XML Evolution: A Two-phase XML Processing Model Using XML Prefiltering Techniques

Chia-Hsin Huang[1,2] jashing@iis.sinica.edu.tw

Tyng-Ruey Chuang[2] trc@iis.sinica.edu.tw

James J. Lu[3,2] jlu@mathcs.emory.edu

Hahn-Ming Lee[2,4] hmlee@mail.ntust.edu.tw

[1] Department of Electronic Engineering, National Taiwan University of Science and Technology, Taipei 106, Taiwan

[3] Department of Mathematics and Computer Science, Emory University, Atlanta, GA 30332

[2] Institute of Information Science, Academia Sinica, Taipei 115, Taiwan

[4] Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei 106, Taiwan

ABSTRACT

An implementation based on the two-phase XML processing model introduced in [3] is presented in this paper. The model employs a prefilter to remove uninteresting fragments of an input XML document by approximately executing a user's queries. The refined candidate-set XML document is then returned to the user's DOM- or SAX-based applications for further processing. In this demonstration, it is shown that the technique significantly enhances the performance of existing DOM- and SAX-based XML applications and tools (*e.g.*, XPath/XQuery processors and XML parsers), while reducing computational resource needs. Moreover, the prefilter can be easily integrated into existing applications by adding only one instruction. We also present an enhancement to the indexing scheme of the prefiltering technique to speed up the evaluation of certain axes.

1. INTRODUCTION

In the conventional XML processing model, user applications employ XPath [14] expressions to retrieve XML document fragments (Figure 1(a)). XPath processors translate given XPath expressions into node access instructions to process an in-memory Document Object Model (DOM) tree or a series of Simple API for XML (SAX) events. For applications retrieving a small subset of a large XML document, building a DOM-tree or sequentially parsing the document into SAX-events is costly and inefficient. Therefore, it is necessary to develop efficient XML document processing models or techniques.

Typically, the amount of memory that DOM uses is five times the size of the original document. Lazy XML processing [6] and Apache Xerces's lazy processing [13] avoid parsing an entire document into memory by incrementally building a DOM-tree as different parts of the document are requested by the user. The result shows that reducing the size of a DOM-tree has the potential for improving the conventional XML processing model.

An XML streaming model, *e.g.*, SAX, consumes a constant and small amount of memory when parsing an XML document. SAX-based XPath processors, such as XSQ [5] and TurboXPath [12], have been proposed for querying or filtering streaming XML data. Generally, SAX-based XPath processors suffer from two drawbacks: their query algorithms are complex, and processing reverse axes (*e.g.*, ancestors) may require a significant amount of memory to maintain bookkeeping information. Moreover, current streaming models lack interaction mechanisms while parsing an XML document. Hence, significant computational resources are wasted on processing uninteresting data.

Many indexing techniques, such as structural summaries [9], path indexes [7], and edge indexes [4], have been proposed for improving the efficiency of XML query processing. They generally require large disk storage and complicated query algorithms. Some of the techniques rely on high performance indexing technologies provided by a relational/XML database management system (RDBMS/XDBMS). XDBMS, such as Berkeley DB XML and Natix [11], are designed for storing and manipulating XML documents. Although XDBMS- and RDBMS-based approaches provide efficient solutions for processing XML data, they are too expensive to integrate into small-scale applications. Moreover, they are intrusive and non-transparent; user applications need to be aware of the mechanics of the enhancement; and they typically require considerable modifications to integrate the enhancement.

In our previous work [3], we proposed an XML document prefiltering framework with the following characteristics:

- Accurate: it preserves the results of the original XPath processor.
- Efficient: it performs efficiently.
- Lightweight: it consumes few computational resources, such as CPU time, memory, and disk space.
- Transparent: it works transparently with existing applications. Users/applications need not be aware of its underlying mechanics.

