Prefiltering Techniques for Efficient XML Document Processing

Chia-Hsin Huang
jashing@iis.sinica.edu.tw
Institute of Information Science
Academia Sinica
Taipei 115, Taiwan

Tyng-Ruey Chuang
trc@iis.sinica.edu.tw
Department of Electronic Engineering
Department of Computer Science and Information Engineering
Taipei 106, Taiwan

Hahn-Ming Lee
hmlee@mail.ntust.edu.tw
National Taiwan University of Science and Technology

ABSTRACT
Document Object Model (DOM) and Simple API for XML (SAX) are the two major programming models for XML document processing. Each, however, has its own efficiency limitation. DOM assumes an in-core representation of XML documents which can be problematic for large documents. SAX needs to scan over the document in a linear manner in order to locate the interesting fragments. Previously, we have used tree-to-table mapping and indexing techniques to help answer structural queries to large, or large collections of XML documents. In this paper, we generalize the previous techniques into a prefiltering framework where repeated access to large XML documents can be efficiently carried out within the existing DOM and SAX models. The prefiltering framework essentially uses a tiny search engine to locate useful fragments in the target XML documents by approximately executing the user’s queries. Those fragments are gathered into a candidate-set XML document, and is returned to the user’s DOM- or SAX-based applications for further processing. This results in a practical and efficient model of XML processing, especially when the XML documents are large and infrequently updated, but are frequently being queried.

Categories and Subject Descriptors
F.2.2 [Theory of Computation]: Nonnumerical Algorithms and Problems – Pattern matching; H.3.3 [Information Systems]: Information Storage and Retrieval – Search process.

General Terms
Algorithms, Performance.

Keywords
Prefiltering, DOM, SAX, Structural Query, Two-phased XML processing model.

INTRODUCTION
The eXtensible Markup Language (XML) has been accepted as the standard for document representation and exchange over the Web. Document Object Model (DOM) [9] and Simple API for XML (SAX) [8] are the two major programming models for XML document processing. Each, however, has its own efficiency limitation. DOM assumes an in-core representation of XML documents which can be problematic for large documents. SAX needs to scan over the document in a linear manner in order to locate the interesting fragments. As a result, both DOM- and SAX-based applications may waste computational resources processing unnecessary document fragments.

XPath is one of the core components used in several XML-based query processors and tools. For example, XQuery [24] uses it for binding variables, XSLT [25] uses it for generating templates, and XPointer [23] uses it for addressing the internal structures of the XML documents. It is also commonly used in many applications to address parts of an XML document. An XPath expression consists of one or many location steps (or simply called as steps), each of which has three parts: an axis, a node test, and predicates [22]. The axis determines which nodes in the document tree are to be reached from the context node. The node test then selects the nodes that have the required tag name or node type. Finally, the predicates refine the result set. The result set is recursively treated to address parts of an XML document. An XPath expression “/child::A/ descendant::E”, abbreviated as “/A/E”, returns two sub-trees which are rooted at $E_{8,15}$ and $E_{9,14}$, respectively.

XPath often uses DOM as its data model, which can be problematic for large documents. Many indexing techniques, such as structural summaries [16][17], path indexes [14], and edge indexes [4], have been proposed for improving XML query efficiency. In our previous work [4], we used a tree-to-table mapping and the robust indexing techniques provided by relational database management system (RDBMS) to help answer structural queries to large, or large collections of, XML documents efficiently. However, almost all of the above schemes rely on expensive indexing schemes or external data storage systems that are unavailable in small-scale applications.

In this paper, we propose a prefiltering framework where repeated access to large XML document can be efficiently carried out within the existing DOM and SAX models. The prefiltering
framework essentially uses a tiny search engine to locate useful fragments in target XML documents by approximately executing the user’s queries. Those fragments are gathered into a candidate-set XML document, and is returned to the user’s DOM- or SAX-based applications for further processing. This results in a practical and efficient model of XML processing, especially when the XML documents are large and infrequently updated, but are frequently being queried. In addition, we have successfully integrated an XML streaming parser into the prefiltering technique. Our prefiltering technique enables the streaming parser to parse a document in a random access manner. To the best of our knowledge, enabling a streaming parser to provide random access ability has not been proposed and implemented until now. By using our enhanced streaming parser, the response times of two existing applications, one being a Chinese Treebank search engine [4], and the other one a geographic information systems (GIS) maps rendering system [5], have been greatly reduced. These experiment results will be shown in this paper.

The rest of this paper is organized as follows: Section 2 describes the details of our prefiltering framework, including the design philosophy and its system architecture. We also depict each module in detail in this section. In Section 3, we give two realistic applications to illustrate how practical our prefiltering framework is. In Section 4, we show the performance of the prefiltering framework as applied to three datasets. We discuss related work in Section 5. Finally, we conclude the paper in Section 6.

2. XML Prefiltering Framework

In this section, we describe the philosophy behind the development of the prefiltering framework. After that, we present the system architecture and show the details of each module.

