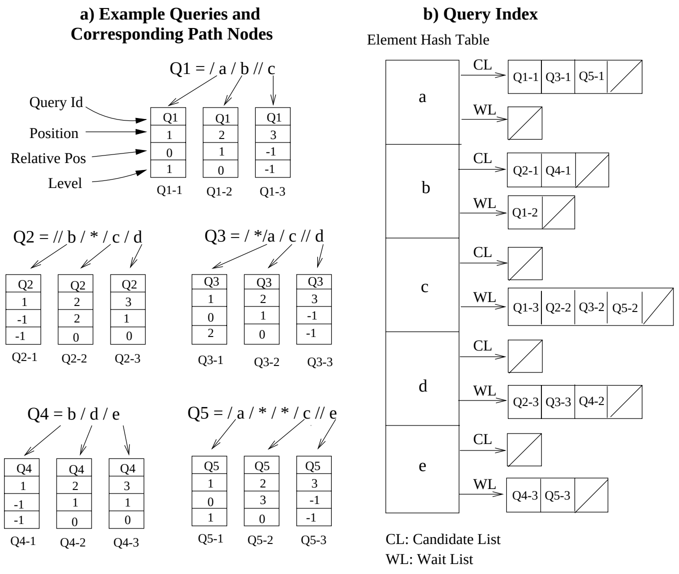

Figure 3: Path Node Decomposition and the content of the Query Index

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.

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 a descendant 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.

Filters: If a node contains one or more filters, these are stored as expression trees pointed to by the path node.

NextPathNodeSet: Each path node also contains pointer(s) to the next path node(s) of the query to be evaluated. In the restricted case where filters do not include path expressions, there is at most one pointer. Nested path expressions may raise the need for pointers to additional nodes.

Figure 3(a) shows how five example XPath expressions are converted into path nodes by the XPath parser. These nodes are then added to the Query Index. As shown in Figure 3(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*.

Since each query can only be in a single state of its FSM at a time, each query has a single path node that represents its current state. We refer to this node as the “current node”. The current node of each query is placed on the Candidate List of the index entry for its respective element name. All of the path nodes representing future states are stored in the Wait Lists of their respective element names. A state transition in the FSM of a query is represented by promoting a path node from the Wait List to the Candidate List.

The initial distribution of the path nodes to these 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, which are described in Section 5. Figure 3(b) shows the most straightforward case, where the path nodes for the initial states are placed on the Candidate Lists.

4.2 XML Parsing and Filtering

When a document arrives at the Filter Engine, it is run through an XML Parser which then drives the process of checking for matching profiles in the Index. We use an XML parser that is based on the SAX interface, which is a standard interface for *event-based* XML parsing [Meg98]. We developed the parser using the *expat* toolkit [Cla99a], which is a non-validating XML processor.

The SAX event-based interface reports parsing events (such as encountering the start or end tag of an element) directly to the application through callbacks, and does not usually build an internal tree. To use the SAX interface, the application must implement handlers to deal with the different events, much like handling events in a graphical user interface. For our application, we use the events to drive the profile matching process. Figure 4 shows an example of how a SAX event-based interface breaks the structure of an XML document down into a linear sequence of events.

For XFilter, we implemented callback functions for the parsing events of encountering: 1) a begin element tag; 2) an end element tag; or 3) data internal to an element. All of the handlers are passed the name and document level of the element for (or in) which the parsing event occurred. Ad-

An XML Document	SAX API Events
<?xml version="1.0"> <doc> <para> Hello, world! </para> </doc>	start document start element: doc start element: para characters: Hello, world! end element: para end element: doc end document

Figure 4: SAX API Example

ditional handler-specific information is also passed as described below.

Start Element Handler: When an element tag is encountered by the parser, it 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 two checks: a level check and an attribute filter 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. The attribute filter check applies any simple predicates that reference the attributes of the element.

If both checks succeed and there are no other filters to be checked, 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.

End Element Handler: When an end element tag is encountered, the corresponding path node is deleted from the Candidate List in order to restore that list to the state it was in when the corresponding start element tag was encountered. This "backtracking" is necessary to handle the case where multiple elements with the same name appear at the different level in the document.

