Beyond Lazy XML Parsing[1]

Fernando Farfán, Vagelis Hristidis, Raju Rangaswami

School of Computer and Information Sciences
Florida International University
`{ffarfan,vagelis,raju}@cis.fiu.edu`

Abstract. XML has become the standard format for data representation and exchange in domains ranging from Web to desktop applications. However, wide adoption of XML is hindered by inefficient document-parsing methods. Recent work on lazy parsing is a major step towards alleviating this problem. However, lazy parsers must still read the entire XML document in order to extract the overall document structure, due to the lack of internal navigation pointers inside XML documents. Further, these parsers must load and parse the entire virtual document tree into memory during XML query processing. These overheads significantly degrade the performance of navigation operations. We have developed a framework for efficient XML parsing based on the idea of placing internal physical pointers within the document, which allows skipping large portions of the document during parsing. The internal pointers are generated in a way that optimizes parsing for common navigation patterns. A double-Lazy Parser (2LP) is then used to parse the document that exploits the internal pointers. To create the internal pointers, we use constructs supported by the current W3C XML standard. We study our pointer generation and parsing algorithms both theoretically and experimentally, and show that they perform considerably better than existing approaches.

Keywords: XML, Document Object Model, Double Lazy Parsing, Deferred Expansion, XPath.

1 Introduction

XML has become the de facto standard format for data representation and exchange in domains ranging from the Web to desktop applications. Examples of XML-based document types include Geographic Information Systems Markup Language (GML) [8], Medical Markup Language (MML) [17], HL7 [10], and Open Document Format (ODF) [21]. This widespread use of XML requires efficient parsing techniques. The importance of efficient XML parsing methods was underscored by Nicola and John [19], showing that the parsing stage is processor and memory consuming, needing main memory as much as ten times the size of the original document.

There are two de facto XML parsing APIs, DOM [2] and SAX [22]. SAX reads the whole document and generates a sequence of events according to the nesting of the elements, and hence it is not possible to skip reading parts of the document as this would change the semantics of the API. On the other hand, DOM allows users to explicitly navigate in the XML document using methods like `getFirstChild()` and

[1] This project was supported in part by the National Science Foundation Grant IIS-0534530 and by the Department of Energy Grant ER25739.

`getNextSibling()`. DOM is the most popular interface to traverse XML documents because of its ease of use. Unfortunately, its implementation is inefficient since entire subtrees cannot be skipped when a method like `getNextSibling()` is invoked. This also leads to frequent "Out of Memory" exceptions. In contrast to SAX, parsing a document using DOM could potentially avoid reading the whole document as the sequence of navigation methods may only request to access a small subset of the document. In this work we focus on parsing using a DOM-like interface.

Lazy XML parsing has been proposed (e.g., [25, 20]) to improve the performance of the parsing process by avoiding the loading of unnecessary elements. This approach substitutes the traditional eager evaluation with a lazy evaluation as used by functional programming languages [1]. The architecture shown in Figure 1, based on the terminology of [20], consists of two stages. First, a pre-parsing stage extracts a virtual document tree, which stores only node types, hierarchical structure information, and references to the textual representation of each node. After this structure is obtained, a progressive parsing engine refines this virtual tree on demand, expanding the original virtual nodes into complete nodes with values, attributes, etc. as they are needed.

