2LP: A Double-Lazy XML Parser

Fernando Farfán [a], Vagelis Hristidis [a], Raju Rangaswami [a],

[a] *Florida International University, School of Computing and Information Sciences, Miami, Florida 33199, USA*

Abstract

XML is acknowledged as the most effective format for data encoding and exchange over domains ranging from the World Wide Web to desktop applications. However, large-scale adoption into actual system implementations is being slowed down due to the inefficiency of its document-parsing methods. The recent development of lazy parsing techniques is a major step towards improving this situation, but lazy parsers still have a key drawback –they must load the entire XML document in order to extract the overall document structure before document parsing can be performed. We have developed a framework for efficient parsing based on the idea of placing internal physical pointers within the XML document that allow the navigation process to skip large portions of the document during parsing. We show how to generate such internal pointers in a way that optimizes parsing using constructs supported by the current W3C XML standard. A double-Lazy Parser (2LP) exploits these internal pointers to efficiently parse the document. The usage of supported W3C constructs to create internal pointers allows 2LP to be backward compatible – i.e., the pointer-augmented documents can be parsed by current XML parsers. We also implemented a mechanism to efficiently parse large documents with limited main memory, thereby overcoming a major limitation in current solutions. We study our pointer generation and parsing algorithms both theoretically and experimentally, and show that they perform considerably better than existing approaches.

Key words: XML, DOM, Trees, Document management, Optimization.

1 Introduction

XML has become the de facto standard format for data representation and exchange in domains ranging from the Web to desktop applications. Exam-

Email addresses: ffarfan@cis.fiu.edu (Fernando Farfán), vagelis@cis.fiu.edu (Vagelis Hristidis), raju@cis.fiu.edu (Raju Rangaswami).

ples of XML-based document types include Geographic Information Systems Markup Language (GML) [5], Medical Markup Language (MML) [7], HL7 [6], and Open Document Format (ODF) [8,9]. This widespread use of XML requires efficient parsing techniques. The importance of efficient XML parsing methods was underscored by Nicola and John [41]; they showed that the parsing process is processor and memory consuming, particularly needing main memory as much as five times the size of the original document.

There are two de facto XML parsing APIs, DOM [2] and SAX [10]. 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()`, `getNextSibling()`, and so on. 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., [1]) 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 [18]. The architecture shown in Figure 1, based on the terminology of [42], consists of two stages. First, a preprocessing stage extracts a virtual document tree, which stores only node types, hierarchical structure information and references to the data associated with each node. After this structure is obtained, a progressive parsing engine refines this virtual tree on demand, which grows as needed, expanding the original virtual nodes into complete nodes with values, attributes, etc.

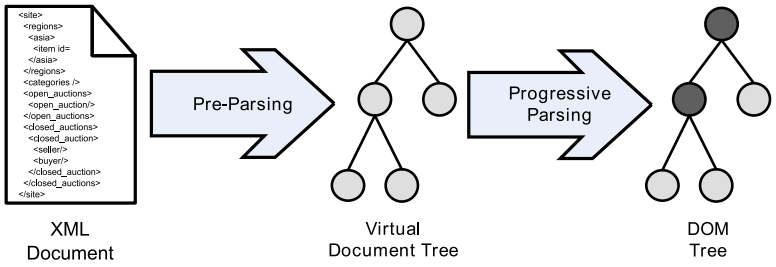


Fig. 1. Lazy XML Parser Architecture. A pre-parsing phase extracts a virtual document tree and a progressive parsing engine refines this virtual tree on demand.

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. We propose a method to (a) insert such internal physical pointers in the document, and (b) exploit them to optimize the parsing method and specially the pre-parsing stage. In particular, our approach is called *double-Lazy Parsing (2LP)* because both stages in Figure 1 are lazy, in contrast to previous work where only the second stage is lazy. The pre-parsing phase will lazily process only the subtrees of the XML document that are necessary to satisfy the navigation request.

We address two key issues in inserting such physical pointers. First, we need to decide how we can implement the pointers given the current W3C XML standard specification [20]. Second, we need to decide where to add the pointers, considering the incurred overhead adding pointers on every node can cause the size of the file to double. Also, following a pointer would typically require a random disk access, and hence excessive use of such pointers must be avoided.

Regarding the first issue, we emulate physical pointers, by partitioning the original XML document into several fragments (subtrees) which are then interlinked using the XML Inclusion [12] feature. A drawback of this approach is that the XML document is split into a set of smaller XML documents/files [1]. However, we shall argue and demonstrate in the rest of this paper that the performance gains far outweigh this drawback. Regarding the second issue, we investigate in detail the tradeoff decisions to be made with respect to fragment size, and propose an optimal configuration that can be applied in general cases.

We also propose a method to manage the parsing of large XML documents under limited main memory configurations. This approach allows 2LP to scale to large XML documents, even when the total size of the document surpasses the available amount main memory. This is not possible with current parsers, which report "Out of memory" exceptions under such condition.

This paper makes the following contributions:

(1) We develop a framework to allow efficient XML parsing, which improves the pre-parsing time as well as the memory requirements of parsing. Our framework is based on the idea of placing internal physical pointers within the document. Such pointers are currently realized using the XML Inclusion feature.

[1] Unfortunately, the XML standard does not support an alternative physical pointer construct (XPointer [13] is logical and not physical) due to the complication this would incur during cross-platform document exchange. If such a feature becomes available in the future, it could be used instead of the described partitioning approach.

(2) We present algorithms to perform double-Lazy XML Parsing (2LP) for DOM-like navigation, given internal physical pointers. We have implemented 2LP as a backward compatible modification of the Apache Xerces2 Java Parser [1].
(3) We present algorithms to add internal physical pointer to the XML document by partitioning it into subtrees given an optimal partition size. We show how the theoretically optimal partition size can be computed assuming knowledge of the navigation patterns on complete XML trees and knowing the hard disk characteristics.
(4) We efficiently manage the main memory consumption of our XML parser, making it possible to parse and navigate large documents under conditions in which other approaches fail.
(5) 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.

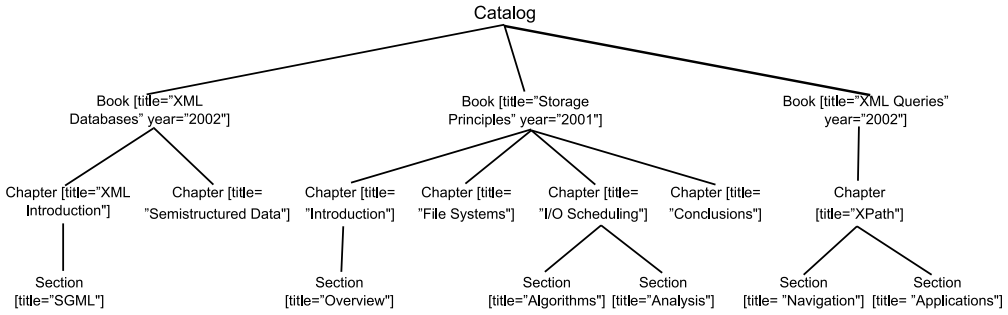
The rest of the paper is organized as follows: Section 2 presents the system framework and the overview of our approach. We describe our double-Lazy parsing techniques in Section 3. Section 4 presents techniques for partitioning the original document into smaller subtrees. An approach to parse using a limited amount of main memory is presented in Section 5. The implementation of all these techniques is discussed in Section 6. Our experiments are discussed in Section 7. We present related work in Section 9. Finally, Section 8 discusses our conclusions.