Element Characters Handler: This handler is called when the data associated with an element is encountered. The data is passed in to the handler as a parameter. It works similarly to the Start Element Handler except that it performs a content filter check rather than an attribute filter check. That is, it evaluates any filters that reference the element content. Like the Start Element Handler, this handler can also cause the query to move to its next state.

4.3 Handling Nested Path Expressions

Recall that the description above was simplified by excluding the processing of element node filters that contain path expressions. Such nested path expressions complicate the filtering process because they introduce non-linearity into the paths. Our implementation of XFilter fully supports such nested path expressions. Due to space limitations, however, we only outline the basic approach to how such expressions are handled.

When the XPath parser encounters an element node filter in a query, it converts it to an expression tree and stores it in the current path node. If the filter contains an XPath query, then that nested query is treated like a separate query from the one it is embedded in. That is, the XPath parser decomposes it into series of path nodes and assigns it a Query ID. A leaf node for the nested query is created in the expression tree of the filter so that its result can be used to evaluate the filter expression. If the filter query starts with an absolute path (i.e. '/' or '//'), that is all that needs to be done. Otherwise, its first path node has to be inserted into the NextPathNodeSet of the current path node to provide relative execution. By doing so, the filter query will be examined relative to current node since its first path node will be copied to Candidate List after the current path node is processed.

If, when evaluating a filter, the result of a nested path expression in that filter is not yet known, we allow the execution for the current path node to continue as if the filter succeeded. At the same time, the filter is marked to be reevaluated when the XML document parsing is finished as the results of all the queries are available at that point. If a query contains nested path expressions that must be evaluated in this manner, it is not considered successful until all of its marked filters are determined to be successful. Likewise, if a query fails after marking a filter, this filter mark is cleared by the system so that the filter need not be reevaluated.

5 Enhanced Filtering Algorithms

In this section we describe several enhancements to the *basic* filtering algorithm described in Section 4. As described in that section, XFilter's basic approach is actually far from "basic" as it incorporates sophisticated indexing and FSM-based evaluation mechanisms. These mechanisms are not strictly necessary to perform SDI filtering, but rather, were developed solely to enhance performance and scalability. Thus, before extending the basic approach it is important to highlight its potential benefits over other, perhaps simpler approaches.

There are two fairly obvious *brute force* strategies that could be employed for performing SDI filtering of XML documents. The first strategy, which is similar to the method used by the database-oriented CQ systems, is to store each profile (query) in an unindexed fashion. When an XML document arrives at the Filter Engine, the document is parsed and indexed. The CQ tool then iterates over all of the profiles, matching them against the document using the document index. This approach is easy to implement and has the benefit of using existing XML search tools, but it is obviously inappropriate for a system with many users, as all profiles must be examined for each new document.

A second brute force strategy applies previous work on keyword-based filtering such as [YM94, YM95] and indexes the profiles using the text, element names, and attribute names that appear in them as keywords. When a new document arrives, this index is used to locate candidate profiles that may possibly be satisfied by the document. These profiles are then checked against the document (i.e., for ordering constraints, etc.) sequentially in a second pass. While this approach is likely to perform better than the first brute force approach, it suffers from the need to perform expensive checks of the document for all candidate profiles.

The main drawback of the brute force methods is that for each input document, they must invoke the profiles individually. XFilter avoids the pitfalls of the brute force approaches by being more careful in the identification of candidate profiles using specialized indexing structures and by more efficiently evaluating those candidate profiles. We now describe two enhancements to the basic XFilter approach: List Balancing and Prefiltering. The performance of XFilter and these enhancements is then examined in Section 6.

5.1 List Balancing

In the basic approach, the Query Index is constructed by simply placing the first path node of each XPath query in the Candidate List for its corresponding element name, and placing the remaining path nodes in the Wait Lists as was shown in Figure 3. 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 which 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 5 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 we will see in the experiments that follow, this additional cost is far outweighed by the benefits of List Balancing.