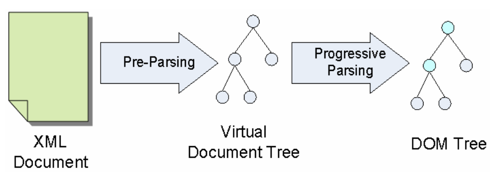


Figure 1. Lazy XML Parser Architecture

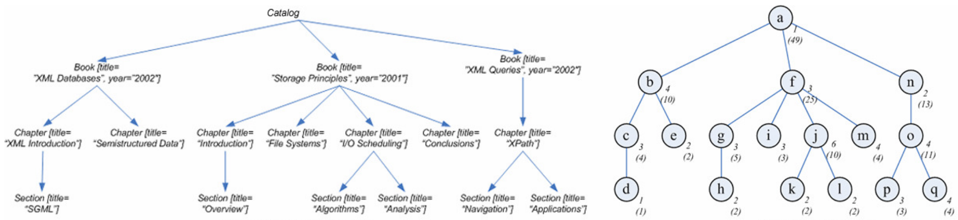
Clearly, the lazy parsing technique is a significant improvement. However, it still suffers from the high initial cost of pre-parsing (Figure 1) where the whole document must be read before the lazy/progressive parsing starts. The pre-parsing stage is inevitable due to the lack of internal physical pointers (or something equivalent) within the XML document. Further, the entire virtual document tree must be loaded and processed in main memory during the progressive-parsing stage, i.e. during query processing.

Overview of Approach: We call our XML parsing approach double-Lazy Parsing (2LP) because both stages in Figure 1 are lazy, in contrast to previous works where only the second stage is lazy. The first stage is performed offline, when the document is partitioned into a set of smaller XML files, then interlinked using XInclude [26] pointers. The optimal partition size is computed by considering the random versus sequential access characteristics of a hard disk.

The second stage parses a partitioned document, reading a minimal set of partitions to perform the sequence of navigation commands. 2LP loads (pre-parses using the terminology of Figure 1) the partitions in a lazy manner (only when absolutely necessary). In the case of DOM, we maintain an overall DOM tree $D(T)$ which is initially the DOM tree of the root partition P_0 of the XML tree T. Then $D(T)$ is augmented with the DOM trees $D(P_i)$ of the loaded partitions P_i.

Our approach dramatically reduces the cost of the pre-parsing stage by only pre-parsing a typically small subset of the partitions. Furthermore, our approach leads to significantly faster progressive-parsing times than traditional lazy parsing, as we show experimentally, due to the fact that whole subtrees are skipped.

To complement lazy partition loading, our approach also performs lazy unloading of inactive partitions (described in Section 2) if the total amount of main memory used by the DOM tree exceeds a threshold. Hence, in addition to a fast pre-parsing stage, our method also allows DOM-based parsing with limited memory resources. Note that previous lazy parsing techniques can also in principle achieve this to a smaller degree;



(a) A sample XML Document (b) Corresponding tree

Figure 2. Sample XML Document and Corresponding Tree

the virtual document tree must still be stored in memory at all time. However, this optimization is not used in the current implementation of the Xerces DOM parser.

A drawback of the 2LP approach is that the XML document is split into a set of smaller XML documents/files. Unfortunately, the XML standard does not support an alternative physical pointer construct (XPointer [29] is logical and not physical) due to the complication this would incur during cross-platform document exchange. We argue and demonstrate in the rest of this paper that the performance gains in XML document navigation far outweigh the drawbacks of document splitting. Further, if physical pointers are introduced for XML in the future, our work can be immediately applied.

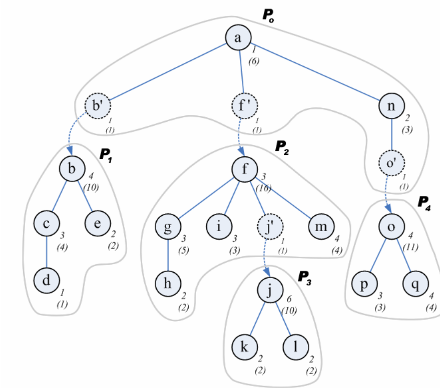
This paper makes the following contributions: *(1)* We develop a framework to allow efficient XML parsing, which improves both the pre-parsing and progressive parsing time as well as the memory requirements of both parsing phases. *(2)* We present algorithms to perform partitioning and double-Lazy XML Parsing (2LP) for DOM-like navigation. Note that 2LP-enabled documents are backward compatible i.e., they can be parsed by current XML parsers. *(3)* We show how the theoretically optimal partition size can be computed assuming knowledge of the navigation patterns on complete XML trees and the hard disk characteristics. *(4)* We study our partitioning and parsing algorithms both theoretically and experimentally. Experiments on various XML navigation patterns, including XPath, confirm our theoretical results and show consistent and often dramatic improvement in the parsing times.

