Title: “Filtering XML-Documents Using XPath: XFilter and YFilter”.

Course: T-106.5800 XML ja tietokannat seminaari, Spring 2010

Written By: Sami El-Mahgary

“When processing storage, and transmission costs micro-dollars, then the only real value is the data and its organization.”
About These Two Papers
This is a summarizing work on two papers that deal with a solution to what is now commonly known as SDI, or Selective Dissemination of Information. The examined solutions are XFilter and its more advanced version YFilter. The models described herein assume that data is transmitted in the form of XML documents.

Words that appear in italics are succinctly defined in the glossary at the end.

1. Background

Filtering as described here is limited to data that is in XML documents. Documents in XML have become the de-facto standard for transmitting structured information on the Web. The XML document supports a hierarchical structure with unlimited depth for its elements. Since a document stores data (Unicode assures portability) and its structure, XML is an attractive format for building a Web database.

Traditionally, databases have been a storage place for organized data which sits passively on disks or in another permanent storage device. The data comes alive only when some portion of it is requested by the user in the form of a query. With XML databases and filtering, the scenario is reversed in the sense that it is now the queries which are stored and it is the data, arriving in the form of XML documents, which triggers a chain of events known as filtering.

1.1 Previous Work

The field we're dealing with here is often known as the publish/subscribe scenario, where a site that produces data (i.e. the publisher) makes that data available by providing it to an information broker or more specifically, a message broker. On the other side, users who would like to receive information about specific topics of interest do so by subscribing to the broker [XPATH CHUNK]. As pointed out in [XQUERIES], the user’s interest may be quite sophisticated, s/he may want to be only notified when a certain exchange rate goes over a given threshold value.

The problem thus involves using a message broker, which is understood to mean a lightweight and scalable software component that performs three basic tasks (i) filtering (ii) transformation (for instance putting the results in readable format) and (iii) delivery, which is moving the actual bits through the overlay network [YFILTER/ONYX].

One of the earliest SDI approaches developed in the early 1990s includes the trigger model where each trigger represents a user subscription and thus fires when the information stored in the database meets the user's condition. This model does not work well when the number of subscriptions and hence triggers increases into the tens of thousands [XPATH CHUNK].

In an effort to avoid the problems associated with triggers, systems which grouped queries 'NiagaraCQ' (and also 'Le Subscribe') were developed. NiagaraCQ is a so-called continuous query system where users submit queries expressed in a query language known as XML-QL. Its optimization technique is based on grouping: to reduce the processing time, similar queries (e.g. those that differ only in their constant predicate values) are grouped together [XQUERIES,
YFILTER]. Although NiagaraCQ is a filtering system for XML-documents, its approach is drastically different from XFilter in the sense that it uses execution plans for grouping queries [XFILTER MOBILE].

Another of the shortcomings with NiagaraCQ was its query language. The newer generation of XML filters adopted a different query language known as XPath. The underlying idea is to express user queries as regular expressions using XPath and view the streaming XML documents as the alphabet against which to match. Because this approach is similar to the classical Turing machine with its states and transitions, it is often known as the automata-based approach. This group includes XFilter, YFilter and Xtrie [XQUERIES].

1.2 Basic Idea on Filtering
Someone who wishes to receive documents that are of special interest can join the filtering system as a user by indicating his/her preferences though a form for instance. This set of preferences for each user is known as the user profile. Note that it is also possible for the system to build the user profile more or less automatically through learning techniques.

Because each user's profile is converted into a suitable format, the profile actually turns into a passive query or what the developers call 'standing query' [YFilter]. A query remains on waiting until a suitable XML-document arrives, if the document contains elements that appear in the query, then the document will be inspected to verify if it actually is of interest to the user. Users receive only documents that are of real interest to them, so extra documents get filtered from the other countless documents circulating on the Web.

Filtering is challenging given the incredible volume of data available online and the number of potential users, that is, people with access to the Internet. Unlike NiagaraCQ which operated only on whole documents, XFilter and YFilter have a finer level of granularity: they can respond to given elements (tags) in a document [YFILTER/ONYX].