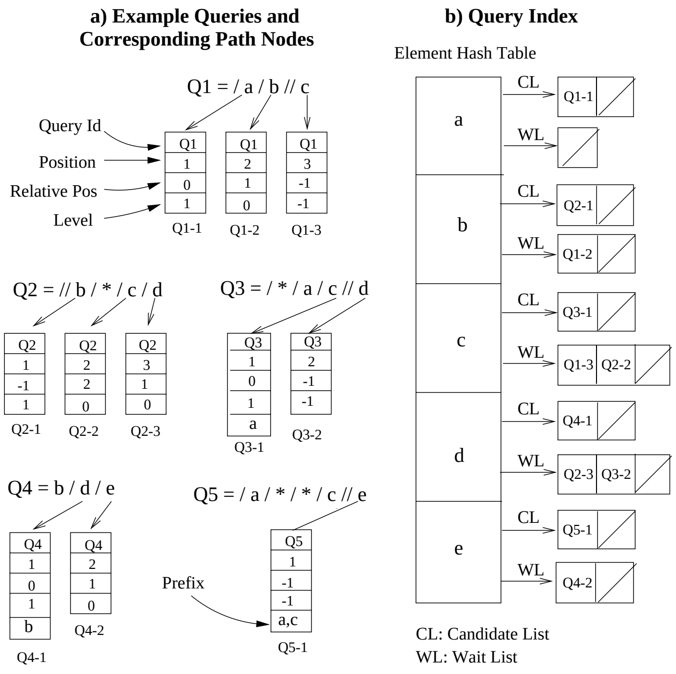

Figure 5: Path Nodes and the content of the Query Index in List Balance

5.2 Prefiltering

Another potential problem with the basic approach is that it proceeds through a path expression one level at a time. Therefore, a considerable amount of unnecessary work may be done for queries that fail due to missing elements late in the evaluation of the path. The idea of Prefiltering is to eliminate from consideration, any query that contains an element name that is not present in the input document. Prefiltering is implemented as an initial pass that is performed before order and filter checking. Thus, in this technique, each incoming document is parsed twice.

Fortunately, previous algorithms developed for the filtering of plain text documents can be used for this purpose. We employed Yan and Garcia-Molina's *Key Based* algorithm [YM94] since it has been shown to be efficient and it fits well with the existing structures used in XFilter. In this method, each query, when initially parsed, is assigned a "key" element name chosen from the element names it contains. In XFilter the key element is chosen using the same algorithm described above for choosing initial nodes in List Balancing.

When a document arrives, an *occurrence table* is constructed, which is a hash table containing an entry of each element name that appears in the document. The entry for an element name in an occurrence table contains a list of all queries whose key is that element name. Once the table has been constructed, the queries referenced by the table are checked to see if all of the element names they contain are in the document. Then, the successful queries are checked further using the normal Basic or List Balance algorithm. That is, the document is parsed a second time during which only queries that have passed the prefiltering step are considered.

The Prefiltering pass introduces an additional cost to the query processing, but can reduce the number of profiles checked by the basic or List Balancing algorithms. The benefits of the Prefiltering method depend on the selectivity of the first step. When the first step discards only small number of profiles, then the advantage of having prefiltering can turn into a disadvantage. We examine the performance of Prefiltering in the next section.

6 Performance Analysis

In this section we evaluate the performance of the basic filtering algorithm and its enhancements. We examine four algorithms: basic, list balance, basic with prefiltering, and list balance with prefiltering.

6.1 Experimental Environment

We implemented XFilter using Gnu C++ version 2.8.1. The experiments were conducted on a Sun Ultra-5 workstation with 128MB memory running Solaris 2.6. All structures are kept in memory in the experiments.

We created our benchmark using the NITF (News Industry Text Format) DTD [Cov99]. The NITF DTD is intended for news copy production, press releases, wire services, newspapers, broadcasters, and Web-based news organizations. It was developed as a joint standard by news organizations and vendors worldwide, and it is supported by most of the world's major news agencies. It is already in use in several commercial applications. For example, the NewsPack product by Wavo corporation [New00] delivers real-time news and information over the Internet in XML format using NITF. The NITF DTD contains 158 elements organized in 7 levels with 588 attributes.

We generated XML documents for our experiments using IBM's XML Generator tool [IBM99], which is a Java program designed to automate creating test cases for XML applications. This tool generates *random* instances of valid XML documents from a single input DTD according to user-provided constraints. In order to create user profiles, we implemented a query generator that takes a DTD as input and creates set of XPath queries based on input parameters similar to IBM's XML Generator. We describe our use of these tools for workload generation in the following section. In the experiments each user profile contains a single XPath query.