2.1 Philosophy

We propose an XML prefiltering framework to improve the conventional XML processing model. It is a two-phased XML processing model, of which a prefiltering framework is included, as shown in Figure 1. In the prefiltering phase, the prefiltering framework extracts candidate fragments in the target XML document by rapidly and approximately executing a user XPath expression. Those candidate fragments are either gathered into a candidate-set XML document or transformed into SAX-events. Then the candidate-set XML document or the SAX-events are processed by the conventional XML processing models (i.e., DOM or SAX) to yield the results.

Several requirements and limitations of developing the prefiltering framework are briefly described as follows. Our prefiltering framework essentially employs a simple index scheme with approximate matching ability. Ideally, this framework has to work as transparently as possible so that it is easy to be used in DOM- and SAX-based applications. In our experience, the prefiltering framework can be integrated into SAX-based applications with little modifications. A key requirement of this prefiltering framework, however, is that it must guarantee a 100% recall rate in order to maintain the correctness of user applications. The main limitation of the prefiltering framework is that it can only be used in the applications that involve query processing. Moreover, as the prefiltering framework need to index the target XML documents and to execute user XPath expressions to extract candidate fragments, it is more suitable for the applications that dealing with infrequently updated and large XML documents. We now show how to integrate the prefiltering framework into the DOM and SAX processing models in the following sections.

2.1.1 Two-phased DOM Processing Model

The DOM processing model is shown in Figure 2 (a). First, before the XPath engine accesses the XML document, the DOM parser builds a DOM-tree in the main memory. After that, the XPath engine addresses document fragments by evaluating the XPath expressions issued from the user program. Even if the user program requires only a small part of the document, the DOM parser and the XPath engine have to process the entire document in order to determine the matched fragments. As a result, many resources, such as CPU time, memory, and disk I/O, may be wasted. Our prefiltering framework eases this problem by
reducing the number of mismatched fragments. As shown in Figure 2 (b), the user program also issues an XPath expression to our prefiltering engine. The prefiltering engine then selects candidate fragments in the document by approximately executing the XPath expression. Those candidate fragments are then gathered into a candidate-set XML document and is parsed for generating the necessary DOM-tree. After that, the XPath engine searches the DOM-tree for exact-match document fragments.

2.1.2 Two-phased Streaming Processing Model
SAX-based applications can be integrated into our prefiltering framework as well. Figure 2 (c) and Figure 2 (d) show, respectively, the SAX processing model and a SAX model with our prefiltering framework. In Figure 2 (c), the SAX parser transforms the entire XML document into a series of SAX-events. Meanwhile, the stream-based XPath engine evaluates the XPath expression issued from the user program. It then returns matched document fragments to the user program. In contrast, our prefiltering framework enables an XML streaming parser to parse the XML document in a random access manner. As shown in Figure 2 (d), the prefiltering framework selects candidate fragments in the document by approximately executing the XPath expression issued from the user program. Those candidate fragments are then transformed into SAX-events as input to the stream-based XPath engine. While a candidate fragment is being parsed, several flow-control instructions can be used for changing the parsing behavior (see Section 2.8). After that, the stream-based XPath engine returns to the user program document fragments that are exactly matched by the XPath expression.

2.2 System Architecture
Figure 3 shows the system architecture of the XML prefiltering framework. It consists of five major modules: the Indexer, the Query Simplifier (QS), the Fast Lightweight Steps-Axes Analyzer (FLISA4), the Fragment Gatherer (FG), and the Micro XML Streaming Parser (MXSP). The Indexer, a preprocessing module, scans the XML document and constructs an inverted index table that is to be used by FLISA4 to evaluate user XPath expressions. In the prefiltering process, the user application issues an XPath expression to the QS. After QS simplifies the XPath expression,
FLISA, a fast tiny search engine, determines the candidate fragments in the XML document by evaluating the simplified XPath expression. Those fragments either are transformed into a series of SAX-events by MXSP or are gathered into a candidate-set XML document by FG. Details of each module are described in the following sections.

2.3 Indexer
Building up an inverted index table of the XML document is the first step in our prefiltering framework. This indexing process only need to be executed once. After that, the table will be referenced by FLISA when evaluating the user queries. Note that users need not add extra instructions in their programs because this indexing process is executed automatically.

Each record in the table has two fields: name and position list. The value of the name field is either an element name or a string that is concatenated by an attribute name and its value (e.g., SIZE=10). The value of the position list is an ordered list; each element of the list is a pair of numbers: (start-tag position, end-tag position), i.e., the starting- and ending-byte offsets of an element or an attribute in the XML document. Furthermore, the position list is sorted by the start-tag position. As a result, by applying a binary search algorithm, FLISA can evaluate the user XPath expressions rapidly. Note that we can also check whether the XML document is well-formed so that our streaming parser, MXSP, can work more efficiently.

2.4 Query Simplifier
To reduce the cost of query evaluation, the user XPath expression is simplified by the Query Simplifier module (QS). The simplified XPath expression contains fewer steps and has simpler structure as compared to those of the original XPath expression. Therefore, the simplified XPath expression can be quickly evaluated. In general, the last step of a query specifies the root node of a candidate fragment within the target XML document. The others, called restriction steps, are used to filter out useless fragments that cannot be reached along these steps. That is, when there are fewer restriction steps, more candidate fragments will be matched and returned; this lowers the precision rate but increases the performance of the query evaluation. Over-simplifying the user query, however, may not benefit efficiently in our prefiltering technique. In Section 4, we will discuss this issue.