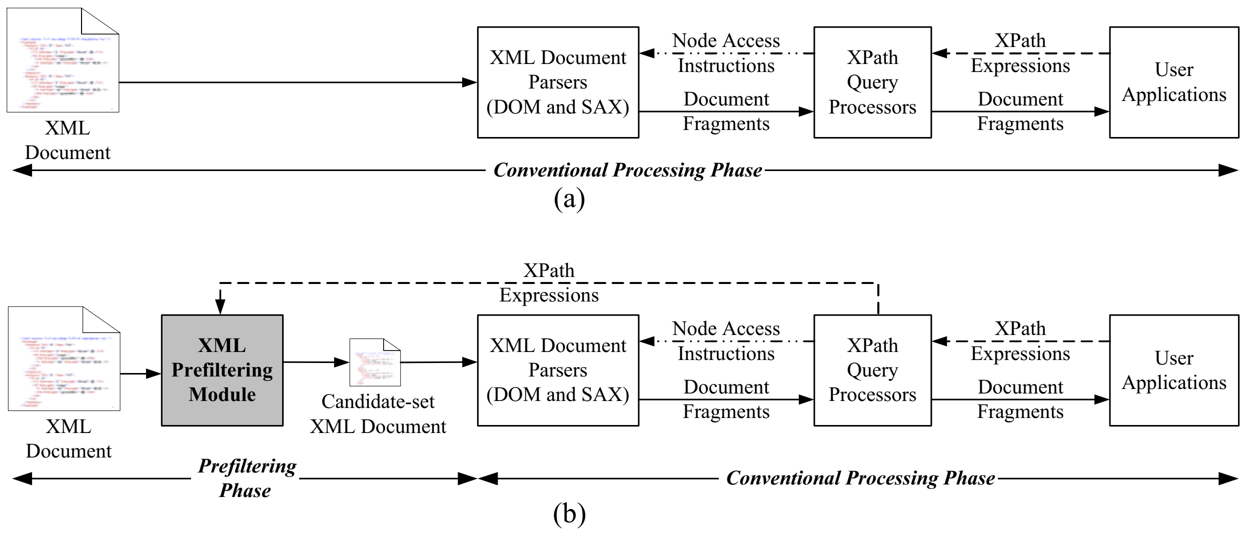


Figure 1. (a) The conventional XML processing model. (b) The two-phase XML processing model.

- Non-intrusive: it can be integrated into existing tools or applications with minimal modifications.
- Small-scale: it can be installed and operated in small-scale applications or devices (with limited computational resources), such as mobile phones, PDAs, or mote-level platforms in a sensor network.

In this demonstration, we realize a two-phase XML document processing model using the prefiltering technique. The contributions of our work summarized as follows.

- We present a two-phase XML processing model that provides a framework for improving the performance of the conventional XML processing model.
- We implement a small-scale, lightweight, and efficient prefiltering technique that improves XML processing. The prefilter possesses the characteristics of the prefiltering framework [3], and can be integrated into existing XML applications by simply adding one instruction.
- We demonstrate the integration of the prefiltering technique into an XML streaming parser that allows for XML parsing in a random access manner. To the best of our knowledge, no such implementation has been reported in the literature.
- We also demonstrate the integration of the prefiltering technique into DOM-based XPath/XQuery processors.
- A prototype of the prefiltering technique is implemented as JAVA packages, and is available at http://www.iis.sinica.edu.tw/~jashing/prefiltering/.

2. TWO-PHASE XML PROCESSING MODEL

The key idea of a two-phase XML processing model is to rapidly filter uninteresting fragments out of the input XML document before the actual application process it. The interesting fragments and some bookkeeping information (*e.g.*, document structures and pointers to uninteresting fragment) are returned to applications by the prefilter. As shown in Figure 1(b), the prefiltering module refines the input XML document.

2.1 XML Prefiltering Technique

The prefiltering technique employs a tiny search engine to extract candidate fragments from the input XML documents by approximately executing the user's query. The fragments are collected together with some bookkeeping metadata into candidate-set XML documents, which are returned to the user applications for the actual query processing. Uninteresting fragments are not returned, but they are still accessible by external fragment links specified in the candidate-set XML document.

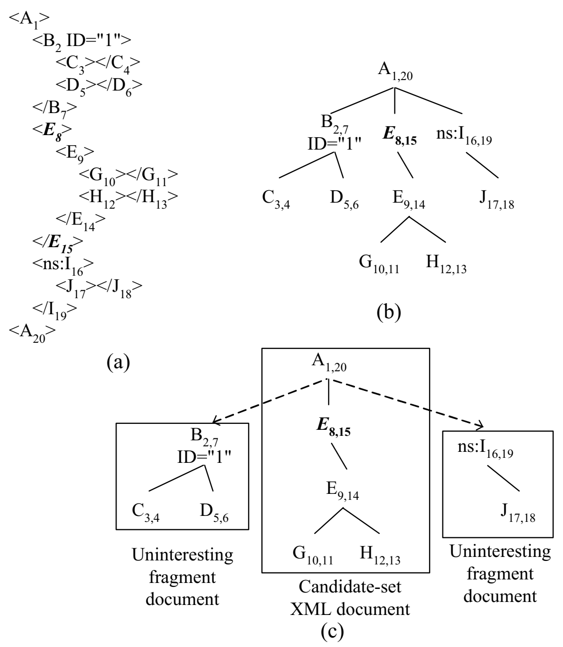
For example, when prefiltering the source XML document in Figure 2(a) over the XPath expression "/*A*/child::*E*", the candidate-set XML document in Figure 2(c) is returned. Two uninteresting fragments rooted at $B_{2,7}$ and $ns{:}I_{16,19}$ are filtered out. Also, two external fragment links (the dotted lines) that indicate the uninteresting fragment are added to node $A_{1,20}$.