We created different workloads by changing the parameters of the document and query generators. For each experiment, we first generated a set of profiles and created the Query Index and other structures from them. Then, we ran the XML generator to produce a random XML document and submitted that document to the system. We measured the "filter time" as the total time to find all matching profiles — the costs of creating the document and profiles and of sending the document to the users are not included in this metric. For each experimental setting we generated and filtered XML documents until the 90% confidence intervals for the measured filter times were within plus or minus 3% of the mean.

6.2 Workload Parameters

Descriptions of the parameters used in the experiments and their value ranges are shown in Table 1. P denotes the number of profiles in the Query Index, which is used to measure the scalability of the system in terms of number of users. The *maximum* depth (i.e., the level number of the lowest level) of the XML document and XPath queries is denoted by D. For a given experiment, D is set to the same value for both documents and queries. However, due to differences in the way the generators work, the actual number of levels in the documents and queries tends to be different. The document generator always starts from the root of the DTD, while the query generator may start at any level depending on which element node it initially chooses. Also, the document generator always includes elements that are identified as "required" in the DTD, so it sometimes generates documents that are deeper than the maximum depth. Thus, for a given value of D the average depths of the documents tend to be larger than the average depths of the queries. Table 2 shows the mean depth values of the documents and queries with various D values. Note that the average depth of a document is 2 for input level 1 due to the presence of required elements at level 2 in the NITF DTD.

Parameter	**Range**	**Description**
P	1,000 to 100,000	Number of Profiles
D	1 to 10	Maximum depth of the XML document and queries
W	20% to 80%	Probability of a wildcard ('*') in the element nodes of the queries
F	0 to 3	Level of the element node filter in the queries. 0 means there is no element node filter.
S	1% to 100%	Selectivity of the element node filter
θ	0 and 1	Skewedness of element names in query generation

Table 1: Workload Parameters

Input Max Depth	Avg Doc Depth	Avg Query Depth (Uniform)	Avg Query Depth (Skewed)
1	2	1	1
2	2	2	2
3	2.99	2.65	2.64
4	3.63	3.04	2.96
6	4.36	3.34	3.21
8	4.76	3.4	3.23
10	5.09	3.42	3.24

Table 2: Mean Depth of Workload Document and Queries

Four additional parameters are used to help shape the query workload. W is the probability that a given element node in a query will be a wildcard operator. F and S are used to control the presence and characteristics of filters in the queries. F determines which level of a query (if any) will contain a filter, and S specifies the selectivity of such a filter if it does exist. Finally θ is the parameter of the zipf distribution [Zip49] that is used to determine the skewedness of the choice of element names at each level in query generation. When it is 0, each element name in the query is selected randomly from the set of element names allowed at its level with a uniform distribution, whereas at a setting of 1, the choice is highly skewed. Note that all of the documents are generated with a uniform distribution of element names as provided by the IBM's XML generator.

6.3 Analysis of Experimental Results

We now describe the results of four experiments that investigate the performance of the filtering algorithms for varying 1) number of profiles; 2) depth of queries and documents; 3) probability of wildcards; and 4) filter placement and selectivity.

Experiment 1: Varying P (D=5, W=0, F=0)
In this set of experiments we measure the filter time of the algorithms as the number of profiles in the system is increased. For the results shown we fixed the maximum depth of the input XML documents and queries to 5. Figure 6 shows the results when element names used in queries are chosen with a uniform distribution. As expected, the Basic method has the lowest performance. List Balance provides some improvement, but Prefiltering dramatically improves the performance of both algorithms since most of the profiles are filtered out in the prefiltering step. In this experiment, on average, 2.6% of profiles matched a given document. By itself, the Basic algorithm examined about 12% of the profiles in order to find these. In contrast, when prefiltering was applied, only 3.5% of the profiles were examined in the second phase. In this case (and in virtually all others we have studied) the combination of List Balance and Prefiltering provided the best performance. Here, with 100,000 profiles in the system, that combination was over 5 times faster than the basic approach.[3]

[3] Note that we were able to run experiments on our (very modest) hardware configuration with at most 100,000 queries since increasing beyond that point led to swapping, which distorted the results.

