XML PUBLISH/SUBSCRIBE

Yanlei Diao
Department of Computer Science
University of Massachusetts Amherst
yanlei@cs.umass.edu

Michael J. Franklin
Department of Electrical Engineering and Computer Science
University of California Berkeley
franklin@cs.berkeley.edu

SYNONYMS

Selective XML dissemination, XML filtering, XML message brokering

DEFINITION

As stated in the entry "Publish/Subscribe over Streams", publish/subscribe (pub/sub) is a many-to-many communication model that directs the flow of messages from senders to receivers based on receivers' data interests. In this model, publishers (i.e., senders) generate messages without knowing their receivers; subscribers (who are potential receivers) express their data interests, and are subsequently notified of the messages from a variety of publishers that match their interests.

XML publish/subscribe is a publish/subscribe model in which messages are encoded in XML and subscriptions are written in an XML query language such as a subset of XQuery 1.0.[1]

In XML-based pub/sub systems, the messages brokers that serve as central exchange points between publishers and subscribers are called XML message brokers.

HISTORICAL BACKGROUND

As described in "Publish/Subscribe over Streams", XML pub/sub has emerged as a solution for loose coupling of disparate systems at both the communication and content levels. At the communication level, pub/sub enables loose coupling of senders and receivers based on the receivers' data interests. With respect to content, XML can be used to encode data in a generic format that senders and receivers agree upon due to its flexible, extensible, and self-describing nature; this way, senders and receivers can exchange data without knowing the data representation in individual systems.

SCIENTIFIC FUNDAMENTALS

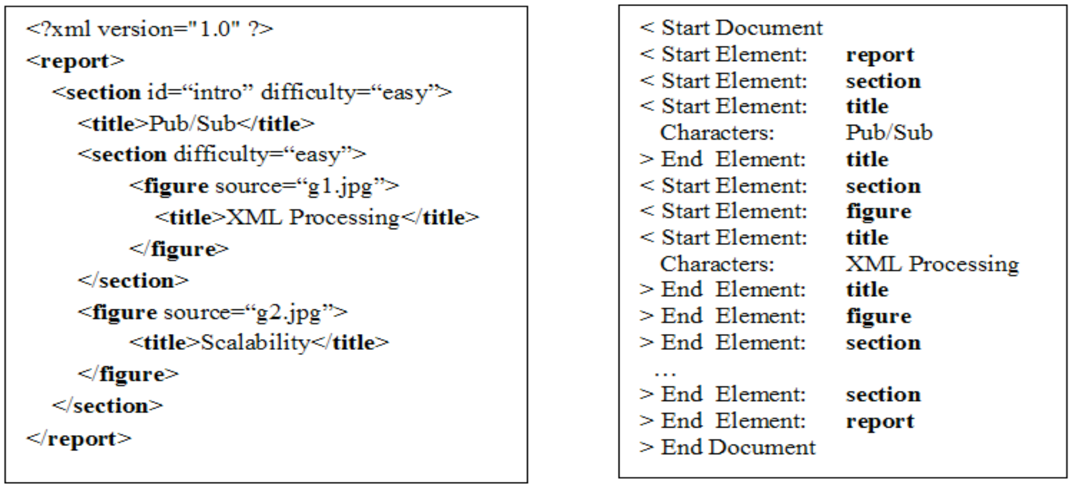
XML pub/sub raises many technical challenges due to the requirements of large-scale publish/subscribe (as described in the entry "Publish/Subscribe over Streams"), the volume of XML data, and the complexity of XML processing. Among all, two challenges are highlighted below:

- *XML stream processing*. In XML-based pub/sub systems, XML data continuously arrives from external sources, and user subscriptions, stored as continuous queries in a message broker, are evaluated every time when a new data item is received. Such XML query processing is referred to as *stream-based*. In cases where incoming messages are large, stream-based processing also needs to start before the messages are completely received in order to reduce the delay in producing results.
- *Handling simultaneous XML queries*. Compared to XML stream systems, a distinguishing aspect of XML pub/sub systems lies in the size of their query populations. All the queries stored in an XML message broker are simultaneously active and need to be matched efficiently with each incoming XML message. While multi-query processing has been studied for relational databases and relational streams, the complexity of XML processing, including structure matching, predicate evaluation, and transformation, requires new techniques for efficient multi-query processing in this new context.