2.4.1 Simplification Rules
We suggest four simplification rules: (SR1) omitting internal steps; (SR2) omitting branch steps; (SR3) omitting wildcard steps; and (SR4) replacing the parent/child axes with the ancestor/descendant axes. We describe these four simplification rules below.

SR1: Omitting internal steps properly can greatly reduce the cost of query evaluation. The internal steps, i.e., the restriction steps, are used to filter out useless document fragments that cannot be reached. In our opinion, the most informative steps are the root step and the leaf step. The leaf step is the root node of a candidate fragment in the target XML document. Thus, we may simplify XPath expression by just keeping the root and the leaf step.

SR2: Any branch path (which may consist of many branch steps) can be omitted. Similarly, we may just omit the internal steps of branch paths.

SR3: The wildcard steps “*” may be eliminated.

SR4: The parent/child axes may be replaced with ancestor/descendant axes because our query evaluation algorithm can determine the ancestor/descendant relationships more efficiently than the parent/child relationships. However, this rule can cause the side-effect of duplicated parsing of a candidate fragment that is a sub-tree of yet another candidate fragment. For example, suppose Figure 4 (a) is the target XML document. The XPath expression “/A/child::E” can only match the sub-tree rooted by E\textsubscript{9} or E\textsubscript{14}. However, the simplified XPath expression “/A/ancestor::E” can match two sub-trees rooted by E\textsubscript{9} or E\textsubscript{14}, respectively. E\textsubscript{9} or E\textsubscript{14} would be parsed twice. This problem can be solved by avoiding taking overlapping fragments. That is, we can ignore a fragment whose start-tag position precedes the end-tag position of the just parsed fragment.

Any attribute-testing predicate clauses will be preserved as special steps. Those attribute-testing steps are very useful for evaluating the user query. Moreover, keeping a few restriction steps in the simplified XPath expression can greatly improve the precision rate but at the cost of slight increment of the query evaluation time. We find that by taking into consideration of the relative element frequency and density in an XML document, we can balance the precision rate with query evaluation time. In general, infrequent elements and non-uniformly distributed elements can be kept for the purpose of shrinking the size of candidate-set document. We will illustrate this in Section 4.2.

2.4.2 Examples of Simplification
We give some examples of applying the simplification rules. The Q\textsubscript{c} in Table 1 shows the original user XPath expression and others show several simplified XPath expressions. First, Q\textsubscript{c} can be simplified as Q\textsubscript{a} by omitting all the internal steps and changing the axis of the last step to descendant axis. This is the most simplistic way to simplify an XPath expression. Similarly, the results of completely omitting the branch steps and omitting the internal steps of the branch path are shown as Q\textsubscript{b1} and Q\textsubscript{b2}, respectively. Next, Q\textsubscript{d} shows the result of omitting the wildcard step with a branch path. In this case, its preceding step, Dex, inherits its predicate clause, and the axis of the first step of the branch path must be changed to “descendant.” Finally, the result of relaxing all the parent/child axes to the ancestor/descendant axes is shown as Q\textsubscript{e}.

<table>
<thead>
<tr>
<th>Query</th>
<th>Simplified XPath Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q\textsubscript{c}</td>
<td>/Root/ancestor::/child::[child::Bran1/child::Bran2]/ancestor::Anc/child::Leaf</td>
</tr>
<tr>
<td>Q\textsubscript{a}</td>
<td>/Root/ancestor::Leaf</td>
</tr>
<tr>
<td>Q\textsubscript{b1}</td>
<td>/Root/ancestor::/ancestor::Anc/child::Leaf</td>
</tr>
<tr>
<td>Q\textsubscript{b2}</td>
<td>/Root/ancestor::/child::[descendant::Bran2]/ancestor::Anc/child::Leaf</td>
</tr>
<tr>
<td>Q\textsubscript{d}</td>
<td>/Root/ancestor::/ancestor::A/child::/child::/child::Des/child::*[descendant::Bran2]/ancestor::Anc/child::Leaf</td>
</tr>
<tr>
<td>Q\textsubscript{e}</td>
<td>/Root/ancestor::/ancestor::A/child::/child::/child::Des/child::*[descendant::Bran2]/ancestor::Anc/child::Leaf</td>
</tr>
</tbody>
</table>

2.5 Fast Lightweight Steps-Axes Analyzer
The Fast Lightweight Steps-Axes Analyzer, FLISA, determines the candidate fragments in the XML document by evaluating the simplified XPath expression. Suppose the two steps of a piece of XPath expression are “u/axis::v”, where u and v refer to two
element names and \( axis \in \{\text{ancestor}, \text{descendant}, \text{preceding}, \text{following}, \text{ancestor-or-self}, \text{descendant-or-self}, \text{self}, \text{attribute}\} \). We discuss how to evaluate \( \text{"u/axis::v"} \) below.