The results of running this experiment with the skewed selection of elements are shown in Figure 7. In this case, the execution times are higher for all of the algorithms since more profiles are examined and more match the document due to the element skew. In this case the benefits of Prefiltering are less dramatic than in the uniform case and in fact, List Balance performs substantially better than the combination of Prefiltering and Basic. With element selection skew, the profiles tend to be more similar so the selectivity of prefiltering is lower. In contrast, the List Balance enhancement is highly effective here, as it evens out the distribution of profiles to Candidate Lists. Again in this case, the combination of List Balance and Prefiltering performs the best, although here there is only a slight advantage over List Balance.

Experiment 2: Varying D (P=50,000, W=0, F=0)
The depth of the XML documents and queries in user profiles varies according to application characteristics. In this experiment we evaluated the performance of the algorithms as the maximum depth is varied. Here, we fixed the number of profiles at 50,000 and varied the maximum depth of the XML document and queries from 1 to 10. At each step, the same depth value is used in generating the document and queries.

Figure 8 shows the filter time as D is increased in the case that the element names in queries are selected uniformly. The filter time increases for all the algorithms because the input document contains more elements with each additional level resulting in more checking of path nodes, and because the queries become larger. Again, Basic performs worst while the combination of List Balance and Prefiltering performs best. One interesting aspect of this graph is that beyond a depth of 8, the List Balance and Basic with Prefiltering lines cross. This happens because the increase in levels decreases the effectiveness of prefiltering (more opportunities for element names to appear in queries) while List Balancing benefits slightly by having more choices for the pivot elements.

For the skewed case (Figure 9), there is a sharp increase in the filtering time of Basic because as the number of levels increases, more and more *popular* elements appear in the document, boosting the filtering time. List Balance has a smaller increase compared to Basic, thanks to the presence of less popular elements in the queries which can be used as pivot nodes. After level 4, the presence of element names in the queries does not change much because of skewed distribution, hence the workload characteristics remain similar. As a result, the filtering times of the algorithms increase just slightly. At level 1 and 2 prefiltering discards virtually none of the profiles, so the prefiltering algorithms have worse performance than the others at these points. Starting from level 3, prefiltering becomes more effective due to the presence of less popular elements in queries resulting in better filter time. These experiments indicate that the combination of List Balance and Prefiltering can adapt well to different workload characteristics.

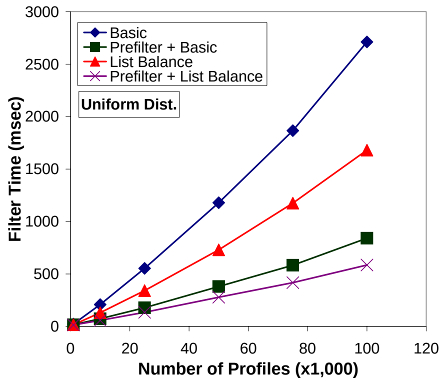

Figure 6: Uniform Dist. Varying P
(D=5, θ=0, W=0, F=0)

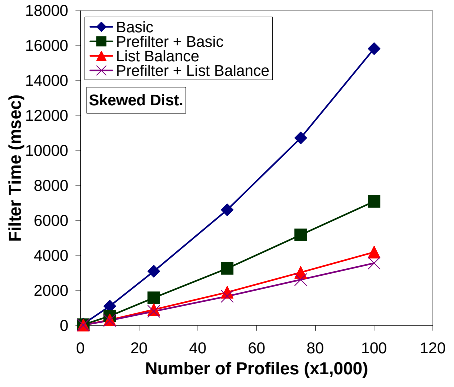

Figure 7: Skewed Dist. Varying P
(D=5, θ=1, W=0, F=0)

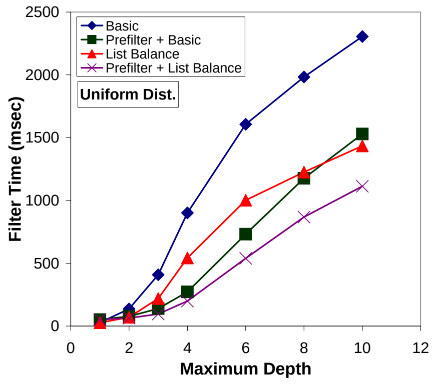

Figure 8: Uniform Dist. Varying D
(P=50,000, θ=0, W=0, F=0)