[1] In the context of XML pub/sub, "messages" and "documents" are often used exchangeably.

Foundation of XML Stream Processing for Publish/Subscribe

Event-based parsing. Since XML messages can be used to encode data of immense sizes (e.g., the equivalent of a database's worth of data), efficient query processing requires fine-grained processing upon arrival of small constituent pieces of XML data. Such fine-grained XML query processing can be implemented via an event-based API. A well known example is the *SAX* interface that reports low-level parsing events incrementally to the calling application. Figure 1 shows an example of how a SAX interface breaks down the structure of the sample XML document into a linear sequence of events. "Start document" and "end document" events mark the beginning and the end of the parse of a document. A "start element" event carries information such as the name of the element and its attributes. A "characters" event reports a text string residing between two XML tags. An "end element" event corresponds to an earlier "start element" event and marks the close of that element. To use the SAX interface, the application receiving the events must implement handlers to respond to different events. In particular, stream-based XML processors can use these handlers to implement event-driven processing.



```
<?xml version="1.0" ?>
<report>
  <section id="intro" difficulty="easy">
    <title>Pub/Sub</title>
    <section difficulty="easy">
        <figure source="g1.jpg">
          <title>XML Processing</title>
        </figure>
    </section>
    <figure source="g2.jpg">
        <title>Scalability</title>
    </figure>
  </section>
</report>
```

```
< Start Document
< Start Element:    report
< Start Element:    section
< Start Element:    title
  Characters:       Pub/Sub
> End  Element:     title
< Start Element:    section
< Start Element:    figure
< Start Element:    title
  Characters:       XML Processing
> End  Element:     title
> End  Element:     figure
> End  Element:     section
 …
> End  Element:     section
> End  Element:     report
> End Document
```

Figure 1: An example XML document and results of SAX parsing

An automata-based approach. A popular approach to event-driven XML query processing is to adopt some form of *finite automaton* to represent path expressions [Altinel and Franklin 2000; Ives et al., 2002]. This approach is based on the observation that a path expression (a small, common subset of XQuery) written using the axes ("/", "//") and node tests (element name or "*") can be transformed into a regular expression. Thus, there exists a finite automaton that accepts the language described by such a path expression [Hopcroft and Ullman 1979]. Such an automaton can be created by mapping the location steps of the path expression to the automaton states. Figure 2 shows an example automaton created for a simple path expression, where the two concentric circles represent the accepting state. When arriving XML messages are parsed with an event-based parser, the events raised during parsing are used to drive the execution of the automaton. In particular, "start element" events drive the automaton through its various transitions, and "end element" events cause the execution to backtrack to the previous states. A path expression is said to match a message if during parsing, the accepting state for that path is reached.

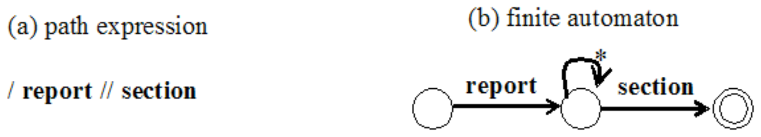


Figure 2: A path expression and its corresponding finite automaton

XML Filtering

In XML filtering systems, user queries are written using path expressions that can specify constraints over both *structure* and *content* of XML messages. These queries are applied to individual messages (hence, stateless processing). Query answers are "yes" or "no"—computing only Boolean results in XML filtering avoids complex issues of XML query processing related to multiple matches such as ordering and duplicates, hence enabling simplified, high-performance query processing.

XFilter [Altinel and Franklin 2000], the earliest XML filtering system, considers matching of the structure of path expressions and explores *indexing* for efficient filtering. It builds a dynamic index over the queries and uses the parsing events of a document to probe the query index. This approach quickly results in a smaller set of queries that can be potentially matched by a message, hence avoiding processing queries for which the message is irrelevant. Built over the states of query automata, the dynamic index identifies the states that the execution of these automata is attempting to match at a particular moment. The content of the index constantly changes as parsing events drive the execution of the automata.