The rest of the paper is organized as follows: We describe our double-Lazy parsing techniques in Section 2. Section 3 presents techniques for partitioning the original document into smaller subtrees. Our experiments are discussed in Section 4. We present related work in Section 5. Finally, Section 6 discusses our conclusions and future work.

2 2LP on Partitioned XML Documents

Let T be the original XML document, and P_0, … P_n be the partitions to which *T* was split during the partitioning stage (elaborated in Section 3). P_0 is the root partition, since it contains the root element of *T*. Figure 3(a) shows an example of a partitioned XML tree. All the partitions are connected by XInclude elements, containing the URI to the partition file. The XInclude elements are represented in the figure by *b′*, *f′*, *j′*. Note that by creating a partition (e.g., P_2), the key result is that we facilitate skipping the subtree rooted at this partition. That is, by creating partition P_2 we can access directly node *a* from node *f′* to node *n*.

The XML representation of two of the partitions in Figure 3(a) is shown in Figure 3(b). Partition P_0 contains the document root and is then the root partition. The subtree rooted at the first *Book* element was partitioned and the *Book* element has been replaced by the XInclude pointer to the XML document of Partition P_1. This additional element



(a) Tree Partitions

```
partition0.xml
<Catalog>
 <xi:include href="partition1.xml"
  xmlns:xi="http://www.w3.org/2001/XInclude"/>
 <xi:include href="partition2.xml"
  xmlns:xi="http://www.w3.org/2001/XInclude"/>
 <Book title="XML Queries" year="2002">
  <xi:include href="partition4.xml"
   xmlns:xi="http://www.w3.org/2001/XInclude"/>
 </Book>
</Catalog>

partition1.xml
<Book title="XML Databases" year="2002">
    <Chapter title="XML Introduction">
         <Section title="SGML" />
    </Chapter>
    <Chapter title="XML Introduction" />
</Book>
```

(b) Sample Document Partitions

Figure 3. Partitioned XML Tree and Document Partitions

added to the tree upon partitioning will hold the reference to the root of the partition's subtree. We explain this in detail in Section 3.

Figure 4 describes the process of loading (pre-parsing) a partition. After loading a partition, progressive parsing occurs as needed. The `loadPartition()` method replaces, in the working DOM tree, the XInclude pointer element e with the DOM tree of the partition that *e* points to.

To ensure the double-lazy processing of the partitions, we need to decide when it is absolutely necessary for a partition to be loaded. Intuitively, a partition must be loaded when a navigation method (e.g., `getFirstChild()`) cannot be executed without doing so, that is, the return value of the method cannot be computed otherwise.

```
procedure loadPartition(XIncludeElement e) {
    newPartitionRoot =
          preParse(e.getAttribute("href"));
    replace(e, newPartitionRoot);
          /*replace e by newPartitionRoot
          in the DOM tree*/
  }
```

Figure 4. Load Partition Algorithm

Note that if limited memory is available, we unload inactive partitions as needed. A partition is inactive if none of its nodes appear on the path from the root of the XML document to the currently accessed XML node. Traditional replacement techniques can be used to decide which inactive partition to unload like LRU.

We now discuss the 2LP versions of the key DOM methods that may trigger the loading of a partition: `getFirstChild()`, `getNodeName()` and `getTextContent()`. Note that the `getNextSibling()` method cannot trigger a partition loading, because even if the sibling node is an XInclude pointer, we do not have to load the partition before the user asks for the details of the returned node.

```
Node getFirstChild() {
    if this is XIncludeElement {
      loadPartition(this);
    }
    return firstChild;
  }
```

Figure 5. 2LP version of getFirstChild()

Figure 5 presents the logic to decide the loading of a partition for the `getFirstChild()` method. The original method only returns the `firstChild` member of the current object ("`this`"). In our modification, the loading is performed if the current node is an XInclude element, and it will replace the current object with the root element of the loaded partition. Thus, instead of returning directly the first child of the XInclude node, we return the first child of the root element of the partition.