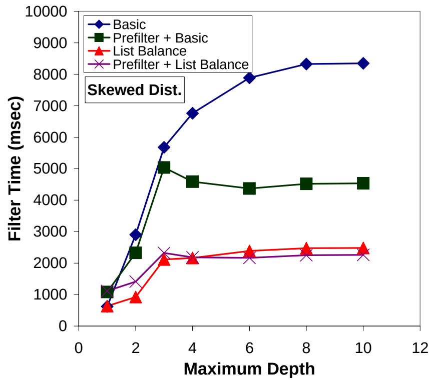

Figure 9: Skewed Dist. Varying D
(P=50,000, θ=1, W=0, F=0)

Experiment 3: Varying W (P=50,000, D=6, F=0)
In this experiment we evaluate the effect of the number of wildcards ('*') that are likely to occur in the queries. We performed this experiment for 50,000 profiles, and set the maximum depth of queries and documents to 6. In each step of the experiment, we varied the probability that an element node may be a wildcard. Figure 10 shows the filtering times of the algorithms as the probability of wildcards is increased. The important result in this experiment is that, with Prefiltering, the algorithms are relatively insensitive to wildcards, while without prefiltering, they are quite sensitive to them. This is because as more wildcards are introduced, the selectivity of prefiltering drops, but the work done in the second step also decreases. List Balance has slightly better performance than List Balance with Prefiltering when the wildcard probability is very high (>60%), but it is unlikely that many profiles will have such a high proportion of wildcards.

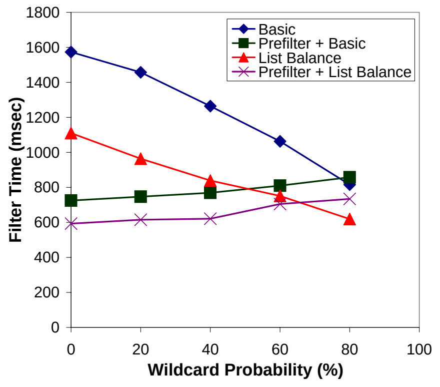

Figure 10: Varying Wildcard Probability
(P=50,000, D=6, θ=0, F=0)

Experiment 4: Varying F and S (P=50,000, D=6, W=0)
We performed two experiments to find out the effect of element node filters in the queries. In particular, we examined the effect of the level of the filter and its selectivity on filter time. For this purpose, we modified the NITF DTD and added a fixed attribute named *dummy* to every element. Then, in the queries we created a simple element node filter containing only that fixed attribute. We adjusted the selectivity of the element node filters by changing the appearance probability of *dummy* in the input document using a parameter (called fixedOdds) of the XML document generator.

In the first experiment, we placed a single element node filter in different levels of the query and fixed the query selectivity at 10%. We performed the experiment with 50,000 profiles a maximum depth of 6; No wildcards were used in the queries. The results of this experiment are shown in Figure 11. All the algorithms benefit from the element node filter when it is in the upper levels of the queries as in such cases most of the queries are filtered out in their early level checks. As we move the element node filter to deeper levels, its effect diminishes because the path length of some queries is less than the filter level (so they do not have a filter).

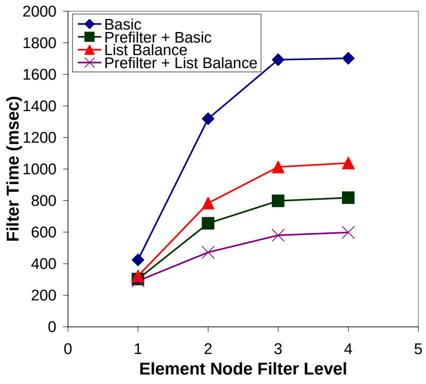

Figure 11: Varying Filter Level
(P=50,000, D=6, θ=0, W=0, S=10)

In the second experiment, we fixed the element node filter at level 2 and varied its selectivity. We assigned selectivity values in logarithmic scale to focus on the behavior of the algorithms when the filter is highly selective. As shown in Figure 12, the selectivity of the element node filter has a relatively small effect on the algorithms and affects all of them to almost the same degree. The slope of the Basic algorithm is a bit sharper than others as it has much worse performance than the others when the effect of the filter diminishes.