2 System Framework and Overview of Approach

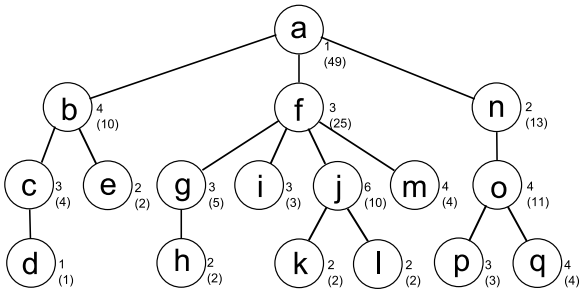
2.1 Data and Query Models

XML data: We view an XML document as a labeled tree T, where each node v has a label $\lambda(v)$, which is a tag name for non-leaf nodes and a value for leaf nodes. Non-leaf nodes v have an optional set $A(v)$ of attributes, where each attribute a $A(v)$ has a name and a value. For simplicity in the presentation we assume that there are no ID-IDREF edges (which would make the tree a graph). However, our framework can support ID-IDREF edges by including the partition id, in addition to the attribute id, in the IDREF attributes. Figures 2(a) and 2(b) show a sample XML document and its corresponding tree representation respectively. We annotate each node with its size and the size of its subtree (in parenthesis) in Figure 2(b). To simplify the discussions in the rest of the paper, we assume that these sizes are in numbers of disk blocks. Similarly, the number in the parenthesis represents the size (in blocks

again) of the subtree rooted at the node.



(a) XML Document



(b) XML Tree

Fig. 2. Sample XML Document and its corresponding tree. We annotate each node with its size and the size of its subtree (in parenthesis) in Figure 2(b).

We clarify that this work is not aiming at improving the performance of XML database systems [4,11,17,34,40], where indexes [27,31] and other optimizations are possible, but at improving the efficiency of using XML as a format to store documents for general applications as motivated in Section 1.

XML navigation patterns: We consider two types of XML navigation patterns in our experiments. The first type is a simple *root-to-leaf traversal*, in which a path is traversed from the root of the XML document to any of its leaves. We use this simple yet common and useful pattern to model the theoretical behavior of our approach.

Second, we use XPath queries. We use XPath and not XQuery because our work tackles the problem of efficient parsing for the purpose of efficiently navigating the XML data, which is XPath's role. However, our 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 [28] shown in Figure 3. Intuitively, this algorithm processes an XPath query Q in a depth-first manner on the XML document, one step of Q ($Q.first$) at a time, and stores the intermediate results in a set S.

```
procedure processLocationStep(n0, Q)
    /* Node n0 is the context node;
     * Query Q is a list of location steps */
1   NodeSet S := apply Q.first to node n0;
2   if(Q.tail is not empty) then
3      for each node n in S do
4         processLocationStep(n, Q.tail)
```

Fig. 3. Standard XPath evaluation strategy according to [28].

2.2 Disk Drive Modeling

We base our disk drive modeling on the work of [43]. In their model, *seek*, *rotation*, and *transfer times*, combine the following features:

- A *seek time* that is linear with the distance, using the single-cylinder and full-stroke seek times published in the disk drive specification.
- No head-settle effects or head-switching costs.
- A *rotational delay* drawn from a uniform distribution over the interval [0, *rotation time*).
- A fixed controller overhead.
- A *transfer time* linear with the length of the request [43].

Given these characteristics, we utilize their models to obtain the transfer time and random access time for the set of hard disk drives that use for our theoretical model and experimental section. In Appendix A we present an extended explanation on how the model was used, as well as the summarized data sheet for the featured disk drives.

2.3 Overview of Approach

Our approach for parsing XML documents consists of two stages. First, the document is partitioned into a set of smaller XML files, which are then interlinked using XInclude [12] pointers. The optimal size of a partition is computed using a formula which considers the random versus sequential access characteristics of a hard disk. The second stage involves the parsing of a partitioned document. The key goal is to read a minimal set of partitions in order to perform the sequence of navigation commands. 2LP loads (pre-parses using the terminology of Figure 1) the partitions in a lazy manner, that is, only when they are absolutely necessary for the navigation sequence. 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 T. Then $D(T)$ is augmented with the DOM trees $D(P_i)$ of the loaded partitions P_i.

Further, to control memory usage, our approach also performs lazy unloading of inactive partitions (discussed in Section 3) if the total amount of main memory used by the DOM tree exceeds a threshold. Thus, 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 implement the proposed technique for optimizing memory usage, but to a smaller extent since the virtual document tree must be stored in memory at all times.

3 2LP on Partitioned XML Documents

Let T be the original XML document, and $P_0, \ldots P_n$ be the partitions to which T was split during the partitioning stage, explained in Section 4. P_0 is the root partition, since it contains the root element of T. Figure 4 shows an example of a partitioned XML tree. All the partitions are connected by XInclude elements, containing the Uniform Resource Identifier (URI) to the partition file. The XInclude elements are represented in the figure by nodes b', f' and j', as explained in Section 4.1.

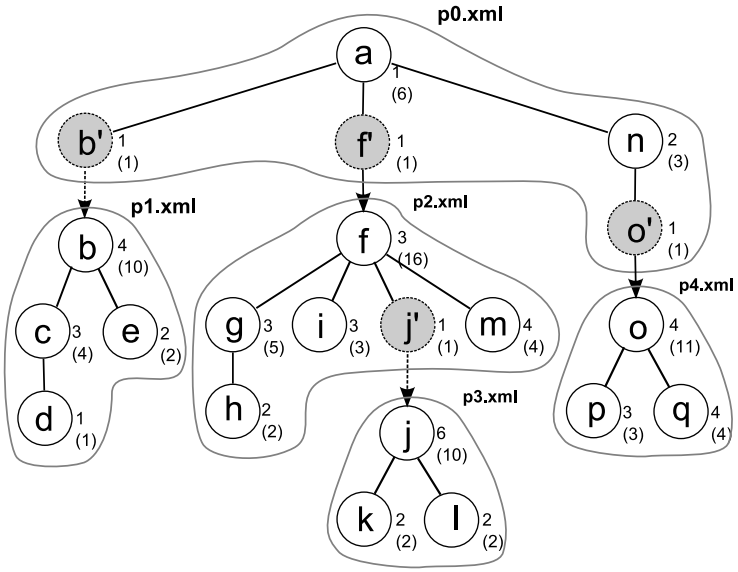


Fig. 4. Partitioned XML Tree after partitioning the tree in Figure 2(b).

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 directly access node n from node f'.

The XML representation of two of the partitions in Figure 4 is shown in Figure 5. Partition P_0 corresponds to the root partition since it contains the root of the original XML document. 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 added to the tree upon partitioning will hold the reference to the root of the partition's subtree. We explain this aspect in detail in Section 4.

p0.xml

```
<Catalog>
   <xi:include href="p1.xml"
       xmlns:xi="http://www.w3.org/2001/XInclude" />
   <xi:include href="p2.xml"
       xmlns:xi="http://www.w3.org/2001/XInclude" />
   <Book title="XML Queries" year="2002">
      <xi:include href="p4.xml"
          xmlns:xi="http://www.w3.org/2001/XInclude" />
   </Book>
</Catalog>
```

p1.xml

```
<Book title="XML Databases" year="2002">
   <Chapter title="XML Introduction">
      <Section title="SGML" />
   </Chapter>
   <Chapter title="Semistructured Data" />
</Book>
```

Fig. 5. XML Documents after partitioning.

Figure 6 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.

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

Fig. 6. Load Partition algorithm.

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.

Similarly, we also need to decide which partitions to unload and when to do so in order to accommodate new partitions that need to be loaded, given that the available system memory is limited. We address unloading of partitions in detail in Section 5.

We now present the 2LP versions of the key DOM methods that may trigger the loading of a partition: `getFirstChild()`, `getTextContent()` and `getNodeName()`. 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 (e.g., using `getNodeName()` shown below).