Example 2.1 *Consider the partitioned XML document depicted in Figure 3 (a). Let's consider the root-to-leaf navigation pattern a→f→j→k. We start by parsing and traversing the root partition P_0. The first node-step, a, is satisfied in P_0, but to satisfy the second node-step, f, we need to follow the XInclude pointer to partition P_2. After pre-parsing P_2, we progressively parse it to reach f. We need to satisfy the last two node-steps by following the pointer to partition P_3, pre-parsing it to then progressively parse the desired nodes. In this example, we omitted the traversal of partitions P_1 and P_4.* □

Example 2.2 *Consider again the XML document in Figure 3 (a). Now consider the XPath query /Catalog/Book[@title="Storage Principles"]/Chapter. The acute reader can verify that this query requires loading all the partitions, even when we lazily process the document.* □

Note that in Example 2.2 we had to load partition P_1 just to read an attribute of its root element. To save such unnecessary partition loadings we extend the attributes of the XInclude element to contain additional information about the root element of the partition. This may save the loading of a partition when only information about its root node is required. Thus, the partition will be loaded only if the information needed by the navigation is not included in the pointer element. The data duplication to implement this idea is minimal, as shown in Section 4, since internal XML nodes are typically small.

Table 1. Inclusion Levels

Inclusion Level	***Data to Include***	***Attribute Name***
NONE	None	N/A
TAG	Tag *(Default)*	xiPartitionTag
TAG_ATR	Tag + Attributes	xiPartitionAtr
TAG_ATR_TXT	Tag + Attributes + Text	xiPartitionTxt

Table 1 summarizes the different inclusion levels regarding the data from the partition's root element to duplicate in the corresponding XInclude element. The names of the attributes used to store this data in the XInclude element are also displayed. For the TAG_ATR level, we use a single attribute whose value has the form *field1=value1&field2=value2& ...*

Example 2.2 (continued) *If we extend the XInclude elements depicted in Figure 3(b) according to Inclusion level TAG_ATR and execute the same XPath query, we will find the necessary information about the tag names and attribute values in the XInclude pointer elements. Thus, partitions P_1 and P_4 will not be loaded, since the attribute values added to the XInclude pointer can help us discriminate which "Chapter" elements satisfy the attribute condition without loading the partition.* □

The detailed code for the `getNodeName()` and `getTextContent()` methods, which varies according to the inclusion level, is available in [5] due to lack of space.

3 Partitioning the XML File

Our main goal when partitioning XML documents is to minimize the 2LP parsing time needed for navigating on the document. Other works (i. e. Natix [13, 14, 18]) have addressed the problem of partitioning the XML documents for storage purposes. Our goal here is to minimize the partitions accessed for a given request.

The key criterion to partition the original document is the number of blocks that each partition will span across the hard disk drive (i.e., the partition size). This size criterion is independent of the particular tree-structure (or schema if one exists) and the query patterns, and is shown to lead to efficient partitioning schemes (Section 4). The rationale behind this is that disk I/O performance is dictated by the average size of I/O requests when accesses are random [3]. The size criterion also allows performing a theoretical study of the optimal partition size. In the future, we plan to experiment with more complex partitioning criteria, like using different sizes for deep and shallow partitions to adapt the partition techniques to the underlying XML schema or to other physical characteristics of the document.

It must be noted that if information about the semantics and usage of the XML document is available, it can be used to further optimize the partitioning of the document. For instance, to partition a Mars document [16] we may consider the page boundaries as candidate partitioning points.