1.3 The User Profile
One should keep in mind that each user profile (or query) represents a user's interest in subscribing to some sort of information that is being disseminated. This could be news bits that are published regularly, such as major headlines, sports results, weather forecasts, or stock quotes. When the user profiles specifically relate to some part of a tree-structured XML-document, they are often referred to as twig patterns. In the context of XFilter and YFilter, queries, user profiles and twig patterns are all synonymous [VALUE-FILTER]. However, as pointed out in [XFIS]
2. The Building Blocks

There are four basic constituents needed to build the filtering systems proposed by XFilter and YFilter. These are (1) an event based parser for dealing with incoming XML documents, (2) a strong enough language for describing the user profiles (e.g. XPath), (3) the actual filtering system (i.e. XFilter/YFilter) and finally (4) a component for disseminating the information to the interested user. It is also useful to mention that besides XPath, there must be a special XPath parser that actually decomposes the queries, as explained in section 4.1.

The actual HTML document parser used in these cases is SAX, which is described in the next section. Each user profile is expressed through the XPath language which is briefly presented in Chapter 3. The dissemination component is not described in detail; but according to the authors, it is just a means of using unicast delivery for sending those XML documents which matched user preferences to the interested user.

2.1 The SAX parser

The SAX parser is a specially developed parser for XML documents that is freely and widely available [SAX,SAX2]. When it encounters the beginning of an element tag (such as <News>) the parser generates an event by calling the Start Element Handler. Similarly, when in the XML document, the parser finds the end of an element tag (such as </News>) the parser generates an event by calling the End Element Handler. For instance for SAX2, the Sun Java interface Start Element Handler looks as follows:

```java
public void StartElement (String uri, String localName, String qName, Attributes attributes)
```

Elements have two-part names where the first part contains the “expanded” name which is known as a URI. The second part is the pure name without a colon(;) and is known as the localName. The variable qName refers to the more familiar single name that may contain colons. For instance in the example below, the URI would be ‘http://www.w3.org/1999/xhtml’ and the localName would contain simply ‘hr’.[SAX] The table below shows the parsing results for a very simple XML document containing one last name and one first name.

```
<html:hr xmlns:html="http://www.w3.org/1999/xhtml"/>
```

<table>
<thead>
<tr>
<th>XML Document</th>
<th>Events</th>
</tr>
</thead>
</table>
| <?xml version="1.0"?>
| <Name>
|  | start document
| <Last_Name>Bururunda</Last_Name>
|  | start element: Last_Name
|  | characters: Bururunda
|  | end element: Last_Name
| <First_Name>Myriam</First_Name>
|  | start element: First_Name
|  | characters: Myriam
|  | end element: First_Name
| </Name>
|  | end element: Name
|  | end document

<table>
<thead>
<tr>
<th>Name</th>
<th>Last_Name</th>
<th>Bururunda</th>
</tr>
</thead>
<tbody>
<tr>
<td>First_Name</td>
<td>Myriam</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Note how the parser identifies the three elements in the document (Name, Last_Name and First_Name) and returns the associated text.
3. XPath

In a nutshell, the XPath language is a syntax for referring to a part of an XML document, known as the nodes. From the point of view of filtering, the essentials are the location paths that refer to the XML data.

An XML document is assumed to be in a treelike structure, consisting of a tree node along with element nodes and most likely some corresponding text nodes as well (the actual values). There may be other types of nodes as well, such as attribute nodes and comment nodes. From the point of view of XFilter, XPath references a set of nodes through what is known as a location step. Each such step always consists of an axis and a node test. (There may be in addition a predicate test). The axis indicates the kind of hierarchy in the step. Adjacent nodes are indicated by a single slash ‘/’ whereas an ancestor relationship is indicated through ‘//’. A node test is usually the name of an element, or it may be a wildcard, represented by ‘*’.

The language is quite versatile and it allows for nested path expressions which means so-called filters can be applied to element nodes. XPath is built on a hierarchical relationship between the nodes. Besides selecting a node, a location step can contain a predicate which must be true before the location step is validated [XPATH].