Figure 7 presents the `getFirstChild()` method with the logic to decide whether a partition has to be loaded. 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 assign the root element of the loaded partition to the firstChild member variable. Thus, instead of returning directly the first child of the XInclude node, we return the first child of the root element of the partition.

```
Node getFirstChild() {
1   if this is XIncludeElement {
2       loadPartition(this);
3   }
4   return firstChild;
 }
```

Fig. 7. Modified `getFirstChild()` method to handle the lazy loading of partitions.

Example 3.1 *Consider the partitioned XML document depicted in Figure 4. Let's also consider the root-to-leaf navigation pattern $a \rightarrow f \rightarrow j \rightarrow k$. We start by parsing and traversing the root partition, labeled P_0. The first node-step, a, is satisfied in partition P_0, but to satisfy the second node-step, f, we need to follow the XInclude pointer to partition P_2, while completely skipping the processing of P_1. After pre-parsing partition 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 3.2 *Let's consider the XML document in Figure 5 and the XPath query* `/Catalog/Book[@title=''Storage Principles'']/Chapter`.
The careful reader can verify that this query requires loading all the partitions, even when we lazily process the document. □

Note that in Example 3.2 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 7.2, since internal XML nodes typically are very small.

Table 1 summarizes the different *inclusion levels* based on the data from the partition's root element that is duplicated 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 will resemble a query string (as used in World Wide Web forms) of the form $field1 = value1\&field2 = value2\&field3 = value3\ldots$ [19].

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`

Example 3.2 (continued) *If we extend the XInclude elements depicted in Figure 5 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 processed at all, since the attribute values added to the XInclude pointer can help us discriminate which "Chapter" elements satisfy the attribute condition without loading the partition.* □

In addition to the `getFirstChild()` method presented above, which is unaffected by the inclusion level, we now show how other key navigation methods of DOM need to be modified for the 2LP. Figure 8 presents two navigation routines that have been modified to allow the double-lazy processing of XML partitions with different inclusion levels. Similar to the `getFirstChild()` method, these two methods return (originally) just the corresponding member variable of the object. By modifying them, the methods will lazily include the corresponding partition if and only if this is needed to satisfy the navigation pattern and if the desired information is not included in the XInclude pointer element. If the inclusion is performed, the root element of the partition is assigned to current object ("`this`") and its member variables (name and text for `getNodeName()` and `getTextContent()` respectively) are returned. Similar modifications are performed for the other DOM methods that can potentially trigger the loading of a partition.

4 Partitioning the XML File

Our main goal when partitioning XML documents is to minimize the 2LP parsing time needed for navigating the document.

In what follows, we first describe (in Section 4.1) how to partition an XML

```
String getNodeName() {
1   if this is XIncludeElement {
2       if inclusionLevel != NONE then { /** tag information present
            in XInclude element **/
3           name = this.getAttribute("xiPartitionTag");
            /** The xiPartitionTag attribute inside the XInclude
            element stores the tag name of the root element of the
            partition **/
4       } else {
5           loadPartition(this); /** ``this'' is now pointing at
            the root element of the loaded partition and the
            ``name'' variable is updated **/
6       }
7   }
8   return name;
 }

String getTextContent() {
1   if this is XIncludeElement then {
2       if inclusionLevel = TAG_ATR_TXT then {
3           text = this.getAttribute("xiPartitionTxt");
4       } else {
5           loadPartition(this); /** ``this'' is now pointing at the
            root element of the loaded partition and the ``text''
            variable is updated **/
7       }
8   }
9   return text;
 }
```

Fig. 8. Key modified Document Object Model Navigation Methods.

document by selecting subtrees of an optimal size. Then in Section 4.2 we make a theoretical analysis to obtain the optimal partition or subtree size, based in simplified navigation patterns.

4.1 Partitioning Algorithm

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 7). The rationale behind this is that disk I/O performance is dictated by the average size of I/O requests when accesses are random [24].

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 optimal partition size is calculated in Section 4.2.

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. This means that our partitions need to have exactly one root element. Thus, the partitioning algorithm must include entire subtrees when creating a new partition. This constraint leads to having a few very large partitions since every XML document typically has very few nodes with very high fanout (e.g., open_auctions node in XMark [26]). However, as we shall show in Section 7, this does not degrade the parsing performance since these partitions typically need to be completely navigated by XPath queries.

Figure 9 describes the basic tree partitioning algorithm. The execution of `partitionTree(T.root, threshold)` will recursively traverse T in a bottom-up fashion, calculate the size of each subtree, and if this size exceeds the threshold, then the `createPartition()` method is called for this subtree. The `createPartition()` method will move the entire subtree to a new XML document and a new XInclude element will replace its root node in the original XML file to reference the new partitioned subtree. Also, depending on the inclusion level flag, specific information of the partition's root element will be added to the newly created XInclude element.

Figure 5 shows the resulting partitioned XML tree for the XML tree of Figure 2(b) with a threshold of 10 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'.

Example 4.1 *Consider the XML document in Figure 2(b). When we execute the method* `partitionTree(a, 10)`*, the depth-first traversal of the tree rooted at a begins. The traversal will descend until it reaches the leftmost branch, and from there it will begin the bottom-up search for the subtree whose size in blocks is larger or equal to the specified threshold. Hence, we first create a new partition for the subtree rooted at node b, replacing this node with an XInclude pointer to the newly created partition. We assume in this case that we are using the default inclusion level (NONE), and thus an extra block is used by the pointer to maintain the data. We continue the navigation and create another partition with the subtree rooted at node j, repeating the same steps; we further create the new partitions rooted at nodes f and o.* □

```
int partitionTree(node n, int threshold) {
1   size = getSize(n); /** Returns the size in bytes of the node n,
        including attribute names and values and text section **/
2   for each child c of n {
3       size = size + partitionTree(c);
4   }
5   if size >= threshold and not isRoot(n) {
6       createPartition(n);
7       size = getSize(n); /*Recalculates the size after the
            partition is created*/
8   }
9   return size;
 }

createPartition(node n) {
1   x = create new XML file;
2   addXIncludePtr(n, x); /** Replace the subtree rooted at n in the
        current XML document by an XInclude element pointing at
        file x **/
3   moveSubtree(n, x); /** Move the subtree rooted at n to file x **/
 }
```

Fig. 9. Partitioning Algorithm.

4.2 Estimating the Optimal Partition Size

To obtain an appropriate value for the partition size, we conduct the following analysis for the root-to-leaf navigation pattern described in Section 2.1. In particular, we calculate the average access time to navigate from the root to each of the leaves of the XML document. While performing a similar analysis for general XPath patterns is infeasible due to the complexity and variety of the navigation patterns, we show, in Section 7, that using the theoretically obtained partition sizes leads to good results for general XPath queries as well.

We assume, for sake of simplicity, that our tree is complete and each node of T occupies a single disk block [2]. Therefore, the XML tree T, which has N nodes and degree d, has height $h = \log_d N$. As we shall see in the evaluation section, the simplifying assumptions used in our theoretical model do not significantly impact the key results; the theoretically optimal is found to be very close to the experimentally computed optimal size.

Cost with no partitions: When the XML document is not partitioned (and hence 2LP is not applicable), the average cost of a root-to-leaf traversal is given

[2] Later on, we shall show that in spite of these simplifying assumptions, the experimentally obtained optimal partition sizes closely match our theoretical estimates.

by the following equation. Note that for simplicity we assume the document is parsed from scratch every time a navigation pattern occurs.

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

where 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 progressive parsing phase since the document has been already loaded in memory during pre-parsing.