Partitioning Algorithm: The key idea of the algorithm is a bottom-up traversal of the XML tree, where nodes are added to a partition until the size threshold (in number of blocks) is reached. We show how the partition size is calculated in Section 4. Since we are using XInclude to simulate the physical pointers, we need to comply with the XInclude definition and hence provide partitions that are themselves well-formed XML documents. Thus, our partitions need to have exactly one root element and include a single subtree. This constraint leads to having a few very large partitions since every XML document typically has very few nodes with very high fan-out (e.g., *open_auctions* node in XMark [4]). Still, as shown in Section 4, this does not degrade the parsing performance as these partitions usually need to be fully navigated by XPath queries.

The partitioning algorithm, which is detailed in [5], recursively traverses T in a bottom-up fashion, calculates each subtree's size, and if this size exceeds the partitioning threshold, moves the entire subtree to a new XML document and a new XInclude pointer replaces its root node in the original XML file. Also, depending on the inclusion level flag, specific information of the partition's root node will be added to the newly created XInclude element. Figure 3 shows the resulting partitioned XML tree for the XML tree of Figure 2(b) with a threshold of ten blocks per partition. Node *b'* is the XInclude element which points to the partition rooted at node b. The same holds for nodes *f'*, *j'*, *o'*.

Partition Size: To obtain an appropriate value for the partition size, we conduct the following analysis for the root-to-leaf navigation pattern. The details of the cost model and the derivations are available in [5]. Note that performing a similar analysis for general XPath patterns is infeasible due to the complexity and variety of the navigation patterns and axes. In particular, we calculate the average access time to navigate from the root to each of the leaves of the XML document. In Section 4 we show that using the theoretically obtained partition sizes leads to good results for general XPath queries as well. When the XML document is not partitioned (and hence 2LP is not applicable), the average cost of a root-to-leaf traversal is given by the following equation:

$$Cost_{root-leaf}^{noPart} = t_{rand} + N \cdot t_{transf} \tag{1}$$

where N is the number of blocks in T, t_{rand} is the random access time needed to reach the root of the tree and t_{transf} is the time required to transfer one block of data for the specific disk drive. Note that the whole tree must be read (pre-parsed in Figure 1) to create the intermediate structure used to later progressively parse the document. No cost is assigned to the progressively parsing since the document has been already loaded in memory during pre-parsing. An equivalent cost model has been derived for the case where the tree has been segmented into equally sized partitions:

$$Cost_{root-leaf}^{Part} = \frac{\ln N}{\ln x}(t_{rand} + x \cdot t_{transf}) \tag{2}$$

where x is the number of nodes in a partition.

Taking the first derivative with respect to x of the right hand side and equating it to zero provides the optimal partition size.

4 Experiments

In this section, we evaluate our XML Partitioning and 2LP schemas. First, we evaluate the theoretical model on the partition size proposed in Section 3. Second, we measure the performance of our techniques with two navigation patterns, root-to-leaf patterns and XPath queries. The experiments were run on a 2.0GHz Pentium IV workstation with 512MB of memory running Linux. The workstation has a 20GB Maxtor D740X disk.

Evaluation of the Theoretical Model: We generated XML files of various sizes using the XMark generator [24]. We applied the partitioning algorithm to these documents, with several partition sizes (in blocks) to compare our theoretical model described in Section 3 against experimental results performing the same type of root-to-leaf navigation patterns detailed in [5]. Note that throughout the experiments the 2LP parser is used for partitioned documents and the Xerces for un-partitioned.