2.2 System Architecture of the XML Prefiltering Technique

The XML prefiltering technique consists of five components. The *Indexer*, a preprocessing module, scans the XML document *D* and constructs an inverted index table. Next, in the prefiltering process, the *Query Simplifier* simplifies user XPath expressions (*XPEs*) to reduce the query evaluation time. In the third step, the *Fast Lightweight Step Analyzer*, a tiny search engine, determines the candidate fragments in *D* by evaluating the simplified *XPEs*. Those fragments and necessary document structures are then either transformed into a series of SAX-events by the *Micro XML Streaming Parser* or gathered into a candidate-set XML document *D'* by the *Fragment Gatherer*. Note that it is unnecessary to yield a physical file for *D'*. Instead, a memory-based input/output buffer is used to temporarily store data streams. More details are given in [3].

2.3 Index Scheme

Our previous research revealed that the size of an edge index is large and that edge join operations are expensive [4]. Therefore, we employ a node index scheme in the prefiltering technique.



```
<A1>
  <B2 ID="1">
    <C3></C4>
    <D5></D6>
  </B7>
  <E8>
    <E9>
      <G10></G11>
      <H12></H13>
    </E14>
  </E15>
  <ns:I16>
    <J17></J18>
  </I19>
<A20>
```

Figure 2. (a) An XML document. (b) The tree view of (a). (c) The candidate-set XML document prefiltered by the XPath expression "/*A*/child::*E*". Note that pre-order numbering is used to represent the start- and end-tag positions.

1. $S_{ancestor}[v](u, s, e, h) = \{(u', s', e', *) \mid s' < s \wedge e' > e \wedge u' = v\}$
2. $S_{ancestor\text{-}or\text{-}self}[v](u, s, e, h) = \{(u', s', e', *) \mid s' \leq s \wedge e' \geq e \wedge u'=v\}$
3. $S_{parent}[v](u, s, e, h) = \{(u', s', e', h') \mid s' < s \wedge e' > e \wedge h'=h-1 \wedge u'=v\}$
4. $S_{descendant}[v](u, s, e, h) = \{(u', s', e', *) \mid s' > s \wedge e' < e \wedge u'=v\}$
5. $S_{descendant\text{-}or\text{-}self}[v](u, s, e, h) = \{(u', s', e', *) \mid s' \geq s \wedge e' \leq e \wedge u'=v\}$
6. $S_{child}[v](u, s, e, h) = \{(u', s', e', h') \mid s' > s \wedge e' < e \wedge h'=h+1 \wedge u'=v\}$
7. $S_{preceding}[v](u, s, e, h) = \{(u', s', e', *) \mid e' < s \wedge u'=v\}$
8. $S_{following}[v](u, s, e, h) = \{(u', s', e', *) \mid e < s' \wedge u'=v\}$
9. $S_{following\text{-}sibling}[v](u, s, e, h) = \{(u', s', e', h') \mid e < s' \wedge (u', s', e', h') \in S_{child}[v](u'', s'', e'', h'') \wedge (u'', s'', e'', h'') \in S_{parent}[*](u, s, e, h) \wedge u'=v\}$
10. $S_{preceding\text{-}sibling}[v](u, s, e, h) = \{(u', s', e', h') \mid e' < s \wedge (u', s', e'\ h') \in S_{child}[v](u'', s'', e'', h'') \wedge (u'', s'', e'', h'') \in S_{parent}[*](u, s, e, h) \wedge u'=v\}$
11. $S_{self}[v](u, s, e, h) = \{(u', s', e', h') \mid s' = s \wedge e' = e \wedge u'=v \wedge h'=h\}$
12. $S_{attribute}[v](u, s, e, h) = \{(u_{att}', s', e', h') \mid s' = s \wedge e' = e \wedge h'=h \wedge u'=$ "attribute=value" $= v\}$

Figure 3. The XPath semantics.