4. XFilter: The Basics

XFilter was first published in a conference paper written by Mehmet Altinel paper and Michael J. Franklin in 2000 [XFILTER]. XFilter is one of the first successful SDI techniques for filtering XML documents using the XPath language. It was XFilter which came up with idea of transforming each user profile into an XPath query; its nodes are then ultimately converted into an FSM.

Figure 4.1 below shows the basic idea of how XFilter works. Data that is deemed interesting to users is produced in XML-format, and this data stream then arrives into the system to be processed by the SAX parser (not shown in the figure). The other input involves users’ profiles. These users or subscribers to the system have access to a user-friendly GUI for entering their preferences, which are then converted into XPath queries. The major work is done by the filter engine, which is responsible for decomposing the query into a group of path nodes and performing the actual matching between the incoming documents and the queries.
4.1 Setting up

Before setting up the index, the XPath parser must first identify all the unique element names or nodes in the queries and decompose these into path nodes. Wildcard nodes (‘*’) in the queries are simply overlooked. Each path node refers to an actual element in the query and also represents a state in the FSM. Once the path nodes have been extracted, they can be used to build the index, or the hash table.

The query index is built so that the unique elements in the queries act as the entries in the hash table. Related to each entry, are two lists: the candidate list (CL) and the wait list (WL). The candidate list is initialized so that the first path node in each query is placed as first in the CL corresponding to the element of the corresponding node. In other words, the initial states go to the candidate list, and the rest of the states end up in the waiting list.

More precisely, the CL contains the nodes referring to the states that are trying to be matched by the FSM at a current moment. At start-up, the WL contains the nodes that are the subsequent to those in the CL. Eventually during matching, it is possible that both a node and its subsequent node end up on the candidate list through the copying procedure described in section 4.3.
4.2 The Data Structures

In the following terminology, the term parent query simply refers to the query that the path node belongs to.

For each path node, the following information needs to be stored. QueryID: This is a unique identifier such as 'Q1', that relates the node to the parent query.

1. **Position**: This is a positive integer that further identifies the node in its query. Its value indicates the position of the given node in the query it belongs to. So for instance a node with QueryID 'Q2' and Position '3' means the node occupies the third place in the query 'Q2'.

2. **RelativePos**: This is a positive integer value that can also be assigned a negative value (the value -1). RelativePos indicates the relative position that exists between the current path node and the previous node in the parent query. It is set according to the following rules: (1) RelativePos is set to -1 any time the node is preceded by '//'. (2) RelativePos = 0 for the first node, except if first node is preceded by '//', in which case RelativePos = -1. (3) In all other cases, 1) RelativePos = nr_wild + 1, where nr_wild is the number of wild cards ('*') that separate the current node from the predecessor node.

3. **Level**: This integer value is slightly more complex than RelativePos. Basically, the purpose of Level is to indicate the level (with regard to the whole XML document) at which the current/corresponding path node should be checked. The value of Level is dependent on RelativePos in the sense that it too will be set to -1 for cases in which RelativePos = -1. If the current node is the first node in the query and refers to an absolute distance from the root node, then Level = dist_from_root + 1 where dist_from_root is the current node's distance from the root node. In all other cases, Level is set to 0.
4.3 Matching: State Transitions

Once the Query Index has been setup, the algorithm proceeds by reading in a start-element and thus calling the 'Start Element Handler'. The element name in question is looked up in the Index and if it’s found, then all the nodes in the CL are examined (for that element). For each node, a level check is performed to check that the level of the node in CL corresponds to the current level being processed. In other words, the level of the CL node must equal to the level of the element being processed in the XML document, unless the level of the CL node has been designated as negative (-1). In case the CL node does equal -1, the node is said to pass\(^1\), and there is no need for the two levels to correspond.

When an end-of-element tag such as ‘</news>’ is encountered in the SAX parsing, then the corresponding path node gets deleted from the CL. This effectively restores the state of CL to the state it was in when the corresponding start-of-element tag was encountered. This mimics 'backtracking' as it allows XFilter to handle the case where a given element name appears in the same document multiple times, but at different levels of nesting.