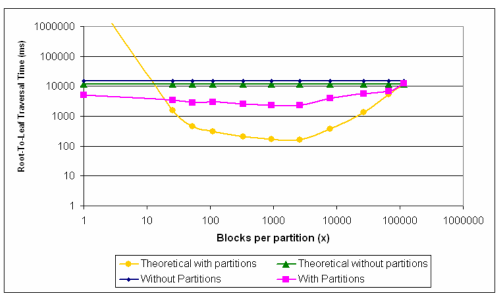


Figure 6. Average Traversal Time for Partition Sizes

Figure 6 shows the average time to traverse all the root-to-leaf paths for an XML document with XMark factor 0.5 (50MB), running on a Maxtor D740X hard drive as detailed in [5]. The theoretical curves are based on the model presented in Section 3. Notice that the scale is logarithmic and the patterns of the graphs are similar, with a slight deviation in the experimental graph. The gap between the theoretical and experimental graphs is caused because the theoretical model does not take into account the processing overhead and memory requests needed for navigating these paths, but only the I/O time involved. From the graph, we can infer the optimal size of the partition to be 2680 disk blocks, which is approximately one Megabyte. In [5] we show that the theoretical partition size is very close to the experimental one for various document sizes.

Performance Evaluation: We now present the evaluation of our approach using two types of navigation patterns, root-to-leaf traversals (also used in [6]) and XPath queries. The results for XPath carry to XQuery as well, since XQuery queries are typically evaluated by combining the results of the involved XPath queries. We adopt the "standard" XPath evaluation strategy described in [7]. As explained in Section 2, the comparisons assume that the XML document has not been already parsed before a query or navigation pattern, that is, we measure both the pre-parsing and progressive parsing times of Figure 1. We measure three time components in the total execution time:

<u>Pre-Parsing:</u> The Xerces parser uses its deferred expansion node feature by initially creating only a simple data structure that represents the document's branching and layout. This phase requires scanning the whole document to retrieve this structure. For un-partitioned documents, it means that the first time we load the file, the whole document has to be traversed and processed; for partitioned documents, every time we process a new partition, it is pre-parsed to create the logical structure in memory.

Progressive Parsing: As the navigation advances, this initial layout built in the pre-parsing phase is refined, and all the information about the nodes is added to the skeleton. This phase is performed only on the visited nodes and will have the same behavior in both un-partitioned and partitioned documents.

Inclusion: This phase is introduced by the 2LP components, and captures the time required to include and import the new partition into the working document. This component does not apply to un-partitioned documents.

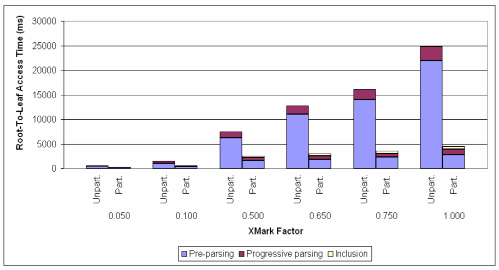


Figure 7. Root-To-Leaf Access Cost

Root-to-leaf traversal cost: Figure 7 shows the average access cost in milliseconds for the root-to-leaf access patterns, comparing the performance for different XMark factors. To compute the average time, we sampled 10% of the leaves of each document, adding each tenth leaf into the sample, and performed root-to-leaf traversals for each sampled leaf. A traversal in this case results in a sequence of parent-to-first-child and sibling-to-next-sibling operations in order to reach the desired leaf. These experiments were performed with the theoretical optimal partition size and the NONE inclusion level (the inclusion level does not impact the simple root-to-leaf traversals).

Note that in addition to the pre-parsing time, 2LP offers a significant improvement of the progressive parsing time as well. This is due to the fact the partitions are equivalent to physical pointers like node to sibling, which are not available in a traditional virtual document tree. These pointers avoid the loading and progressive parsing of unnecessary subtrees.

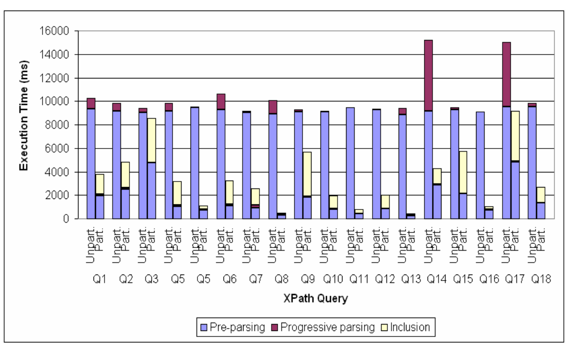


Figure 8. Average XPath Query Performance

XPath query cost: Our second experiment executes a set of XPath queries over the XML data. We selected the performance queries from queries exploit several execution constructs of the XPath syntax and several navigation axes to illustrate the behavior of our algorithms under a large range of circumstances. The complete list of queries can be found in [5]. We have included the performance queries from XPathMark [4], that is, the ones that test the execution time and not specific XPath functional aspects. We added more queries to have a larger input set in order to obtain more reliable results.