Cost with partitions: Let us assume that the tree will be segmented into equally sized partitions, and we can describe each partition as having:

x: Number of nodes in partition
$h' = log_d x$: Height of the partition

In this case, the average cost for a root-to-leaf traversal is given by the following equation:

$$Cost_{root-leaf}^{Part} = (\ \#\ partitions\ accessed) \times (t_{rand} + x \cdot t_{transf})$$

where $t_{rand} x \cdot t_{transf}$ is the cost to pre-parse and load a partition. The number of partitions along a root-to-leaf traversal is h/h'. Hence we have the following equation:

$$Cost_{root-leaf}^{Part} = \frac{h}{h'}(t_{rand} + x \cdot t_{transf})$$

Observe that the ratio of heights can be simplified using logarithmic properties, and is independent of d. As a result, we obtain:

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

Based on (2), we model the optimal cost of the partition size for four different hard disk drives, described in Table 2. A detailed description of our hard disk drive model and how we calculate the data transfer and random access times is included in Appendix A.

Figure 10 presents the times $Cost_{root-leaf}^{Part}$ for the four different disk drives presented in Table 2 for varying partition sizes x. The optimal partition size is the value of x that minimizes the time.

The un-partitioned cost $Cost_{root-leaf}^{noPart}$ is equal to the time for the maximum partition size, where the whole document fits in a single partition.

Table 2
Disk Drive Characteristics

Disk Model	Size (GB)	t_{transf} (ms)	t_{rand} (ms)
Maxtor D740X	20.0	0.009446	5.441876
Fireball Plus	27.3	0.007688	5.359013
Cheetah 15K.4	36.7	0.002560	2.359615
Hitachi Ultrastar	73.4	0.003810	3.520530

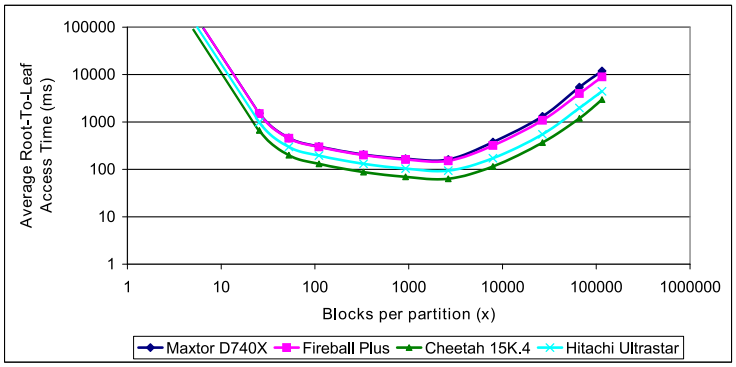


Fig. 10. Effect of varying the partition sizes on the average root-to-leaf navigation access time.

5 Management of Limited Main Memory

As mentioned earlier, the DOM representation of an XML document can span up to five times its size in main memory. This fact combined to the increasing size of XML documents causes current XML parsers to often fail with an "Out of Memory" exception.

We have created and implemented a mechanism to *unload inactive partitions* from the overall DOM tree. A partition P is considered as inactive if the path from the root of the DOM tree to the point of the current navigation sequence does not include any element from P.

To achieve the unloading of inactive partitions, we add a data structure that stores the information about the root element of the partition, the path of the partition document in the file system as well as a pointer to such root element. Every time a partition is loaded, all the metadata and the pointer are stored for further analysis. Also, after each partition is loaded, the system checks for the size of the overall DOM tree to decide whether one or more partitions have to be unloaded.

Figure 11 presents a modified version of the `loadPartition()` method presented in Figure 6. This new version adds the logic for unloading partitions to restrict the total amount of main memory used by the overall DOM tree to a fixed threshold. Every time a new partition is loaded, the method checks for the overall memory utilization and unload suitable partitions until the mem-

```
procedure loadPartition(XIncludeElement e) {
1   newPartitionRoot = preParse(e.getAttribute("href"));
2   replace(e, newPartitionRoot); /** replace e by newPartitionRoot
        in the DOM tree **/
3   registerPartition(this, e);
4   while(size(T) > memory_threshold) {
5       unloadPartition(getPartitionToUnload(T));
6   }
 }
```

Fig. 11. Modified Load partition Algorithm with Partition Unloading mechanism.

ory usage is below the threshold. Three auxiliary methods are added to handle the logic:

registerPartition(): This method receives as parameters the current element at which the partition has been added as well as the metadata of the partition. It stores the filename of the partition document in the file system and all the necessary information to recreate the XInclude pointer when the partition has to be unloaded.

getPartitionToUnload(): This method analyzes the information stored by the `registerPartition()` method and decides which partition has to be unloaded. We implemented two variants of this method to implement the *First-in, First-out (FIFO)* and *Least Recently Used (LRU)* [46] strategies, as used in the context of virtual memory page replacement. The *FIFO* strategy picks the oldest partition, taking the one at the head of the partition queue. When a partition is loaded, we insert it at the tail of the queue. Notice that the root partition, P_0, will never be unloaded, since it contains the root of the original XML document and we need to maintain that information accessible at all times. The *LRU* strategy discards the least recently used partitions; it requires keeping track of what was used when, which is more expensive than *FIFO*.

unloadPartition(): Once the `getPartitionToUnload()` method selects a partition, this method removes the underlying subtree from the overall DOM tree and reconstructs the XInclude pointer using the metadata stored by the `registerPartition()` method.

6 System Implementation

In this section, we describe the architecture and implementation of the two key components of our system: the XML Partitioner and the 2LP parser.

The system architecture is shown in Figure 12. The XML Partitioner takes a source XML document and partitions it based on a threshold determined

using the model presented in the previous section. The 2LP Parser can also parse un-partitioned XML documents.

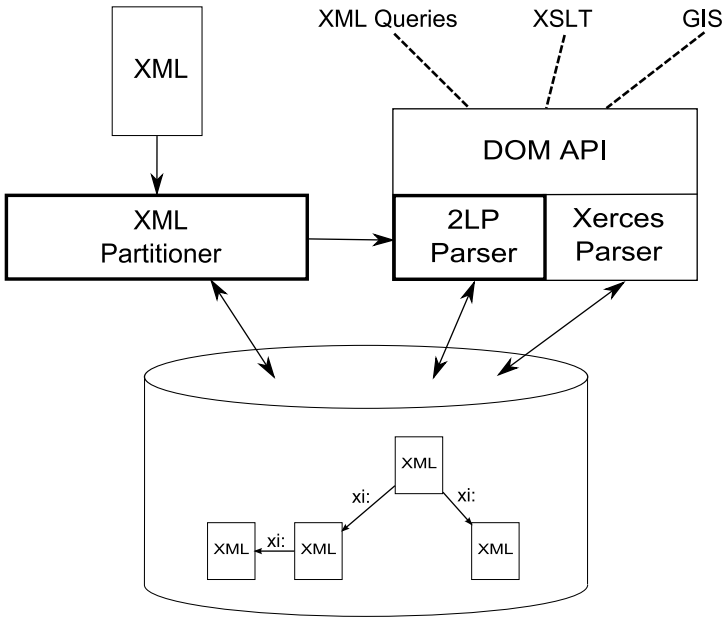


Fig. 12. XML Partitioning and 2LP Architecture.

The 2LP parser was implemented by modifying the Xerces2 Java Parser, allowing it to handle the XInclude-defined partitions, but also preserving its backward compatibility. Figure 13 shows a simplified class diagram in Unified Modeling Language (UML) notation [21,30], for the classes involved in our modification. The top layer is the W3C DOM Interface, followed by the Xerces2 Java Parser which is the implementation of such interface. The shadowed classes are the ones modified from the open-source package. The bottom layer is our own package, which encapsulates the modifications required to handle the partitioning and inclusion mechanisms.