\(^{1}\) More specifically, if the node contains an attribute check (such as @player='Nieminen') then there must also be a match in the attribute value. For brevity, the attribute check has been omitted here.
If instead a data(text) element is encountered, then it is processed similarly to a ‘start-of-element’ tag except that there is no need for an attribute check.

5. XFilter: Advanced Issues

This chapter examines some extensions and possible improvements for XFilter.

There is some ineffectiveness in the index structuring with XFilter that is evident in the setup stage. We recall that the first path node in any query is always assured a place in the candidate list as it comes first in the list. The fact that it comes in first may not be a very efficient implementation as the first elements usually suffer from poor selectivity, since they are often the same among lots of the queries [XFILTER INDEX].

So as a result, uneven lengths of candidate lists are likely, as the length of a CL list for the element ‘news’ is likely to be longer than for element ‘Nieminen’ for instance. One way to even out the length of the candidate lists is to use an improvement developed by the same authors, called List Balancing [XFILTER].

5.1 List Balancing

When a new query is being added to the index structure, instead of automatically adding its first node to the candidate list, the procedure is now as follows. Assuming that the new query is Q3 and contains the path */sports/news/wimbledon, the algorithm first finds out which of the three nodes (‘sports’, ‘news’ or ‘wimbledon’) has the shortest CL in the index structure. In case this node would be ‘news’, that element would become the so-called pivot element, which would go first to the Candidate List. In addition, the node that is right before the pivot element (in this case ‘sports’) is stored in the stack. This node is sometimes referred to as the prefix node. [XFILTER]

Now any time the pivot node gets activated, the prefix node is also checked for a match. If there is no match, then the matching process for the pivot node in question must be aborted. Even though there is an additional check for the prefix node, the fact that the candidate lists are now well-balanced increases processing speed so much that its use is well justified [YFILTER].

5.2 Prefiltering

A quick and straightforward way to improve on XFilter is by using prefiltering, which consists of excluding those queries from the matching process which contain an element name that does not appear in the document being processed. This means that effectively each incoming XML-document needs to be parsed twice, but in most cases, this additional step speeds things up later.

The idea is to build an occurrence table for a so-called key-element, which is chosen similarly to the pivot element[XFILTER].

Let’s assume there are three queries as shown below and that the key elements are ‘Wimbledon’ for query Q₁ and ‘Anaheim’ for queries Q₂ and Q₃. Then the occurrence table would contain an entry for ‘Wimbledon’ (Q₁) and ‘Anaheim’ (Q₂). If now a document were to arrive containing the element, say ‘Ducks’, then that document would have to be ignored because the element ‘Ducks’
is not in the occurrence table. Prefiltering is thus an additional useful step to be performed at the very beginning of filtering, prior to actually building the hash table and any list balancing.

<table>
<thead>
<tr>
<th>Q1</th>
<th>/sports/Wimbledon/scores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>/sports/NHL/Anaheim/news</td>
</tr>
<tr>
<td>Q3</td>
<td>/sports/Anaheim/news</td>
</tr>
</tbody>
</table>

5.3 XFilter Applied to Mobile Clients
The XFilter for Mobile clients (referred to here as XFilter/CQMC) was developed to address several shortcomings in XFilter. One of these was that on matching, the whole document is returned to the user. This requires lots of storage capacity that a mobile device would not be able to handle. Another point is that in case the user profiles are complex, a more powerful language than XPath is needed, and in this case the choice was XML-QL[XFILTER MOBILE].

Not only is XML-QL more powerful than XPath, it is considered the most powerful among all XML query languages [XML QUERY]. With XFilter/CQMC, the profiles are stored in an XML-document using XML-QL. Like XFilter, XFilter/CQMC too uses FSM, but when the document matches, the matches are stored in a special content list, so that the whole document need not be sent.

6. YFilter

6.1 Basics of YFilter
The idea with XFilter was to build an FSM for each query. To be able to handle large numbers of queries that may reach tens or even hundreds of thousands in number, a different and efficient approach is required. This paved the need for YFilter which combines all the queries into a single machine, which is represented a single non-deterministic finite automata, or NFA. Effectively this allows for sharing the prefixes that are common in the queries.