YFilter [Diao et al., 2003] significantly improves over XFilter in two aspects. By creating a separate automaton per query, XFilter can perform redundant work when significant commonalities exist among queries. Based on this insight, YFilter supports *sharing* in processing by using a combined automaton to represent all path expressions; this automaton naturally supports shared representation of all common prefixes among path expressions. Furthermore, the combined automaton is implemented as a *Nondeterministic Finite Automaton* (NFA) with two practical advantages: 1) a relatively small number of states required to represent even large numbers of path expressions and complex queries (e.g., with multiple wildcards '*' and descendent axes '//'), and 2) incremental maintenance of the automaton upon query updates. Results of YFilter show that its shared path matching approach can offer order-of-magnitude performance improvements over XFilter while requiring only a small maintenance cost.

Structure matching that XFilter considers is one part of the XML filtering problem; another significant part is the evaluation of predicates that are applied to path expressions (e.g., addressing attributes of elements, text data of elements, or even other path expressions) for additional filtering. Since shared structure matching has been shown to be crucial for performance, YFilter supports predicate evaluation using *post-processing* of path matches after shared structuring matching, and further leverages relational processing in such post-processing.

Figure 3 shows two example queries and their representation in YFilter. Q1 contains a root element "/nitf" with two nested paths applied to it. YFilter decomposes the query into two linear paths "/nitf/head/pubdata[@edition.area="SF"]", and "/nitf//tobject.subject[@tobject.subject.type ="Stock"]". The structural part of these paths is represented using the NFA with the common prefix "/nitf" shared between the paths. The accepting states of these paths are state 4 and state 6, where the network of operators (represented as boxes) for the remainder of Q1 starts. At the bottom of the network, there is a selection (σ) operator above each accepting state to handle the value-based predicate in the corresponding path. To handle the correlation between the two paths (e.g., the requirement that it should be the same nitf element that makes these two paths evaluate to true), YFilter applies a join (⊳⊲) operator after the two selections. Q2 is similar to Q1 and hence shares a significant portion of its representation with Q1.

Index-Filter [Bruno et al., 2003] builds indexes over both queries and streaming data. The index over data speeds up the processing of large documents while its construction overhead may penalize the processing of small ones. Results of a comparison between Index-Filter and YFilter show that Index-Filter works better when the number of queries is small or the XML document is large, whereas YFilter's approach is more effective for large numbers of queries and short documents.

```
Q1:  $msg/nitf[head/pubdata[@edition.area="SF"]]
       [.//tobject.subject[@tobject.subject.type="Stock"]]

Q2:  $msg/nitf[head/pubdata[@edition.area="SF"]]
     [.//tobject.subject[@tobject.subject.matter="fishing"]]
```

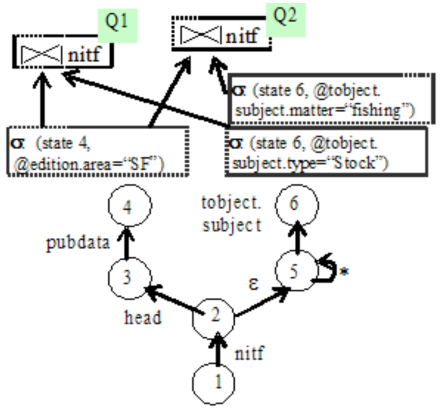

Q1
⋈ nitf
Q2
⋈ nitf
σ (state 6, @tobject.subject.matter="fishing")
σ (state 4, @edition.area="SF")
σ (state 6, @tobject.subject.type="Stock")
4
tobject.subject
6
pubdata
3
ε
5
*
head
2
nitf
1

Figure 3: Example queries and their representation in YFilter

XMLTK [Green et al., 2004] converts YFilter's NFA to a *Deterministic Finite Automaton* (DFA) to further improve the filtering speed. A straightforward conversion could theoretically result in severe scalability problems due to an explosion in the number of states. This work, however, shows that such explosion can be avoided in many cases by using lazy construction of the DFA and placing certain restrictions on the types of documents and queries supported (when suitable for the application). *XPush* [Gupta and Suciu, 2003] further explores a pushdown automaton for shared processing of both structure and value-based constraints. Such an automaton can provide high efficiency when wildcard ('*') and descendant ('//') operators are rare in queries and periodic reconstruction of the automaton can be used.