Specifically, we index all elements and attributes of an XML document. Text nodes are ignored since the size of the index must remain small. Each record of the index has two fields: the *name* and the *position list*. The value of the *name* field is either an element name (including its namespace, *e.g.*, ns:I) or a string that is the concatenation of an attribute name and its value (*e.g.*, ID=1). The value of the *position list* is an ordered list of triples: (*start tag position*, *end tag position*, *height*), sorted by the *start tag position*. As a result, evaluating the user XPath expressions can be carried out efficiently by a binary search on the index.

2.4 Query Simplification and Evaluation

An input XPath query is simplified by removing certain steps. The last step, specifying the root node of a candidate fragment, is always preserved. The others, which are used to restrict the computed fragments, are removed selectively. Clearly, for each step removed, more and larger candidate fragments will be matched and returned. In our current implementation, we adopt a heuristic that eliminates steps with a low degree of selectivity (*i.e.*, matching a large number of elements). In particular, wildcard steps "/*" as well as those that require scanning the index or accessing the source document (see Section 2.5), are removed. The selectivity of a step can be calculated by computing the length of a position list selected by the step.

In the query evaluation, we adapt the semantics specification of XPath [8] for query evaluation. Let U denote the space of all tuples in the index. Evaluating the XPath expression "*u/axis::v*" over U is formalized in Figure 3. Here, u and v refer to two element names, and *axis* is any one of the XPath axes [14]. The tuple (u, s, e, h) represents a context node, consisting of an element u, its start tag position s, its end tag position e, and its height h.

2.5 Properties

Some properties of the prefiltering technique are as follows:

Property 1. All XPath axes can be computed in O(|*pos_list*(u)| log |*pos_list*(v)|), where |*pos_list*(u)| and |*pos_list*(v)| refer to the size of the *position lists* of u and v, respectively. Note that evaluating the *attribute* and *namespace* axes requires scanning the *name* field of the entire index.

Property 2. The node type of each XPath step must be specified; otherwise, it is necessary to scan the *position list* field of the entire index.

Property 3. The node-set, Booleans, number, and string function calls can be supported but it is necessary to access the source XML document and additional code. We do not consider these operations.

In the current implementation, to keep the system lightweight and small, we omit any XPath step that requires scanning the entire index.

3. DEMONSTRATIONS OVERVIEW

Our demonstration includes reporting the performance results and showing the source codes of the following systems: the prefiltering technique, DOM-based XPath/XQuery processors with the prefiltering technique, and an interactive SAX parser. An example of integrating the prefiltering technique into a Java DOM-based XPath processor by adding a single instruction is shown in Figure 4 (displayed in bold face). In addition, we demonstrate a GML-based (Geography Markup Language [10]) geographic information system that employs the prefiltering technique to speed up geospatial operations over large GML documents. For demonstration purposes, the system only uses small datasets at the server-side. It can be accessed at http://tsm.iis.sinica.edu.tw/~jashing/w3p/gmlsvg_pf/maps.php.

(1) ***Prefilter pf = new Prefilter (source_xmlfile, candidate-set_xmlfile, xpath_exp);*** // prefilter *source_xmlfile* and generate a candidate-set XML document.

(2) *InputSource in = new InputSource(new FileInputStream(**candidate-set_xmlfile**));* // use *candidate-set_xmlfile* as input to build up a DOM-tree.

(3) *DocumentBuilderFactory df = DocumentBuilderFactory.newInstance();*

(4) *Document doc = df.newDocumentBuilder().parse(in);* //build up the DOM-tree of the *candidate-set_xmlfile*.

(5) *NodeIterator nl = XPathAPI.selectNodeIterator(doc, xpath_exp);*// evaluate the *xpath_exp* against the DOM-tree.

Figure 4. A source code fragment of an XPath processor with prefiltering technique.

Table 1. Performance results of the query "*/site/regions/item[@id="item1"]/name*".

Methods / Datasets	Xerces XPath Processor with Lazy DOM Parser		Xerces XPath Processor with Prefiltering	
XMark (factor/size)	Memory Usages (MB)	Run Time (sec.)	Memory Usages (MB)	Run Time (sec.)
1/113MB	770	36.812	34.8	8.1
2/232MB	N/A	N/A	91.7	15.5
10/1,164MB	N/A	N/A	413.9	71.5
20/2,333MB	N/A	N/A	851.8	240.3
30/3,499MB	N/A	N/A	866.4	388.4

*N/A means that the method ran out of memory and did not finish.