The NFA approach has the advantage that it allows for several alternative transitions from one state to the other, and a query matching is fulfilled when the path from any alternative leads to a final accepting state. The initial state is the state that is common to all queries, so the algorithm starts from this state whenever a new document arrives.

Because the outputs depend only on the present state, an NFA in YFilter is referred to as a Moore machine. More formally, the output function can now be viewed as a mapping from accepting states into a partitioning of elements (or identifiers since wildcards and epsilons are allowed) for all the included queries. Each partition contains the identifiers for any query that shares the accepting state in question [YFILTER/ONYX].

Compared with the FSM approach, Dia identifies in his thesis the following basic advantages for YFilter:

1. Large number of queries (path expressions) can be represented with relatively few machine states.
2. Support for complex documents (recursive nesting is allowed).
3. Ease of maintenance, as new queries can be easily added.
The fact that recursive nesting is allowed means that twig patterns are more easily supported in YFilter. This is done by breaking them into linear path expressions [XFIS].

Figure 6.1 Above The NFA representation for the eight queries shown on the left.

6.2 XPath Queries with YFilter
The parsing of a document proceeds in a similar fashion as with XFilter. The events that are generated by the SAX parser callback on specific handlers that handle the state transition in the NFA algorithm. States are indicated with a circle, if there are two concentric circles, then the state is an accepting state as well. In Figure 6.1 next to each accepting state, the IDs of the actual accepted queries are marked. The actual transitions are denoted by edges. Note that following the 'a' element, there are four alternative transitions: (i) 'b', (ii) 'c', (iii) the empty transition $\varepsilon$ (requiring no input) and (iv) the wildcard '*' that will match any element.

So in case this 'a' is followed by inputs 'b' and then 'c', the resulting states may be either $\{Q_3, Q_8\}$ or $\{Q_5\}$, through the wildcard. The empty transition gives more flexibility and it in fact allows for connecting queries or parts thereof together.
6.3 The Data Structures used With YFilter
For each state, the following four basic items are stored as part of the data structure of YFilter.

1. a stateID, to uniquely identify the state.
2. a Type structure classifying it as a ‘//-child state’, an Accepting state or a normal state.
3. a Hash Table with entries stored as <symbol, stateID>.
4. for an accepting state, the QueryIDs or list of corresponding queries.

6.4 How YFilter works.
Like XFilter, YFilter also works in an event-driven way. Because nodes share common prefixes, there must be support for backtracking. With YFilter, any time an ‘end-of-element’ tag is encountered, the algorithm backtracks to those states that were current when the corresponding ‘start-of-element’ event was fired. That is why prior states need to be stored in a stack.

When a new element tag is found in the document, the NFA will follow all the transitions for which there is a match from any state that is currently active. More precisely, the four following checks must be done for any state that is active.

1. Check the incoming element in the hash table of the current state, and if found, then its corresponding ‘stateID’ is added to the list of ‘target states’.
2. The wildcard ‘*’ is checked in the hash table and if found, then its corresponding ‘stateID’ is also added to the target states.
3. Check the type of the corresponding state. When the state happens to be a ‘//-child state’, then the state’s own stateID is added to the target states, which creates a self-loop that is characterized by the wildcard ‘*’ in the NFA diagram.
4. Finally, the hash table is checked for the element e and if it is found, the ‘//-child state’ as indicated by the corresponding entry stated is processed in a recursive fashion, following the three rules given previously.
6.5 Dealing with Value-Based Predicates.

Value-based predicates can be handled in two ways in YFilter. The straightforward way is known as the inline approach and consists of keeping track for each query, which of its predicates were met during matching. When reaching an accepting state, the bookkeeping is used to check which queries of that state have all their predicates satisfied; these queries can then be considered as true matches. This approach does have its drawbacks, especially in cases where backtracking is necessary. As shown in [YFILTER], a fragment such as `<a a_1=v_1> </a> <a a_2=v_2> </a>` could match a predicate in a query, without the algorithm noticing that this fragment actually refers to two distinct ‘a’ elements.