Below, we describe the key ideas behind the modified and newly added classes in the implementation.

ElementImpl: This class was modified to handle inclusion behavior for two methods, `getNodeName()` and `getAttributes()`. Depending on the inclusion level, these methods may answer a query with local information or require an inclusion to import a new partition and answer the query.

PartitionMgr: The `PartitionMgr` class is attached to the `CoreDocumentImpl` class in the Xerces package, to manage the orchestration of traversal and inclusion. Every time a new partition is required, the `XIncludeHandler` will process the specified URI and a new `Partition` object will be created. It also manages the unloading mechanism.

XIncludeHandler: This class handles directly the inclusion operations when

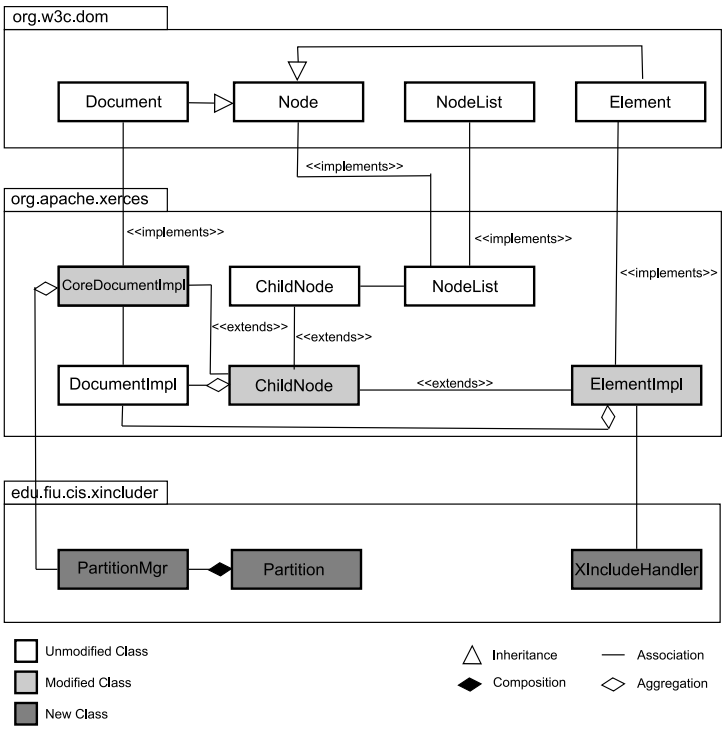


Fig. 13. XML Partitioning and 2LP Class Diagram.

invoked from the `ParentNode` and `ElementImpl` objects in the Xerces package. This class works as a replacement to the default XInclude processor provided by the Xerces parser. In order to achieve this, we turn off the XInclude feature, and let our package handle these pointers.

Partition: This class is an abstraction to manage each partition as it is processed by the `XIncludeHandler` class. Notice that all the user-level interaction is still performed via the DOM Interface, guaranteeing the backward compatibility desired as a design goal. We have made our XML Inclusion feature backward compatible, so another XML document that has XInclude pointers in it will be treated in the same way by our double lazy parser, and any partitioned document joined by XInclude pointers will be handled by any Xerces parser in a correct way.

7 Experiments

In this section, we evaluate our XML Partitioning and 2LP schemas. First, we experiment with optimal size of partitions based on the theoretical model proposed in Section 4.2. Second, we measure the performance of our techniques with two navigation patterns, root-to-leaf patterns and XPath queries, as presented in Section 2.1. Third, we evaluate the impact of our memory management optimization by unloading unnecessary partitions, as presented

in Section 5.

Our framework was developed in Java using JDK 5.0. We modified the Xerces2 Java Parser 2.9.1 [1]. The experiments were performed on a 2.0GHz Pentium IV workstation with 512MB of memory running Linux. The workstation has a 20GB Maxtor D740X disk.

7.1 Evaluation of the Theoretical Model

We generated XML files of various sizes using the XMark generator [44]. We applied the partitioning algorithm to these documents, with several partition sizes (in blocks) to compare our theoretical model described in Section 4.2 against experimental results performing the same type of root-to-leaf navigation patterns described in Section 2. Note that throughout the experiments the 2LP parser is used for partitioned documents and the Xerces for unpartitioned.

Figure 14 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 described in Section 4.2. The theoretical curves are based on the model presented in Section 4.2. Notice that the scale is logarithmic and the patterns of the graphs are similar, with a slight deviation in the experimental graph. We believe that the gap between the theoretical and experimental graphs is caused because the theoretical model does not take into account the processing time needed to navigate these paths and the effect of paging due to the limited amount of memory, but only the primary I/O time involved in reading the partition for the XML file. From the graph, we can infer the optimal size of the partition to be 2680 disk blocks, which is approximately one Megabyte.

Next we compare the optimal partition size (obtained experimentally) for various document sizes (by varying the XMark factor) with the theoretical optimum. Figure 15 shows these results for the same hard drive, where again the theoretical and experimental values are close. For the first two XMark factors, the experimental optimal values are considerably smaller than the theoretical prediction. This is due to the fact that for the case of small files, having smaller partitions will benefit the performance of the navigation patterns, since it is more likely that the partitions (stored in the same directory) are contiguously placed on disk. The file system can efficiently (sequentially) retrieve all the partitions from the disk.

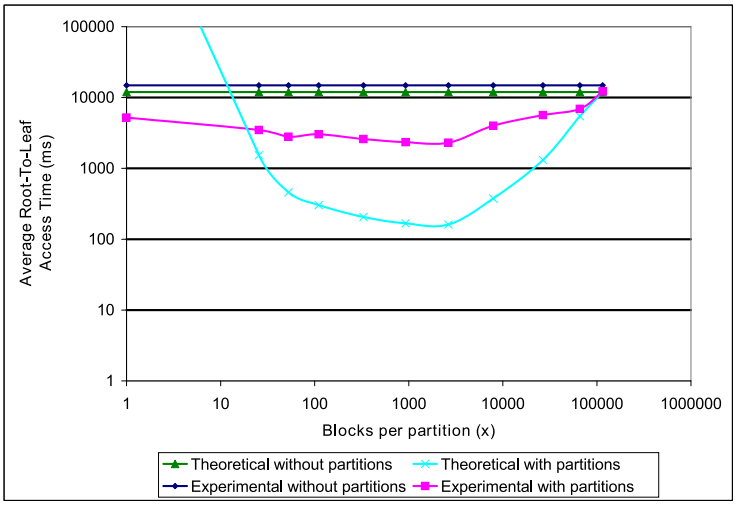


Fig. 14. Average Traversal Time for Partition Sizes.

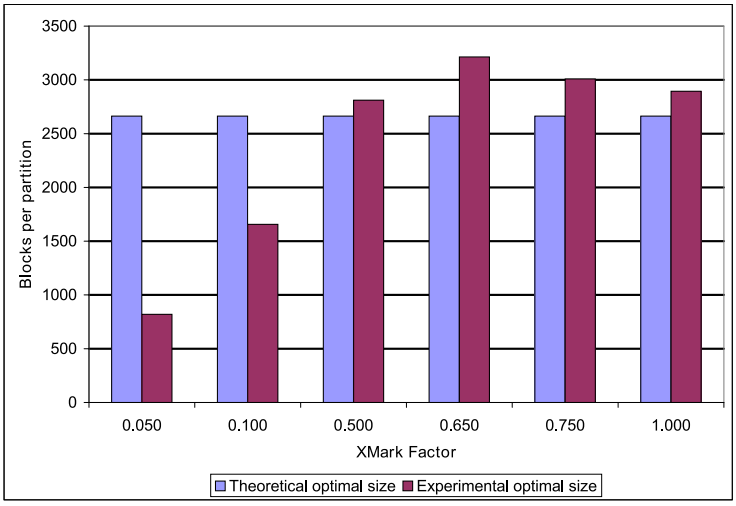


Fig. 15. Optimal Size of Partitions.