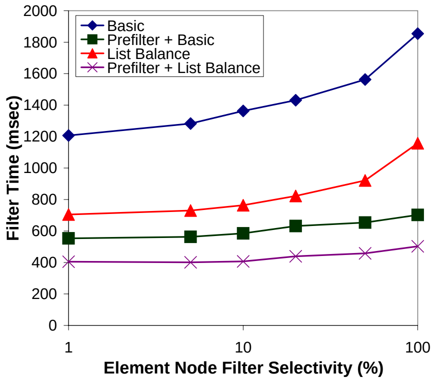

Figure 12: Varying Filter Selectivity
(P=50,000, D=6, θ=0, W=0, F=2)

Summary of Results:
These experiments demonstrate the scalability of the XFilter approach and show that the extensions we proposed for Basic provide substantial improvements to the performance in different document, workload and scale scenarios. In particular, List Balance with Prefiltering has the best filtering performance in virtually all cases. List Balance is also effective by itself when the distribution of elements in queries is highly skewed. Since many SDI applications exhibit such skew, and because List Balance is simpler and requires less space than List Balance with Prefiltering, it may be preferable in many practical cases.

7 Conclusions

In this paper, we have proposed an XML document filtering system, called *XFilter*, for Selective Dissemination of Information (SDI). XFilter allows users to define their interests using the XPath query language. This approach enables the construction of more expressive profiles than current IR-based profile models by exploiting the structural information available in XML documents.

We developed indexing mechanisms and matching algorithms based on a modified Finite State Machine (FSM) approach that can quickly locate and evaluate relevant profiles. By converting XPath queries into a Finite State Machine representation, XFilter is able to (1) handle arbitrary regular expressions in queries, (2) efficiently check element ordering and evaluate filters in queries, and (3) cope with the semi-structured nature of XML documents. We described a detailed set of experiments that examined the performance of the basic XFilter approach and its extensions. The experiments showed that XFilter is effective for different document, workload and scale scenarios, which makes it suitable for use in Internet-scale SDI systems.

XFilter has been implemented in the context of the Dissemination-Based Information Systems (DBIS) project [AAB+99]. This project is developing a toolkit for constructing adaptable, application-specific middleware that incorporates multiple data delivery mechanisms in complex networked environments. We intend to integrate XFilter as the primary filtering mechanism for the toolkit.

Acknowledgments. We would like to thank Fatma Özcan for her useful comments on earlier drafts of the paper.

References

[AAB+98] D. Aksoy, M. Altinel, R. Bose, U. Cetintemel, M. Franklin, J. Wang, S. Zdonik, ”Research in Data Broadcast and Dissemination”, *Proc. 1st Intl. Conf. on Advanced Multimedia Content Processing*, Osaka, Japan, November, 1998.

[AAB+99] M. Altinel, D. Aksoy., T. Baby, M. Franklin, W. Shapiro, S. Zdonik, ”DBIS Toolkit: Adaptable Middleware for Large Scale Data Delivery” (Demo Description), *Proc. ACM SIGMOD Conf.*, Philadelphia, PA, June, 1999.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, ”The Lorel Query Language for Semistructured Data”, *International Journal on Digital Libraries*, 1(1):68–88, April, 1997.

[BC92] N. J. Belkin, B. W. Croft, ”Information filtering and information retrieval: Two sides of the same coin?”, *CACM*, 35(12):29–38, December 1992.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu, ”A Query Language and Optimization Techniques for Unstructured Data”, *Proc. ACM SIGMOD Conf.*, Montreal, Canada, June, 1996.

[BN96] R. Baeza-Yates, G. Navarro, ”Integrating Contents and Structure in Text Retrieval”, *ACM SIGMOD Record*, 25(1):67-79, 1996.

[BPS98] T. Bray, J. Paoli, C. M. Sperberg-McQueen, ”Extensible Markup Language (XML) 1.0”, *http://www.w3.org/TR/REC-xml*, February, 1998.