The other approach, ‘Selection Postponed’, does not process the predicates as soon as the corresponding state is reached during the matching processing. Instead, it is only when reaching an accepting state that the value-based predicates are checked for all otherwise matching queries.

6.6 Hybrid: Variations on YFilter

The YFilter paper also mentions a Hybrid approach, which borrows both from XFilter and YFilter. It differs from XFilter from the very start in that queries are split or decomposed into smaller fragments any time a ‘*’ or ‘/’ is encountered. For each substring, there is one corresponding node, and the data-structure associated with each node is just as with XFilter. However, since
we're dealing with substrings, the variable RelativePos now indicates the distance from the end of the previous substring (rather than the query) to the end of the current substring.

The substrings from all queries are used to build a single trie index. [YFILTER] Transitions between two substrings occurs analogously to XFilter, on a query-by-query basis. Hybrid borrows from YFilter the matching process: the matching of a substring in the trie index can be shared by all the the queries containing the substring being matched [YFILTER/ONYX].

7. Conclusions

7.1 Efficiencies of XFilter and YFilter.
For XFilter the authors measure the efficiency of their algorithm according to the total filtering time, for YFilter a metric known as MQPT is used which gives a better approximation of the true filtering time [YFILTER]. XFilter can be said to give satisfactory performance even in its basic implementation since processing some 100 000 profiles required on the average 3 seconds without any wildcards, which is roughly 1000 faster than CQNiagara [NIAGARA].

A test comparison involved XFilter (with list balancing), YFilter and Hybrid. The XML documents were generated using IBM’s XML generator. The settings for the test were as follows: the depth of the document, D=6. The probability of both a wildcard (‘*’) and of a ‘//’ occurring a certain location step were both 20%. This is indicated in both parameters W and DS being equal to 0.2. It’s clear from Figure 7.1 that increasing the number of queries has little effect on YFilter. In fact it was about 30ms faster when the number of queries was around 150 000.

Tests showed that combining prefiltering with list balancing produced in general the best results for both a uniform and skewed set of documents. It was interesting to note that using only prefiltering was faster than using only list balancing for a uniformly distributed set of input XML documents. The reverse was true for a skewed set[XFILTER].

As for XFilter, it can be very sensitive to the depth of the document, which can increase nearly logarithmically for a skewed test (meaning the element names in queries are not selected randomly)[XFILTER]. However, tests conducted between XFilter and YFilter in a situation involving a significant portion of identical queries (duplicate paths occur often in practice) showed that although XFilter was still lagging behind YFilter, the differences were less strong (Fig. 7.2). In other words the benefits/cost ratio for YFilter getting weaker [YFILTER].
7.2 Final Words
The problem is aptly put in [VALUE-FILTER] as that of processing a very vast, every increasing content against “millions of user subscriptions”. Both XFilter and YFilter make use of the realization that filtering is in sort the reverse of querying a database. It is now the queries which must be indexed and moreover, queries that are somewhat similar or contain common elements must be treated together to achieve an adequate level of efficiency. When first introduced, XFilter achieved very good results since its FSM strategies actually allows queries to be executed in parallel.

YFilter is not without its problems either. A recent dissertation work showed how the non-determinism that comes from the expressions containing ‘*’ and ‘/’ can, in complex queries, result in a quite large number of active states that need to be pushed onto the stack [BOONE]. When the number of queries becomes very large, the approach taken by YFilter becomes significantly faster. However, fully implementing YFilter may not be a simple issue.

Fig 7.1 XFilter, YFilter and Hybrid Speeds. \( D=6, W=0.2, DS=0.2 \)

Fig 7.2 XFilter, YFilter and Hybrid with lots of duplicate queries. The circled area shows how the improved performance for XFilter.
For instance the implementation done at the University of Berkeley (featuring the name of Y. Diao) doesn’t include unlimited nesting of path expressions. This is probably only a very minor issue in most cases, but it does show that implementing all the details of YFilter as described in the paper is no easy task [YFILTER].