7.2 Performance Evaluation

We now present the evaluation of our approach using two types of navigation patterns, root-to-leaf traversals and XPath queries. As explained in Section 4.2, 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:

Pre-Parsing: 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.

Root-to-leaf traversal cost: Figure 16 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).

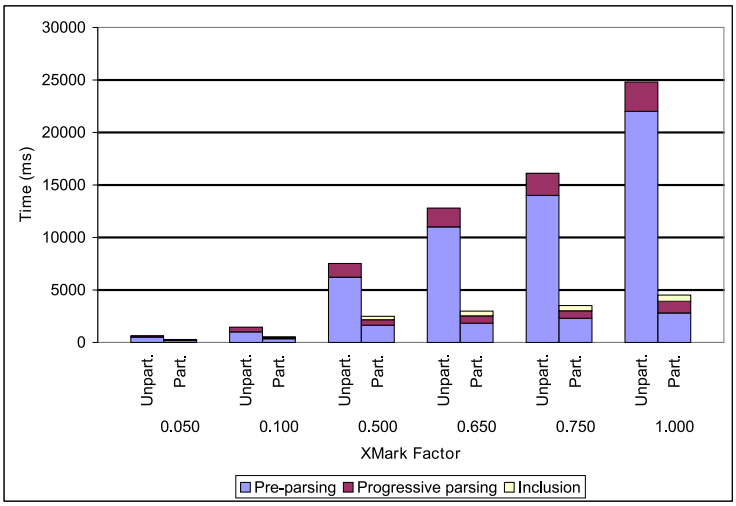


Fig. 16. Root-To-Leaf Access Cost.

XPath query cost: Our second experiment executes a set of XPath queries over the XML data. The queries are shown in Table 3. We have included the performance queries from XPathMark [26], that is, the ones that test the execution time and not specific XPath functional aspects. We added more queries to have more reliable results.

For this set of experiments, we used several XML document sizes corresponding to various XMark factors. Once again, we use the theoretically optimal

Table 3
XPath Queries

#	Query
Q_1	$/site/closed\_auctions/closed\_auction/annotation/description/parlist/listitem/text/keyword$
Q_2	$/site/people/person/watches$
Q_3	$/site/open\_auctions/open\_auction/annotation/description/text/keyword$
Q_4	$/site/people/person/address/country$
Q_5	$/site/regions/australia/item/description/tex/emph$
Q_6	$/site/people/person/*/business$
Q_7	$/site/closed\_auctions/closed\_auction/*/description$
Q_8	$/site/regions/*/item/description/text$
Q_9	$/site/open\_auctions/openauction$
Q_{10}	$/site/closed\_auctions$
Q_{11}	$/site/regions/australia$
Q_{12}	$/site/closed\_auctions/closedauction$
Q_{13}	$/site/regions/*/item$
Q_{14}	$/site/*/australia$
Q_{15}	$/site/open\_auctions/open\_auction[@id =' openauction0']/bidder$
Q_{16}	$/site/regions/asia/item[@id =' item4']/mailbox/mail/from$
Q_{17}	$//keyword$
Q_{18}	$/site/closed\_auctions//itemref$

partition size for partitioning the XML documents. We used the default inclusion level (TAG) for these experiments.

Figures 17 and 18 show the performance of such queries for XMark factors of 0.5 and 1 (100MB) respectively. Figure 19 shows the average values for the same experiment over 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}, the open_auctions partition is loaded which has a size of 15MB (due to the fact that each partition must be a well-formed XML document, as explained in Section 4.1). 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 anyways, except for the rare case when a navigation pattern specifies a child at a particular position (e.g., 1000^{th} child).

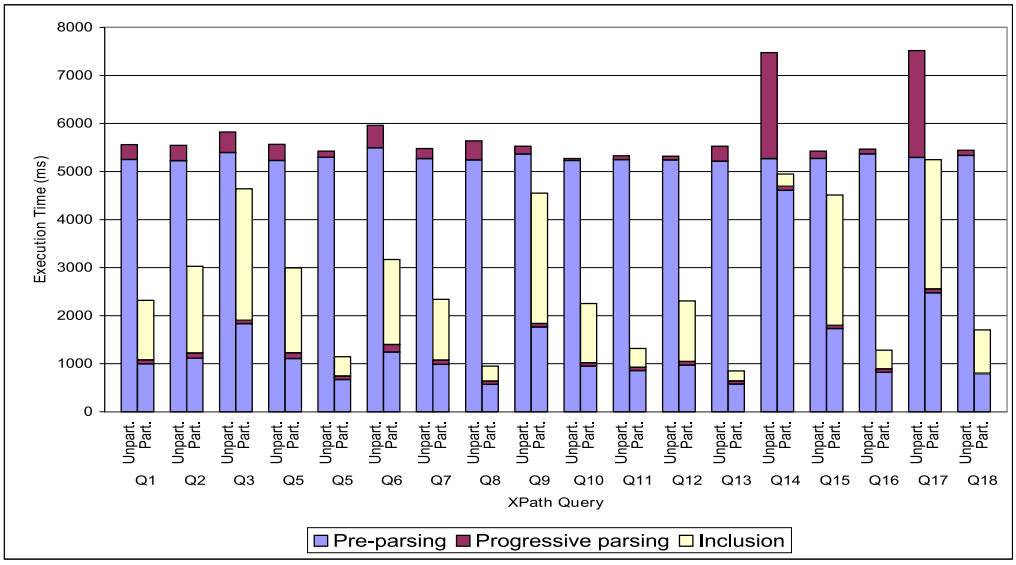


Fig. 17. XPath Query Performance for XMark Factor = 0.5 using the performance XPath queries from XPathMark.

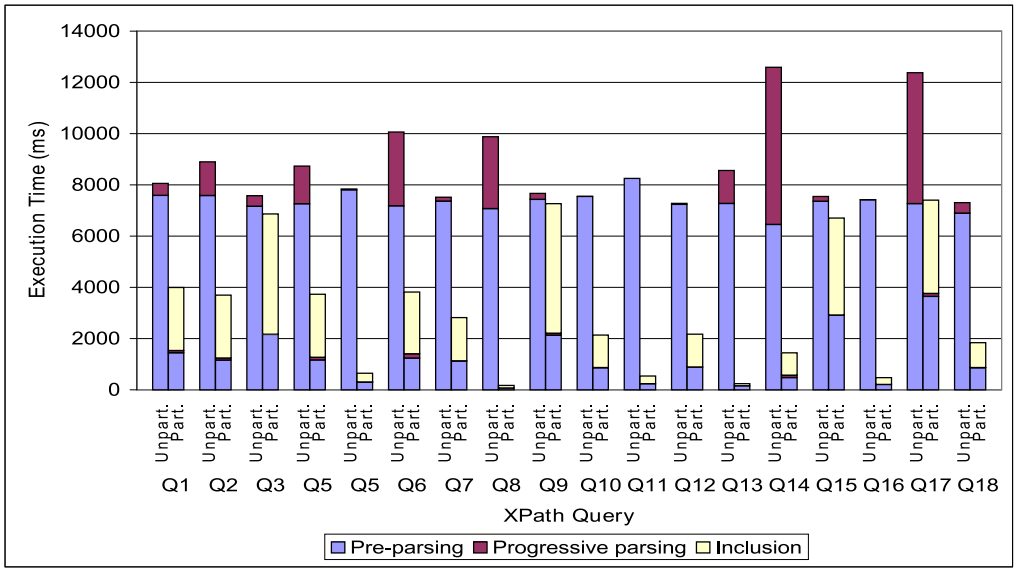


Fig. 18. XPath Query Performance for XMark Factor = 1 using the performance XPath queries from XPathMark..