Figure 4 (a) and (b) show an XML document and its tree view, respectively. We use preorder numbering as node index. These index numbers can be treated as tag positions. For example, 8 and 15 are the start- and end-tag positions of the boldface element \( E \), respectively; or simply, \( \text{start}(E) = 8 \) and \( \text{end}(E) = 15 \). We also use \( E(8, 15) \) to refer to the node \( E \). For the sake of convenience, we suppose that \( u \) has selected the node \( E \) as the context node and that \( v \) is a wildcard step. In this setting, \( E \) is ancestor, descendant, preceding, and following axes as, respectively, \( \{A_{1,20}\}, \{E_{8,14}, G_{10,11}, H_{12,13}\}, \{B_{2,5}, C_{3,4}, D_{6,1}\}, \) and \( \{I_{16,17}, J_{18,19}\} \).

We describe how \( FLISA \) evaluates the XPath expression with “descendant” axis, e.g., “\( u/axis::v \)”. Other axes can be evaluated similarly. First, \( u \) and \( v \) are used to select two \emph{position lists}, called \emph{parent list} and \emph{child list}, respectively, from the inverted index table. The query is then evaluated as follows: for each \( u \) in the \emph{parent list}, find all \( v \) in the \emph{child list} such that \( \text{start}(u) < \text{start}(v) \) and \( \text{end}(v) < \text{end}(u) \). Note that by applying the binary search algorithm, the complexity of query evaluation is \( O(n \log m) \), where \( n \) and \( m \) refer to the size of the parent list and child list, respectively. Table 2 shows four equations of evaluating “\( u/axis::v \)” where \( axis \in \{\text{ancestor}, \text{descendant}, \text{preceding}, \text{following}\} \). In the cases that \( axis \in \{\text{ancestor-or-self}, \text{descendant-or-self}, \text{self}\} \), they can also be evaluated by considering the current context nodes. Additionally, the attribute-testing steps as well as attribute axis can be evaluated by simply checking whether it is covered by the root step. That is, they are treated as descendant of the root step.

Rather than developing another structural query evaluation algorithm to deal with structural queries, we slightly modify the Parent-Child Relationship Filter (PCRF) algorithm developed in our previous work [4] and apply it to \( FLISA \). The design methodology of PCRF is to filter out ineligible candidate fragments from the XML document as soon as possible, and not to spend time evaluating them.

To apply the PCRF algorithm to \( FLISA \), the first and second steps of PCRF need to be slightly modified. The first step, indexing, is replaced by the Indexer module mentioned in Section 2.3. The index scheme used in prefiltering framework is much simpler than that used in the PCRF algorithm. In particular, we do not need a RDBMS. The second step, searching, is replaced by the Query Simplifier module described in Section 2.4.

### Table 2. The equations of evaluating “\( u/axis::v \)”

<table>
<thead>
<tr>
<th>No.</th>
<th>( axis )</th>
<th>Evaluation Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ancestor</td>
<td>( \text{start}(v) &lt; \text{start}(u) ) and ( \text{end}(v) &lt; \text{end}(u) )</td>
</tr>
<tr>
<td>2</td>
<td>Descendant</td>
<td>( \text{start}(u) &lt; \text{start}(v) ) and ( \text{end}(v) &lt; \text{end}(u) )</td>
</tr>
<tr>
<td>3</td>
<td>Preceding</td>
<td>( \text{end}(v) &lt; \text{start}(u) )</td>
</tr>
<tr>
<td>4</td>
<td>Following</td>
<td>( \text{end}(u) &lt; \text{start}(v) )</td>
</tr>
</tbody>
</table>

Generating the candidate fragments \( F \) is trivial because we have known the starting- and ending-byte offsets of the root node of a candidate fragment in \( D \). Generating the path information, however, need to parse over \( D \) and to calculate the descendant relationships between the current node \( p \) in \( D \) and the root nodes of \( F \). We start parsing from the root node of \( D \). When a start-tag is recognized, we used its position as a key to look up the corresponding end-tag position in the inverted index table. That \( p \in D \) if it contains any candidate fragment as its descendant or it is a candidate fragment; otherwise, the sub-tree rooted by \( p \) can be ignored by direct moving the file pointer to its end-tag position.

We use the example described previously at the end of Section 2.5 to show how \( FG \) works. In the example, candidate fragments \( F = \{E(8, 15)\} \) is the result proposed by \( FLISA \). Next, to generate \( D' \), \( FG \) executes the above procedures at \( A_1 \), the root of \( D \). First, it searches the end-tag position of \( A_1 \), and we get \( A(1, 20) \). \( A(1, 20) \in D' \) because it has the descendant \( E(8, 15) \). Then \( B(2, 7) \) is parsed, and the sub-tree rooted by \( B(2, 7) \) is ignored because it does not contain any candidate fragment. Next, \( E(8, 15) \) is parsed, and we find out that sub-tree rooted by \( E(8, 15) \) is in \( D' \) because itself is a candidate fragment. Finally, the sub-tree rooted by \( I(16, 19) \) is ignored because it does not contain any candidate fragment. Figure 4 (c) shows the final result.

Generating the path information could be inefficient if the user’s XPath expression contains the preceding, following, or sibling axes. An intuitive way is to keep all nodes of \( D \) that would be used to evaluate the user’s XPath expression. However, this may greatly increase the size of the refined XML document. Right now we have no efficient way to deal with these axes.