For this set of experiments, we used several XML document sizes corresponding to various XMark factors. Once again, we use the theoretical partition size for partitioning the XML documents. We used the default inclusion level (TAG) for these experiments.

Figure 8 shows the average performance of such queries for three datasets with XMark factors 0.500, 0.750 and 1.000. We see how for un-partitioned files, the pre-parsing time is always similar, since the whole document has to be processed to load the initial layout. For partitioned files, only the required partitions are processed, leading to significant reduction in the pre-parsing phase in most of the cases. We can observe that the partitioned documents perform consistently better than the un-partitioned ones. We have some cases in which the performance of the partitioned documents is almost equal to the performance of the original files. These cases, such as Q_3, Q_9, Q_{14} and Q_{15}, need to traverse most sections of the tree, requiring the inclusion of most partitions.

In the cases of Q_9, Q_{14} and Q_{17}, we load the partition rooted at *open_auctions,* which has a size of 15MB (due to the fact that each partition must be a well-formed XML document, as explained in Section 3). Pre-parsing and progressively parsing this large partition penalizes these queries and they almost match the execution time of the un-partitioned version. However, in a typical scenario, such large partitions must be completely accessed, except for the rare case when a navigation pattern specifies a child at a particular position (e.g., 1000-th child).

The inclusion time component varies correspondingly to the size of the partitions that have to be included into the working document. We see then that the inclusion component for Q_3, Q_9, Q_{14} and Q_{15} is large, but again this is caused by the large size of the *open_auctions* partition required to satisfy all these four queries. For these same queries we found large segments of time consumed by the Inclusion operation. The reason is that we rely on the `Document.importNode()`
method provided by DOM which traverses the whole imported XML tree and updates the owner document for every single node. Even when the tree is already in memory, this operation is CPU intensive, delaying the process of including the new partition.

Inclusion levels: We experimented with different inclusion levels, obtaining practically no space overhead, and observing that the TAG_ATR level is generally the best choice. We show these results in detail in [5].

5 Related Work

Noga et al. [20] introduce the idea of Lazy Parsing as presented in Section 1. The virtual document tree can potentially be stored on disk to avoid the pre-parsing stage; however, the entire virtual document tree has to still be read from disk. If a similar technique would be used with 2LP, only the needed portion of the virtual document tree will have to be read to answer the request. Schott and Noga apply these Lazy Parsing ideas to the XSL transformations [23]. Kenji and Hiroyuki [12] have also proposed a lazy XML parsing technique applied to XSLT stylesheets, constructing a pruned XML tree by statically identifying the nodes that will be referred during the transformation process.

There has been progress in developing XML pull parsers [27] for both SAX and DOM interfaces. Also, [28] presents a new API built just one level on top of the XML tokenizer, claiming to be the simplest and the most efficient engine for processing XML.

Van Lunteren et al. [15] propose a programmable state machine technique that provides high performance in combination with low storage requirements and fast incremental updates. A related technique has been proposed by Green et al. [9] to lazily convert an XPath query into a Deterministic Finite Automata (DFA), after which they submit the XML document to the DFA in order to solve the query. They propose a lazy construction opposed to an eager creation, since constructing the DFA with the latter technique can lead to an exponential growth in the size of the DFA. Kiselyov [11] presents techniques to use functional programming to construct better XML Parsers.