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 the DOM model 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

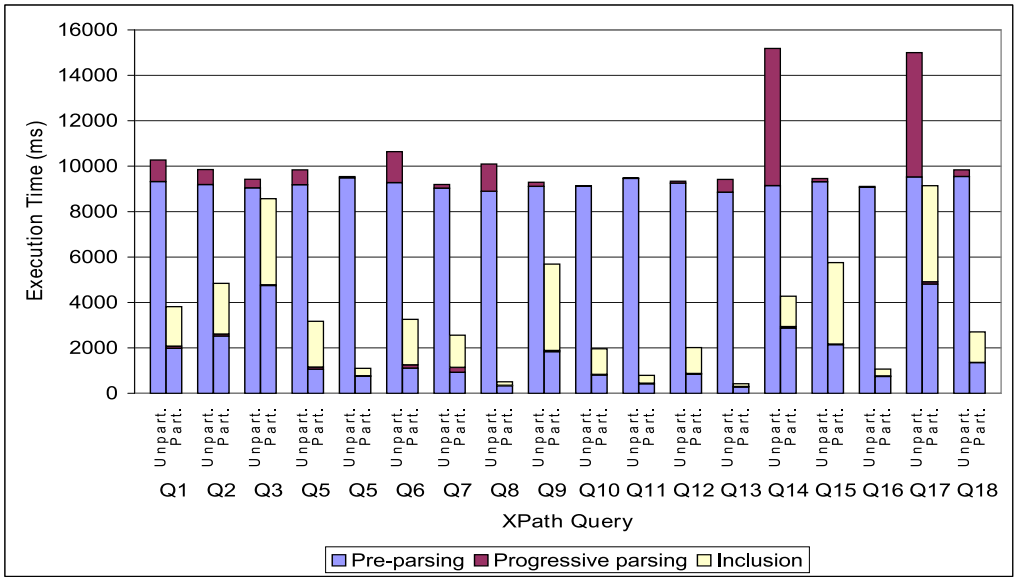


Fig. 19. Average XPath Query Performance for XMark factors from 0.050 to 1.000, using the performance XPath queries from XPath-Mark.

CPU intensive, delaying the process of including the new partition.

Inclusion Levels: We now experiment with the inclusion levels described in Section 3. Initially, we observe the increase of space required by the partitions given the distinct inclusion levels, compared against the original unpartitioned document. We partitioned a document with XMark Factor = 0.5.

Figure 20 presents the results of this experiment, showing low space overhead even when the full information of the partition root is added to the XInclude element. Compared to the size of the original size and to the size of the partitioned file with inclusion level NONE, we can say that practically no overhead exists. We can see how the third inclusion level has the same overhead as the second one. This is due to the fact that most of the nodes that contain text are leaf nodes, and none of the internal nodes that were chosen to root a new partition contain text values.

Figure 21 shows the average query execution time performance when XPath queries are executed over partitioned documents with different Inclusion levels. We picked several XPath queries that represent different categories of queries and different axes. Given the practically inexistent space overhead discussed above, adding information about the root element of the partition in the XInclude physical pointer can give us a significant percentage of gain. In particular, the TAG_ATR level is generally the best choice.

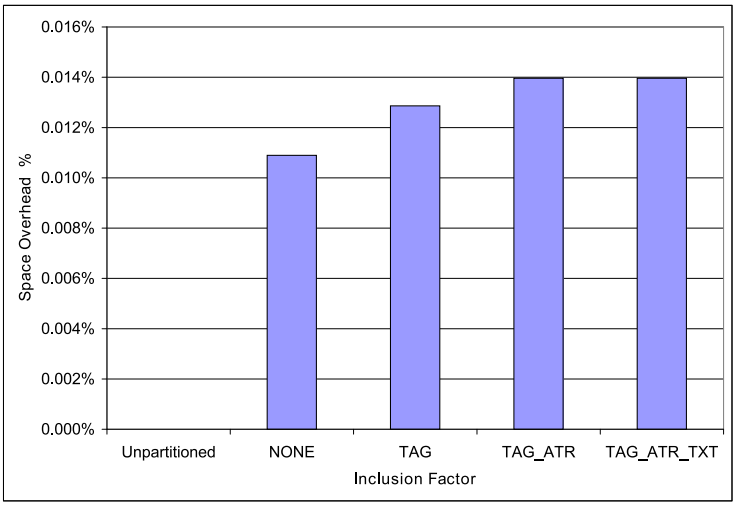


Fig. 20. Space Overhead for Inclusion Levels.

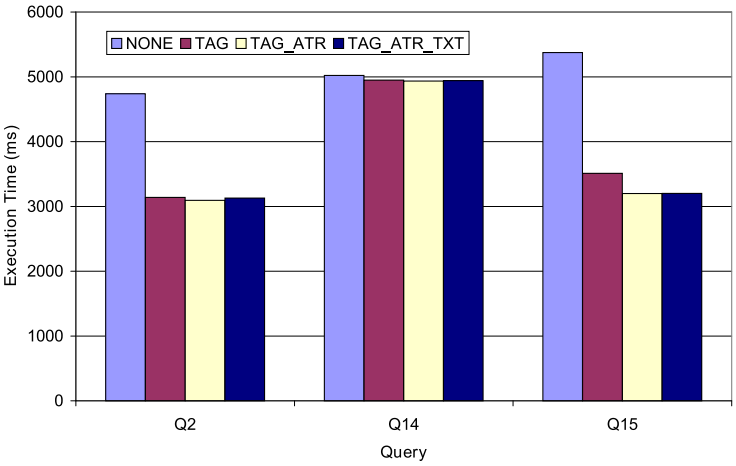


Fig. 21. Execution Time with Inclusion Levels.

7.3 Partition Unloading

In this section we evaluate the performance of 2LP when the amount of main memory designated to store the DOM tree is limited and hence, the partition unloading mechanism is required. We simulated this by introducing a *total memory threshold factor* M_T, which takes into account the DOM overhead claimed by [41] which concludes that a DOM document can expand in main memory up to three or five times the size of the XML file. This factor models the limited number of Megabytes that can be allocated by the DOM document at any given moment.

For this experiment, we repeated the execution of our performance evaluation XPath queries, but this time adding the Partition Unloading mechanism to our 2LP. The XPath queries from Table 3 were executed sequentially, without resetting the DOM tree to the initial partition P_0, this with the objective of

having several partitions loaded before each query was executed. To simulate the *Total Memory Threshold* M_T, we set the Java Virtual Machine's maximum java heap size to 450MB, and used our partitioned XML document for xmark factor = 1.0.

Figure 22 shows the total amount of main memory allocated by 2LP after each query is executed. The JVM Memory Limit resembles the *total memory threshold factor* M_T, as limited by the JVM maximum heap size. We measured the performance of 2LP without Unloading mechanism as well as the behavior of the unloading mechanism following two strategies: *First-In-First-Out (FIFO)* and *Least-Recently-Used (LRU)*, both as used in the context of main memory page replacement. Both strategies restrict the loaded partitions according to M_T, replacing the appropriate partition as dictated by each strategy.