### 2.7 Micro XML Streaming Parser

The Micro XML Streaming Parser, \( M X S P \), takes responsibility for transforming the candidate fragments into SAX-events. This procedure is similar to \( FG \), but it generates SAX-events instead of a candidate-set XML document. We do not repeat the details here.
2.8 Additional Flow-Control Operators
We also design several flow-control operators for changing the behavior of MXSP while a candidate fragment is being parsed. We propose three flow-control operators: close-the-current-fragment (CCF), jump-to-the-next-fragment (JNF), and terminate-the-parsing-process (TPP). To realize these operators, a stack is used to keep track of the visited start tags.

The operator CCF forces MXSP to close the currently parsed fragment. Closing a fragment means stopping the parsing process and generating the corresponding end-tags. The operator JNF forces MXSP to escape from parsing the current fragment and proceed to parsing the next one. Note that this operator will leave the current fragment unclosed, so we have to use CCF to close it first, if necessary. Finally, the operator TPP forces MXSP to terminate the parsing process immediately.

Depending on the requirement of the user application, one can define other operators as well. For example, we also thought about designing a pull-based MXSP. That is, the application has to issue a command to request the next SAX-event, rather than using an event-handler to process SAX-events passively.

2.9 XML Fragment Interchange
We can also use W3C’s XML Fragment Interchange specification (a candidate recommendation) [21], to represent our candidate-set document. The major advantage of using this representation is that fragment-aware applications can manipulate the interesting fragments directly (each of which is a well-balanced XML document), instead of dealing with the entire document. As defined in the specification, an XML document can be divided into two parts: the fragment(s) and the fragment context specification. Similar to our candidate fragment, a fragment is a sub-tree of the source XML document. After removing fragment(s) from the source document and adding the namespace declaration and some linking metadata (i.e., the fragbody element) into it, the resulting document is called the fragment context specification.

Obviously, our prefiltering framework can be viewed as an implementation of the Fragment Interchange specification. The procedure is similar to what the Fragment Gatherer does. We just need to parse the original XML document, add the Fragment Interchange namespace declaration in the beginning of the source document, store each fragment into a file, and add linking metadata to associate each fragment file with the original document.

3. APPLICATIONS
To show that our prefiltering framework is practical, we apply it to two existing applications, a Chinese sentence search engine [4] and an XML-based GIS system [5]. We introduce them briefly in the following sections.

3.1 Chinese Treebank Search Engine
In our previous work [4], we designed a fast structural query algorithm, the Parent-Child Relationship Filter (PCRF) algorithm for querying as well as analyzing the CKIP Chinese Treebank corpus [12]. Taking the advantages provided by a RDBMS, PCRF is able to attain outstanding performance. We also proposed a Stream-based PCRF algorithm using a SAX parser to serialize the XML document and to provide query abilities similar to those provided by PCRF.

A user query $Q$ in the application is usually expressed by a tree structure as shown in Table 4. The answer to be returned by $Q$ is a set of sentences, each of which, exactly or approximately (if advanced options are enabled), contains the structure matched by $Q$. Using our prefiltering framework, the Stream-based PCRF can skip many ineligible sentences thus improves query performance.

3.2 XML-based GIS System
The goal of the XML-based geographic information system (GIS) system described in [5] is to develop a schematic mapping from the Standard Exchange Format (SEF) to the Geography Markup Language (GML) [19], hence to move the existing GIS bases into XML domains. Another structural mapping from GML data to Scalable Vector Graphics (SVG) [11] data was also developed for visualizing the GML-ized maps. In the GML document, a set of geometric objects $O$, e.g., buildings, roads, rivers, mountains, etc., are represented in the leaf nodes. Their geometric features $F$, e.g., the types of layers they belong to, etc., are represented in the path from the root node to the geometric objects themselves. The coordinates of a geometric object $o$ in $O$ are represented by a numerical list of arbitrary length.

A user query $Q$ in the system consists of two parts: an XPath expression and a query range window $W$. $W$ is represented by the most top-right and the most bottom-left coordinates of the window. In the query evaluation, the XPath expression is used to select $O$. Individual geometric object $o$ in set $O$ will be proposed as an answer to $Q$ if its shape (determined by its coordinates) intersects with $W$. The intersecting evaluation may be time consuming because the number of coordinates of $o$ may be very large. Fortunately, $\text{BoundedBy}$ nodes, defined in the GML specification [19], can be used to lower this cost. The $\text{BoundedBy}$ node frames the shape of $o$ by using a pair of the most top-right and the most bottom-left points. In this setting, one can first check whether the $\text{BoundedBy}$ node of $o$ intersects with $W$; if that is not the case, $o$ can be omitted.

The application uses a standard SAX parser and a set of library to evaluate a user query $Q$. Originally, all SAX-events of the GML document must be handled in order to see if the returned geometric object has the structure and shape required by $Q$. However, by applying our prefiltering framework, only the candidate fragments are made to generate SAX-events. Furthermore, two flow-control operators, CCF and JNF, are used to force MXSP to skip candidate fragments if their $\text{BoundedBy}$ child nodes cannot intersect with $W$. Because the number of coordinates of a geometric object $o$ may be very large (e.g., when $o$ has a complex shape), omitting them can reduce a lot of disk I/O and save time. As a result, the performance of the query evaluation is improved greatly.