FiST [Kwon et al., 2005] views path expressions with predicates as twig patterns and considers ordered twig pattern matching. For such ordered patterns, it transforms the patterns as well as XML documents intro sequences using Prufer's method. This approach allows holistic matching of ordered twig patterns, as opposed to matching individual paths and then merging their matches during post-processing in YFilter (which works for both ordered and unordered patterns), resulting in significant performance improvements over YFilter.

XML Filtering and Transformation

XML filtering solutions presented above have not addressed the transformation of XML messages for customized result delivery, which is an important feature in XML-based data exchange and dissemination. For XML transformation, queries are written using a richer subset of XQuery, e.g., the *For-Where-Return* expressions.

To support efficient transformation for many simultaneous queries, YFilter [Diao and Franklin, 2003] further extends its NFA-based framework and develops alternatives for building transformation functionality on top of shared path matching. It explores the tradeoff between shared path matching and post-processing for result customization by varying the extent to which they push paths from the For-Where-Return expressions into the shared path matching engine. To further reduce the remarkable cost of post-processing of individual queries, it employs provably correct optimizations based on query and DTD (if available) inspection to eliminate unnecessary operations and choose more efficient operator implementations for post-processing. Moreover, it provides techniques for also sharing post-processing across multiple queries, similar to those in continuous query processing over relational streams.

Stateful XML Publish/Subscribe

[Hong et al., 2007] addresses efficient processing of large numbers of continuous inter-document queries over XML Streams (hence, stateful processing). The key idea that it exploits is to dissert query specifications into tree patterns evaluated within individual documents and value-based joins preformed across documents. While employing existing path evaluation techniques (e.g., YFilter) for tree pattern evaluation, it proposes a scalable join processor that leverages relational joins and view materialization to share join processing among queries.

XML Routing
As described in "publish/subscriber over streams", distributed pub/sub systems need to efficiently route messages from their publishing sites to the brokers hosting relevant queries for complete query processing. While the concept of content-based routing and many architectural solutions can be applied in XML-based pub/sub systems, routing of XML messages raises additional challenges due to the increased complexity of XML query processing.

Aggregating user subscriptions into compact routing specifications is a core problem in XML routing. Chan et al. [Chan et al., 2002] aggregate tree pattern subscriptions into a smaller set of generalized tree patterns such that (1) a given space constraint on the total size of the subscriptions is met, and (2) the loss in precision (due to aggregation) during document filtering is minimized (i.e., a constrained optimization problem). The solution employs tree-pattern containment and minimization algorithms and makes effective use of document-distribution statistics to compute a precise set of aggregate tree patterns within the allotted space budget.

ONYX [Diao et al., 2004] leverages YFilter technology for efficient routing of XML messages. While subscriptions can be written using For-Where-Return expressions, the routing specification for each output link at a broker consists of a *Disjunctive Normal Form* (DNF) of absolute linear path expressions, which generalizes the subscriptions reachable from that link while avoiding expensive path operations. These routing specifications can be efficiently evaluated using YFilter, even with some work shared with complete query processing at the same broker. To boost the effectiveness of routing, ONYX also partitions the XQuery-based subscriptions among brokers based on exclusiveness of data interests.

Gong et al. [Gong et al., 2005] introduce Bloom filters into XML routing. The proposed approach takes a path query as a string and maps all query strings into a Bloom filter using hash functions. The routing table is comprised of multiple Bloom filters. Each incoming XML message is parsed into a set of candidate paths that are mapped using the same hash functions to compare with the routing table. This approach can filter XML messages efficiently with relatively small numbers of false positives. Its benefits in efficiency and routing table maintenance are significant when the number of queries is large.

KEY APPLICATIONS
Personalized News Delivery. News providers are adopting XML-based formats (e.g., *News Industry Text Format* [IPTC, 2004]) to publish news articles online. Given articles marked up with XML tags, a pub/sub-based news delivery service allows users to express a wide variety of interests as well as to specify which portions of the relevant articles (e.g., title and abstract only) should be returned. *Really Simple Syndication* (RSS) provides similar yet simpler services based on URL- and/or keyword-based preferences.