8. References

About SAX:

A good reading list for XML databases:
http://www.cs.utexas.edu/~wzhang/cs6453/papers/xmldatabase/reading-list.html

About XFilter:
The 'official' XFilter paper is


A version of XFilter adapted to mobile clients:
[XFILTER MOBILE]

About YFilter:
The 'official' YFilter paper is:

A modified version of YFilter (ONYX) is described in the Ph.D. thesis by Dia, which was supervised by Dr M. Franklin.

Other XML-Filtering Systems:
The following describe various implementations of XML-filtering besides Xfilter and YFilter. The thesis [] is good overview of the state-of-the-art until the end of year 2005.

adapted to mobile clients:


About Automata, XPATH, etc.:


[AUTOMATA] J. Squire, Notes on Automata, Univ. of Maryland, Baltimore County


9. Glossary

Continuous query (also continual query): is a query in a monitoring state. Whenever a certain update reaches a given threshold value, a certain result is returned. This could be a notification that NOKIA's stock price increased by at least 7%. The queries in XFilter and Filter are examples of continuous queries.

Dissemination: in the context of documents, means to proactively distribute any relevant data to the subscribers or users. See picture at the end of the glossary

Filtering: The process of filtering out from the Web irrelevant information so as to being able to deliver to users documents that are of real interest to them.

Filtering Time: The total time required to process a given document, that is, parse it and output its results.

FSM (Finite State Machine): models transitions from one state to another where the number of states is finite. With XFilter, each state has at most one target state, so the FSM is also known as a deterministic automata.

Key-element: In XFilter, when doing prefiltering, the key element is an element for which an occurrence table is built. It can be selected in a similar way as the pivot element in list balancing.

List Balancing: In XFilter, a method for improving the efficiency of the algorithm by trying to make lengths of the various candidate lists more even.

Multi-query Processing Time (MQPT): used with YFilter as a performance metric, this measure gives the pure filtering time. It is obtained by taking the actual filtering time and subtracting from it the parsing time.

NFA (Non-deterministic Finite Automata) also models transitions between a finite number of states, but allows for reaching different states from a given state.

Overlay network: A logical network (and hence not a physical one) that runs on top of another network.

Path Query: A query with path expressions that contains constraints relating to the structure and content of the XML document.

Pivot element: In XFilter, 's list balancing, this element is determined from the shortest candidate list. It is placed in front of the candidate list.

Prefiltering: In XFilter, refers to the process of examining the queries for elements that do not appear in the document being processed. If such elements are found, then the whole query can be safely excluded from the matching process. This additional step should be done prior to the actual filtering process, whence the name.

Publish/Subscribe (pub/sub): According to Diao and Franklin, this is a communication model based on the many-to-many paradigm that directs the flow of messages from senders to receivers based on the interests of the receiving party.

Publisher: Any type of information (news agency, database/monitoring application) provider which publishes into information in the form of XML messages.
**Regular expression:** Basically just a formal way of indicating a pattern or set of strings, often without having to list all the actual elements of the string.

**SDI (Selective Dissemination of Information):** According to M.J. Franklin, this basically means sending the right data to the right users at the right time.

**Stream-based Processing:** Based on the idea of processing the incoming data as soon as it arrives. When the messages are large, the existing queries begin the filter process well before the whole message is received.

**Trie:** A prefix tree where the nodes are keys and the values are associated with either leaves or inner nodes. A trie often represents an FSM. Picture below is courtesy of Wikipedia.

**Twig pattern:** A pattern where the nodes represent the elements the user is interested in and the edges indicate the structural relationship placed by the user on the elements in question. Need not be linear

**Unicast stream:** A one-one to connection whereby only those clients that actually requested the data receive it.

**User profile:** A declaration of a single user's (or a group of users') interests in certain data, may also include information on priorities.

**Value-based predicate:** In XPath, a criterion (or filter) that refers to individual elements.

**XML Message Broker:** In its basic form, Diao defines a broker as middleware that acts as the central server between the publishers and the subscribers. The Broker receives as inputs the XML documents, which make up the incoming stream of XML messages along with new XPath queries.

![Trie Diagram](image)

* Absolute distance: A path node is said to refer to an absolute distance when that node is either located with respect to the root or the node is located a given fixed number of wildcard nodes ('*') away from the root node.