4. PERFORMANCE ANALYSIS
We conduct three experiments to demonstrate the performance and characteristics of our prefiltering framework.

4.1 The Environment and the Datasets
Our setup is an Intel Pentium-4 PC running at 2.53GHz, with a 1GB DDR-RAM, a 120GB EIDE hard disk, and the MS Windows

1 “SEF is proposed by the Ministry of the Interior, Taiwan, in 1998 as a data exchange standard for topographic maps,” see [5].
XPath expressions simplify the six XPath expressions in Table 4. The simplified ancestor/descendant axes and omitting the internal steps, to identify (using preorder numbering), and the numbers of graphic views (the numbers beside the nodes are the node identifiers using preorder numbering), and the numbers of nodes using preorder numbering are reported in Table 3. The running time reported in this paper is an average of five runs.

In our experience, the refined XML documents are tens to hundreds times smaller in size than that of the source documents. As a result, the total execution time, including the time for evaluating user’s query and parsing the refined documents, is reduced three-to ten-fold.

### 4.2 Performance Results of the Simplification Rules

We used the CKIP Chinese Treebank corpus and six queries to test the performance of these simplification rules mentioned in Section 2.4.1. Table 4 shows the six XPath expressions, their graphic views (the numbers beside the nodes are the node identifiers using preorder numbering), and the numbers of matched sentences. The root node of each query is “Sentence”, which does not show up in the graphic views. None of them contains a following-sibling axis; that is, the sibling ordering of matched sentences is unimportant.

We applied the simplification rule 4 (SR4), i.e., replacing parent/child axes with ancestor/descendant axes, to simplify the six XPath expressions in Table 4. The simplified XPath expressions Q1SR4 to Q6SR4 are shown in Table 5, and their performance numbers are reported in Table 6. Similarly, we applied the SR4 and SR1, i.e., replacing parent/child axes with ancestor/descendant axes and omitting the internal steps, to simplify the six XPath expressions in Table 4. The simplified XPath expressions Q1SR14 to Q6SR14 are shown in Table 5, and their performance numbers are reported in Table 7. For example, Q2SR4 and Q2SR14 are “//Sentence[.//VP[.//NP[.//N and ./DM]] and C and D and PP[.//NP[.//VP[.//NP[.//VP]]]]]” and “//Sentence[.//NP[N and ./DM]]”, respectively. Note that the cost of loading the index file is 1.333 seconds which has not been included in the results in Table 6, Table 7, and Table 9.

For Q1SR4, our prefiltering engine spent 16.608 and 0.001 seconds matching and parsing 1 sentence with 18 nodes, and it consumed 61/60MB of physical/virtual memory, respectively. Notice the Q1SR4 and Q3SR4 did not benefit from using prefiltering engine because they spent too much time in either evaluating the query or in parsing candidate fragments. As shown in Table 7, because the internal steps of the Q1SR4 had been removed, the cost of query evaluation of Q1SR14 was greatly reduced, from 16.608 to 1.625 second.

After omitting all of the internal steps, in general, the query evaluation is improved. However, Q3SR14 still cannot be improved. That is because almost all of the sentences in the corpus contain the “.//NP/child::NP” structure, and as a result, almost the entire document need to be parsed. It also suffered from expending additional effort on moving the file pointer frequently.

We believe that by considering the relative element frequency and its density (distribution) when simplifying a query can prevent our prefiltering framework from this undesired side-effect. If the leaf
Table 5. The simplified XPath expressions of Table 4

<table>
<thead>
<tr>
<th>Query</th>
<th>Simplified XPath Expression by Applying SR4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1SR4</td>
<td>//Sentence/[.//NP//N and .//C and .//* and .//*] and .//VP//V and .//N] and .//VH and .//VP//V/[.//VH and .//DE and .//VA]]</td>
</tr>
<tr>
<td>Q2SR4</td>
<td>//Sentence/[.//NP//N]</td>
</tr>
<tr>
<td>Q3SR4</td>
<td>//Sentence/[.//NP and .//* and .//* and .//* and .//* and .//*]</td>
</tr>
<tr>
<td>Q4SR4</td>
<td>//Sentence/[.//NP and .//D and .//VC and .//*]</td>
</tr>
<tr>
<td>Q5SR4</td>
<td>//Sentence/[.//VP//NP//NP//N]</td>
</tr>
<tr>
<td>Q6SR4</td>
<td>//Sentence/[.//EMPTY]</td>
</tr>
</tbody>
</table>

Table 6. Performance numbers resulting from replacing parent/child axes with the ancestor/descendant axes

<table>
<thead>
<tr>
<th>Query</th>
<th>Cost of Matching (sec.)</th>
<th>Cost of Parsing (sec.)</th>
<th>#Subtrees</th>
<th>#Nodes</th>
<th>MM/VM (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1SR4</td>
<td>16.068</td>
<td>0.001</td>
<td>1</td>
<td>18</td>
<td>61/60</td>
</tr>
<tr>
<td>Q2SR4</td>
<td>9.313</td>
<td>1.265</td>
<td>1,125</td>
<td>16,260</td>
<td>47/46</td>
</tr>
<tr>
<td>Q3SR4</td>
<td>8.6245</td>
<td>17.0625</td>
<td>17,603</td>
<td>214,124</td>
<td>38/37</td>
</tr>
<tr>
<td>Q4SR4</td>
<td>5.015</td>
<td>2.5</td>
<td>2,018</td>
<td>33,498</td>
<td>32/30</td>
</tr>
<tr>
<td>Q5SR4</td>
<td>8.907</td>
<td>0.562</td>
<td>344</td>
<td>7,846</td>
<td>49/48</td>
</tr>
<tr>
<td>Q6SR4</td>
<td>0.218</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>18/17</td>
</tr>
</tbody>
</table>