Table 2. Performance results of the query "*/site/regions/asia*".

Methods / Datasets	Xerces XPath Processor with Lazy DOM Parser		Xerces XPath Processor with Prefiltering	
XMark (factor/size)	Memory Usages (MB)	Run Time (sec.)	Memory Usages (MB)	Run Time (sec.)
1/113MB	770	48.093	26.8	13.75
2/232MB	N/A	N/A	75.9	27.2
5/581MB	N/A	N/A	227.2	66
10/1,164MB	N/A	N/A	372	130.1
20/2,333MB	N/A	N/A	857.9	268.5
30/3,499MB	N/A	N/A	N/A	N/A

*N/A means that the method ran out of memory and did not finish.

Table 1 and Table 2 show the performance results of evaluating the queries "*/site/regions/item[@id="item1"]/name*" (matching one node) and "*/site/regions/asia*" (matching about 4.5% of nodes) against XML documents generated by XMark [1] on an Intel Pentium-4 PC running at 2.53GHz, with a 1GB DDR-RAM, a 120GB EIDE hard disk, and MS Windows 2000 server OS. Obviously, the XML prefiltering technique helps the XPath processors evaluate queries that return a few fragments from large documents. In addition, for streaming the processing model, the interactive SAX parser in the XML prefiltering technique can achieve nearly a ten-fold performance improvement when evaluating a query that selects a few fragments. In general, an overhead of about 10~15% (loading index and evaluating a simplified query) would be incurred for answering queries that return almost an entire document. In such cases, the complete source document would be returned directly.

4. ACKNOWLEDGMENTS

We would like to thank the reviewers for many valuable suggestions. This work was partially supported by the National Digital Archives Program (NSC95-2422-H-001-007), Taiwan. Chia-Hsin Huang is a Ph.D. candidate in the Department of Electronic Engineering, National Taiwan University of Science and Technology. He is supported by a graduate student fellowship from the Institute of Information Science, Academia Sinica, Taiwan. James Lu was supported by the National Science Council under grant NSC94-2811-Z-001 while visiting the Institute of Information Science, Academia Sinica, Taiwan, in 2005-2006.

5. REFERENCES

[1] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, *Centrum voor Wiskunde en Informatica*, 2001

[2] A. Vyas, M. Fernandez, and J. Simeon. The Simplest XML Storage Manager Ever. In *Informal Proc. of the 1st International Workshop on XQuery Implementation, Experience, and Perspectives*, 2004, pp. 37-42.

[3] C. H. Huang, T. R. Chuang, and H. M. Lee. Prefiltering techniques for efficient XML document processing. In *Proc. of the 2005 ACM Symposium on Document Engineering*, 2005, pp. 149-158.

[4] C. H. Huang, T. R. Chuang, and H. M. Lee. Fast Structural Query with Application to Chinese Treebank Sentence Retrieval. In *Proc. of the 2004 ACM Symposium on Document Engineering*, 2004, pp. 11-20.

[5] F. Peng and S. S. Chawathe. XSQ: A streaming XPath engine. *ACM Transactions on Database Systems*, *30*, 2, 2005, pp. 577-623

[6] M. L. Noga, S. Schott, and W. Löwe. Lazy XML processing. In *Proc. of the 2002 ACM Symposium on Document Engineering*, 2002, pp. 88-94

[7] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: A Path-based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Transactions on Internet Technology*, *1*, 1, 2001, pp. 110-141.

[8] P. Wadler, Two semantics for XPath. Tech. Report, Bell Labs, 2000. Available: http://homepages.inf.ed.ac.uk/wadler/papers/xpath-semantics/xpath-semantics.pdf

[9] Q. Zou, S. Liu, and W. W. Chu. Ctree: A Compact Tree for Indexing XML Data. In *Proc. of the 6th Annual ACM International Workshop on Web Information and Data Management*, 2004, pp. 39-46

[10] S. Cox, P. Daisey, R. Lake, C. Portele, and A. Whiteside, editors. OpenGIS© Geography Markup Language (GML) Implementation Specification, Version: 3.00, 2003.

[11] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *The VLDB Journal*, *11*, 4, 2002, pp. 292-314

[12] V. Josifovski, M. Fontoura, and A. Barta. Querying XML streams. *The VLDB Journal*, 14, 2, 2005, pp. 197-210

[13] Xerces Java Parser 2.8.0 Release. The Apache XML project. Available: http://xerces.apache.org/xerces2-j/.

[14] XML Path Language (XPath) Version 1.0, W3C Recommendation, 1999.