[CAW98] S. Chawathe, S. Abiteboul, J. Widom., ”Representing and Querying Changes in Semistructured Data”, *Proc. 14th ICDE*, Orlando, Florida, February 1998.

[CD99] J. Clark, S. DeRose, ”XML Path Language (XPath) Version 1.0”, W3C Recommendation, *http://www.w3.org/TR/xpath*, November, 1999.

[CDTW00] J. Chen, D. DeWitt, F. Tian, Y. Wang, ”NiagaraCQ: A Scalable Continuous Query System for Internet Databases”, *Proc. ACM SIGMOD Conf.*, Dallas, TX, May, 2000.

[CFG00] U. Cetintemel, M. Franklin, C. L. Giles, ”Self-Adaptive User Profiles for Large Scale Data Delivery”, *Proc. 16th ICDE*, San Diego, February, 2000.

[Cla99a] J. Clark, ”expat - XML Parser Toolkit”, *http://www.jclark.com/xml/expat.html*, 1999.

[Cla99b] J. Clark, ”XSL Transformations (XSLT) Version 1.0”, *http://www.w3.org/TR/xslt*, November, 1999.

[Cov99] R. Cover, ”The SGML/XML Web Page”, *http://www.oasis-open.org/cover/sgml-xml.html*, December, 1999.

[DDM99] S. DeRose, R. Daniel Jr., E. Maler, ”XML Pointer Language (XPointer)”, *http://www.w3.org/TR/WD-xptr*, December, 1999.

[DFF+98] A. Deutsh, M. Fernandez, D. Florescu, A. Levy, D. Suciu, ”XML-QL: A Query Language for XML”, *http://www.w3.org/TR/NOTE-xml-ql*, August, 1998.

[FZ98] M. Franklin, S. Zdonik, ”“Data in Your Face”: Push Technology in Perspective”, *Proc. ACM SIGMOD Conf.*, Seattle, WA, June, 1998.

[FD92] P. W. Foltz, S. T. Dumais, ”Personalized information delivery: an analysis of information filtering methods”, *CACM*, 35(12):51–60, December 1992.

[HCH+99] E. N. Hanson, C. Carnes, L. Huang, M. Konyola, L. Noronha, S. Parthasarathy, J. B. Park, A. Vernon, ”Scalable Trigger Processing”, *Proc. 15th ICDE*, pp. 266-275, Sydney, Australia, 1999.

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

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

[MD89] D. McCarthy, U. Dayal, ”The Architecture of an Active Database Management System”, *Proc. ACM SIGMOD Conf.*, pp. 215-224, May, 1989.

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

[New00] Newspack, The Wavo Corporation, *http://www.wavo.com*, 2000.

[Sal89] G. Salton, ”Automatic Text Processing”, *Addison Wesley*, 1989.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, S. Potamianos, ”On Rules, Procedures, Caching and Views in Data Base Systems”, *Proc. ACM SIGMOD Conf.*, pp. 281-290, 1990.

[TGNO92] D. B. Terry, D. Goldberg, D. A. Nichols, B. M. Oki, ”Continuous queries over append-only databases”, *Proc. ACM SIGMOD Conf.*, pp. 321–330, June 1992.

[VH98] E. Voorhees, D. Harman, ”Overview of the Seventh Text REtrieval Conference (TREC-7)”, *NIST*, Gaithersburg, Maryland, November, 1998.

[YM94] T. W. Yan, H. Garcia-Molina, ”Index Structures for Selective Dissemination of Information Under Boolean Model”, *ACM TODS*, 19(2):332–364, 1994.

[YM95] T. W. Yan, H. Garcia-Molina. ”Sift - A tool for wide-area information dissemination”. *Proc. of the 1995 USENIX Tech. Conf.*, pp. 177-186, 1995.

[WF89] J. Widom, S. J. Finklestein, ”Set-Oriented Production Rules in Relational Database Systems”, *Proc. ACM SIGMOD Conf.*, pp. 259-270, 1990.

[Zip49] G. K. Zipf, *Human Behavior and Principle of Least Effort*, Addison-Wesley, Cambridge, Massachusetts, 1949.