Table 7. Performance numbers resulting from replacing parent/child axes with ancestor/descendant axes, and omitting all internal steps of the queries

<table>
<thead>
<tr>
<th>Query</th>
<th>Cost of Matching (sec.)</th>
<th>Cost of Parsing (sec.)</th>
<th>#Subtrees</th>
<th>#Nodes</th>
<th>MM/VM (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1SR1</td>
<td>1.625</td>
<td>0.062</td>
<td>23</td>
<td>646</td>
<td>60/59</td>
</tr>
<tr>
<td>Q2SR1</td>
<td>2.532</td>
<td>3.921</td>
<td>3,387</td>
<td>47,186</td>
<td>46/45</td>
</tr>
<tr>
<td>Q3SR1</td>
<td>0.516</td>
<td>17.562</td>
<td>19,274</td>
<td>223,263</td>
<td>38/37</td>
</tr>
<tr>
<td>Q4SR1</td>
<td>2.375</td>
<td>4.562</td>
<td>3,846</td>
<td>58,136</td>
<td>32/30</td>
</tr>
<tr>
<td>Q5SR1</td>
<td>0.64</td>
<td>17.578</td>
<td>19,274</td>
<td>223,263</td>
<td>38/37</td>
</tr>
<tr>
<td>Q6SR1</td>
<td>0.218</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>18/17</td>
</tr>
</tbody>
</table>

Table 8. Simplified XPath expressions of Q1 and Q5; by replacing parent/child axes with ancestor/descendant axes, and by omitting half of steps with consideration to element frequencies

<table>
<thead>
<tr>
<th>Query</th>
<th>Simplified XPath Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1SR4</td>
<td>//Sentence/[.//NP//N and .//C and .//* and .//*] and .//VP//V and .//N] [.//VH and .//VP//V/[.//VH and .//DE and .//VA]]</td>
</tr>
<tr>
<td>Q5SR4</td>
<td>//Sentence/[.//NP//NP//NP//NP]</td>
</tr>
</tbody>
</table>

Table 9. Performance numbers resulting from replacing parent/child axes with ancestor/descendant axes, and removing half of steps with consideration to element frequencies

<table>
<thead>
<tr>
<th>Query</th>
<th>Cost of Matching (sec.)</th>
<th>Cost of Parsing (sec.)</th>
<th>#Subtrees</th>
<th>#Nodes</th>
<th>MM/VM (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1SR4</td>
<td>2.906</td>
<td>1.859</td>
<td>1,405</td>
<td>38/37</td>
<td></td>
</tr>
<tr>
<td>Q2SR4</td>
<td>2.291</td>
<td>0.265</td>
<td>97</td>
<td>25/23</td>
<td></td>
</tr>
<tr>
<td>Q3SR4</td>
<td>1.858</td>
<td>0.001</td>
<td>3</td>
<td>21/20</td>
<td></td>
</tr>
<tr>
<td>Q4SR4</td>
<td>0.64</td>
<td>17.578</td>
<td>19,274</td>
<td>38/37</td>
<td></td>
</tr>
<tr>
<td>Q5SR4</td>
<td>2.734</td>
<td>11.562</td>
<td>11,043</td>
<td>30/29</td>
<td></td>
</tr>
<tr>
<td>Q6SR4</td>
<td>2.25</td>
<td>10.375</td>
<td>9,555</td>
<td>23/24</td>
<td></td>
</tr>
</tbody>
</table>

Table 10. Element Frequencies of the CKIP Chinese Treebank

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Frequency</th>
<th>Element Name</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>53,718</td>
<td>DE</td>
<td>6,735</td>
</tr>
<tr>
<td>NP</td>
<td>43,141</td>
<td>C</td>
<td>5,581</td>
</tr>
<tr>
<td>VP</td>
<td>19,703</td>
<td>PP</td>
<td>5,166</td>
</tr>
<tr>
<td>D</td>
<td>14,327</td>
<td>P</td>
<td>5,137</td>
</tr>
<tr>
<td>S</td>
<td>12,235</td>
<td>VA</td>
<td>2,293</td>
</tr>
<tr>
<td>VH</td>
<td>8,158</td>
<td>V</td>
<td>333</td>
</tr>
</tbody>
</table>

and low frequencies steps, respectively. We report their performance numbers in Table 9. The element frequencies of the CKIP Chinese Treebank are listed in Table 10. It is of no surprise that by using the infrequent elements, one can shrink large portion of the XML documents. They also consumed less memory.