6 Conclusions

Lazy XML parsing is a significant improvement to the performance of XML parsing but to achieve higher levels of performance there is a need to further optimize the pre-parsing phase during which the whole document is read, as well as the progressive parsing phase during which a query is processed. In this paper, we address this problem by enabling laziness in the pre-parsing phase and allowing skipping the processing of entire (unwanted) subtrees of the document during the progressive parsing phase. To do so, we have proposed a mechanism to add physical pointers in an XML document by partitioning the original document and linking the partitions with XInclude pointers. We have also proposed 2LP, an efficient parsing algorithm for such documents, that implements pre-parsing laziness. These techniques significantly improve the performance of the XML parsing process and can play a significant role in accelerating the wide adoption of XML.

References

1. S. Abramsky. The Lazy Lambda Calculus. In D. Turner (ed.), Research Topics in Functional Programming. AddisonWesley, 1990.
2. Document Object Model (DOM), http://www.w3.org/DOM/, 2006.
3. Z. Dimitrijevic and R. Rangaswami. Quality of Service Support for Real-time Storage Systems, In IPSI, 2003.
4. M. Franceschet. XPathMark: An XPath Benchmark for the XMark Generated Data. In XSym, 2005.
5. F. Farfán, V. Hristidis and R. Rangaswami. Beyond Lazy XML Parsing Extended Version. http://www.cs.fiu.edu/SSS/beyondLazyExt.pdf. 2007.
6. Joseph Gil and Alon Itai. How to pack trees. Journal of Algorithms, 32(2):108–132, 1999.
7. G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In VLDB, 2002.
8. Geography Markup Language - http://opengis.net/gml/, 2006.
9. T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In ICDT, 2003.
10. Health Level Seven XML - http://www.hl7.org/special/Committees/xml/xml.htm. 2006.
11. O. Kiselyov. A Better XML Parser Through Functional Programming. In Lecture Notes in Computer Science, Vol. 2257. Springer-Verlag, Berlin Heidelberg New York. Pages 209-224. 2002.
12. M. Kenji and S. Hiroyuki. Static Optimization of XSLT Stylesheets: Template Instantiation Optimization and Lazy XML Parsing. In DocEng, 2005.
13. C. C. Kanne and G. Moerkoette. 1999. Efficient storage of XML data. In ICDE, 1998.
14. C. C. Kanne and G. Moerkoette. A Linear-Time Algorithm for Optimal Tree Sibling Partitioning and its Application to XML Data Stores. In VLDB, 2006.
15. J. van Lunteren, T. Engbersen, J. Bostian, B. Carey and C. Larsson. XML Accelerator Engine. First International Workshop on High Performance XML Processing, 2004.
16. Mars Reference. Version 0.7. Adobe Systems Inc., http://download.macromedoa.com/pub/labs/mars/mars_reference.pdf
17. Medical Markup Language, http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&list_uids=10984873&dopt=Abstract. 2006.
18. Natix. http://www.dataexmachina.de/. 2006.
19. M. Nicola and J. John. XML Parsing: a Threat to Database Performance. In CIKM, 2003.
20. M. Noga, S. Schott and W. Löwe. Lazy XML Processing. In ACM DocEng, 2002.
21. OpenDocument Specification v1.0 – http://www.oasis-open.org/committees/download.php/12572/OpenDocument-v1.0-os.pdf. 2006.
22. Simple API for XML (SAX), http://www.saxproject.org/, 2006.
23. S. Schott and M. Noga. Lazy XSL Transformations. In ACM DocEng, 2003.
24. A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In VLDB, 2002.
25. Apache Xerces2 Java Parser. Apache XML Project, http://xml.apache.org/xerces-j/. 2006.
26. XML Inclusion - http://www.w3.org/TR/xinclude/. 2006.
27. XML Pull Parsing. http://www.xmlpull.org/index.shtml. 2006.
28. XML Pull Parser. http://www.extreme.indiana.edu/xgws/xsoap/xpp/. 2006.
29. XML Pointer Language Version 1.0 - http://www.w3.org/TR/WD-xptr. 2006.