Application Integration. XML publish/subscribe has been widely used to integrate disparate, independently-developed applications into new services. Messages exchanged between applications (e.g., purchase orders and invoices) are encoded in a generic XML format. Senders publish messages in this format. Receivers subscribe with specifications on the relevant messages and the transformation of relevant messages into an internal data format for further processing.

Mobile services. In mobile applications, clients run a multitude of operating systems and can be located anywhere. Information exchange between information providers and a huge, dynamic collection of heterogeneous clients has to rely on open, XML-based technologies and can be further facilitated by pub/sub technology including filtering and transformation for adaptation to wireless devices.

DATA SETS
XML data repository at University of Washington http://www.cs.washington.edu/research/xmldatasets/

Niagara experimental data http://www.cs.wisc.edu/niagara/data.html

URL to CODE

YFilter is an XML filtering engine that processes simultaneous queries (written in a subset of XPath 1.0) against streaming XML messages in a shared fashion. For each XML message, it returns a result for every matched query. http://yfilter.cs.umass.edu/

ToXgene is a template-based generator for large, consistent collections of synthetic XML documents. http://www.cs.toronto.edu/tox/toxgene/

XMark is an XQuery benchmark suite to analyze the capabilities of an XML database. http://www.xml-benchmark.org/

CROSS REFERENCES

Publish/Subscribe over streams, Continuous queries
XML, XML document, XML schema, XPath/XQuery
XML parsing, XML stream processing

RECOMMENDED READING

[Altinel and Franklin, 2000] Altinel, M. and Franklin, M.J. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *VLDB,* 53-64, 2000.

[Bruno et al., 2003] Bruno, N., Gravano, L., Doudas, N., and Srivastava, D. Navigation- vs. Index-based XML Multi-query processing. In *ICDE*, 139-150, 2003.

[Chan et al., 2002] Chan, C.Y., Fan, W., Felber, P., Garofalakis, M.N., and Rastogi, R. Tree Pattern Aggregation for Scalable XML Data Dissemination. In *VLDB*, 826-837, 2002.

[Diao et al., 2003] Diao, Y., Altinel, M., Zhang, H., Franklin, M.J., and Fischer, P.M. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM Transactions on Database Systems* (*TODS*), 28(4), 467-516, December 2003.

[Diao and Franklin, 2003] Diao, Y. and Franklin, M.J. Query Processing for High-Volume XML Message Brokering. In *VLDB*, 261-272, 2003.

[Diao et al., 2004] Diao, Y., Rizvi, S., and Franklin, M.J. Towards an Internet-Scale XML Dissemination Service. In *VLDB*, 612-623, 2004.

[Gong et al., 2005] Gong, X., Qian, W., Yan, Y., and Zhou, A. Bloom filter-based XML packets filtering for millions of path queries. In *ICDE*, 2005.

[Hong et al., 2007] Hong, M., Demers, A.J., Gehrke, J., Koch, C., Riedewald, M., and White W.M. Massively multi-query join processing in publish/subscribe systems. In *SIGMOD*, 761-772, 2007.

[Hopcroft and Ullman, 1979] Hopcroft, J. E., and Ullman, J. D. *Introduction to Automata Theory, Languages and Computation*. Addition-Wesley Pub. Co., Boston, MA, 1979.

[Green et al., 2004] Green, T. J., Gupta, A., Miklau, G., Onizuka, M., Suciu, D. Processing XML Streams with Deterministic Automata and Stream Indexes. In *ACM Transactions on Databases* (*TODS*), 29(4), December, 2004.

[Gupta and Suciu, 2003] Gupta, A. K., and Suciu, D. Streaming processing of XPath queries with predicates. In *SIGMOD*, 419-430, 2003.

[IPTC, 2004] Internal Press Telecommunications Council. News Industry Text Format. http://www.nitf.org/, 2004

[Ives et al., 2002] Ives, Z.G., Halevy, A.Y., and Weld, D.S. An XML Query Engine for Network-Bound Data. In *VLDB Journal*, 11(4), 380-402, December 2002.

[Kwon et al., 2005] Joonho Kwon, Praveen Rao, Bongki Moon, Sukho Lee. FiST: Scalable XML Document Filtering by Sequencing Twig Patterns. In *VLDB*, 217-228, 2005.