4.3 Performance Results of the Flow-Control Operators

In this experiment, we used the GML document as the dataset to show the performance improvement by using the proposed flow-control operators. The XPath expression was to find all buildings in the query range window $W$ whose top-right and bottom-left points are (305500, 2767060) and (305600, 2767100), within 20,000 square meters. The XPath expression was simplified by replacing its parent/child axes with the ancestor/descendant axes and by omitting the internal steps. The XPath expression can select 47,522 buildings (fragments), each of which is a geometric object $o$. We first check whether the $BoundedException$ node of $o$ can interest with $W$. If so, $MXSP$ will keep parsing the sub-tree. If not, a jump-to-the-next-fragment ($JNF$) command is sent to $MXSP$, and the rest of the sub-tree will not be parsed. We report the
The time to load the index file, 22.85 seconds, has been included in the total of using MXSP framework.

4.4 Performance Results of Attribute-Testing

In this experiment, we used QI in [3], “/site/regions/namerica/item[@id = “item20748”]/name”, as the testing XPath expression to query the XMark document. The XPath expression was simplified by replacing its parent/child axes with the ancestor/descendant axes, and by omitting the internal steps. The simplified XPath expression was “/site/name” with an attribute-testing step [@id = “item20748”] that is treated as a descendant of the root step site. That is we just check whether its start- and end-tag positions are covered by those of site step (see Section 2.5).

Three SAX parsers, MXSP, Expat SAX, and Primitive SAX, were used to test their performance. We designed two experiments: one that used attribute-testing steps, and the other one that did not. The results are reported in Table 12. Obviously, by using our prefiltering framework with attribute-testing step, it outperformed the others. It took 25.214 seconds to generate the candidate-set document which has only five nodes.

5. DISCUSSION AND RELATED WORK

There are numerous research, such as XTrie [6], XFilter [13], and YFilter [26], that have been proposed to filter streaming XML data. All of them focused on building large-scaled XML filtering systems such as the publish/subscribe system. They index users’ XPath expressions (as profiles) by using trie data structure or the NFA (Nondeterministic Finite Automata). The incoming data is then matched against those query indexes. These filtering approaches do not aim to filter a part of the XML document.

XML DTDs (Document Type Definition) or schemas are useful for formulating queries, browsing data structures, and enabling query optimizations [17]. A prefiltering concept using XML DTDs was mentioned in [7]. The Stream Index (SIX) [1] is defined as a table of (start-tag, end-tag) offsets that are used to filter streaming XML data. If an incoming element does not occur in the SIX table, a query processor can use the end-tag offset to skip the following data without parsing and evaluating them.

Many research topics are closely related to our prefiltering framework as well. Several research work has focused on the topics of indexing schemes. Some of them provided the structural summaries [16][17], and some of them relied on the high performance indexing technologies provided by RDBMS or R-tree data structures [4][14][15][20]. The latter ones usually focused on mapping the structure of an XML document to the relation tables. User queries are then transformed into SQL expressions to query those relation tables. As RDBMS is unavailable in many applications, they are too expensive to be used in a prefiltering framework.

Node identifiers are usually used as the keys for joining among the matched records while evaluating user queries. Several complicated identifying methods, such as combined preordered/postordered numbering [20], extended preoracle/range of descendants numbering [15], and combined start- and end-tag positions numbering [14], have certain properties that help efficient evaluation of ancestor/descendant or parent/child relationships among nodes. In more detail, in [20], they use spatial indexing technique, i.e., R-tree, as index schemes and obtain outstanding performance.

Streaming parsers can be divided into two groups: the pull-based model and the push-based model [2]. The commonly used SAX parser [8] is categorized as a push-based one that triggers SAX-events actively. The pull-based parser, on the other hand, parses and returns an event when it receives a request from the applications. This motives us to integrate our prefiltering framework into a XML streaming parser and hence enables the parser to parse a document in a random access manner.

Finally, simplifying queries can reduce the cost of query evaluation. The flexible structure and full-text search engine of [18] also uses query relaxation to reach the goal of approximate matching. As a comparison, we simplify queries in order to reduce the cost of query evaluation time.

6. CONCLUSIONS

We conclude by outlining some possible lines of future research. First, the memory usage should be reduced. Our prefiltering
framework consumes much memory resource. Lower memory usage can make it more practical. Second, generating path information is a big challenge if user’s XPath expression contains preceding, following, or sibling axes, as mentioned in Section 2.6. This greatly limits the flexibility of our prefiltering framework. Maybe transforming user’s XPath expression into a backward axes free one [9] can ease this limitation. Third, though we have successfully integrated the prefiltering framework into an XML streaming parser; however, we have not completed the integration of the framework with DOM-based XML applications. That is, to incorporate the prefiltering framework into the cache of the existing DOM and SAX modules. Finally, one can also integrate our prefiltering framework into DOM- and stream-based XPath processors. We believe that the enhanced DOM-based XPath processors will be able to handle large XML documents in more efficient way.

7. ACKNOWLEDGEMENT
This work is partially supported by the National Digital Archives Program, Taiwan and the National Science Council.
We would like to thank the reviewers who gave us many valuable suggestions. Chia-Hsin Huang is a Ph.D. candidate in the Department of Electronic Engineering of National Taiwan University of Science and Technology, and is supported by a graduate student fellowship from Academia Sinica.

8. REFERENCES