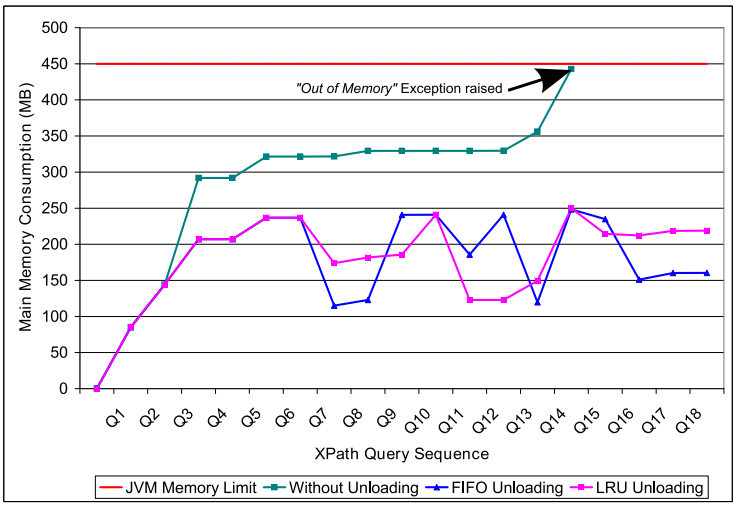


Fig. 22. Loaded Partitions after Sequential Query Executions.

The execution of 2LP without Unloading followed a behavior as shown in Figure 18, where at a point after the execution of Q_{14}, the application crashed with an "Out of Memory" exception, not being able to perform the total execution of our query load. Using a lazy parser like Xerces on the unpartitioned document leads to the same behavior. This was due to the fact that Q_{14} uses a wildcard and requires almost the whole tree to be loaded into main memory. In contrast, both *FIFO* and *LRU* approaches for the unloading strategy were able to manage the critical point of loading several partitions during query Q_{14}, working properly until the last query was executed even under the main memory limitations. We can see that our unloading approach has the potential to scale better to parse large documents under limited memory conditions, whereas current approaches including Xerces will raise "Out of Memory" exceptions.

Both *FIFO* and *LRU* strategies lead to similar behaviors, with slight differences in the order in which the partitions are unloaded as shown for queries Q_7

and Q_{12}; in Q_7 a larger partition is unloaded by the *FIFO* strategy, whereas *LRU* unloads a smaller one. Similarly, for Q_{12}, the *LRU* strategy selects a large partition to unload, while the partitions unloaded by *FIFO* are not as large.

Figure 23 compares the execution time of the XPath queries when the 2LP utilizes the Unloading Mechanism. The figure contains the execution times for 2LP with no Unloading Mechanism, as well as both *FIFO* and *LRU* strategies. The execution of the queries was performed sequentially as explained before, and this caused the first two queries to perform similarly under the three conditions, since the same partitions have to be loaded in the same order to solve the query. For the execution of query Q_3, the size of the overall DOM tree has surpassed the memory threshold and hence one partition has to be unloaded, meaning a penalty in the total execution time. Queries Q_4 and Q_6 show a similar performance for the three variants, since all the partitions that are needed to solve these queries are already loaded into memory in these specific moments, not needing to parse any new partitions. Also none of the currently loaded partitions were unloaded by these queries. In the case of queries Q_{14} and Q_{17}, the maximum memory threshold was reached several times during the query execution, given the large number of partitions required to be parsed by the 2LP. This causes a lot of partitions to be unloaded during the query execution, drastically penalizing the total execution time. Queries Q_3 and Q_9 need to navigate the open_auctions subtree, requiring a larger amount of processing time given the large size of such subtree.

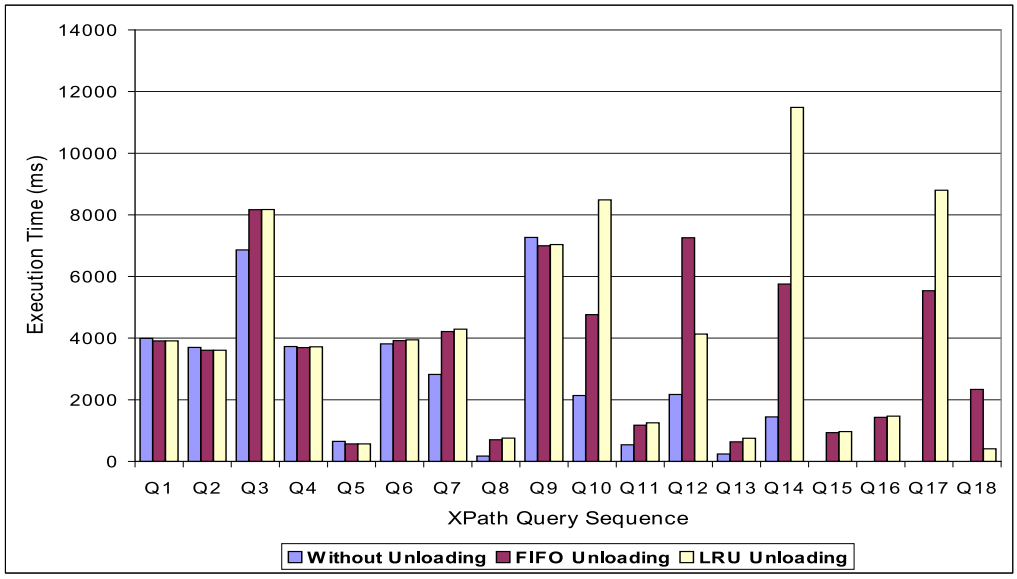


Fig. 23. Execution Time with Unloading Mechanism.

Figure 24 shows the number of partitions that are loaded, re-loaded and unloaded during the execution of each XPath query. A loaded partition means that it has been parsed for the first time by the 2LP. An unloaded partition

is one that has been chosen by the Unloading Mechanism to be discarded. A re-loaded partition is one that has been previously discarded but it is needed to satisfy the query and hence is parsed again.

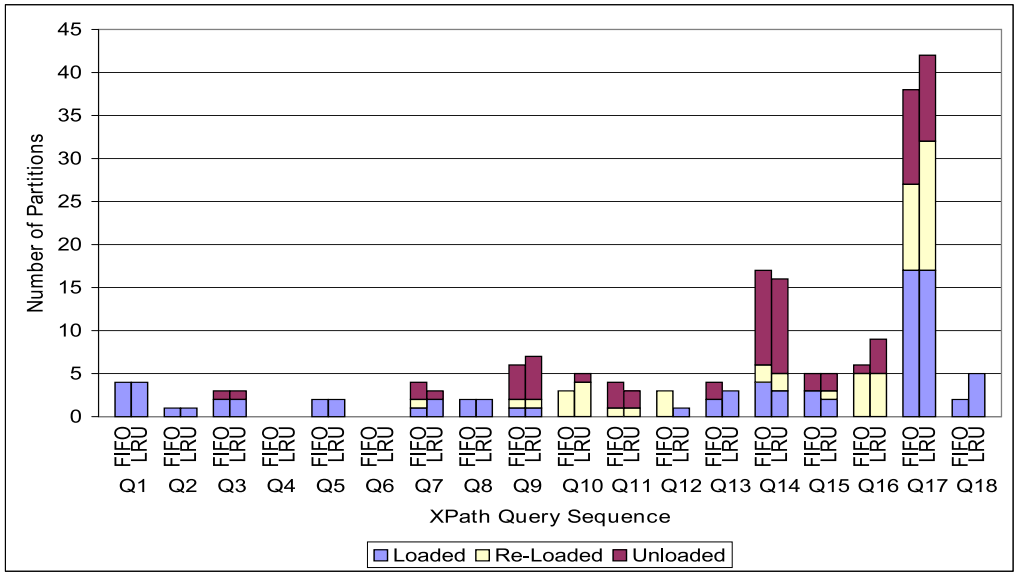


Fig. 24. Loaded, Re-Loaded and Unloaded Partitions.

Again we can see how the performance of queries Q_1, Q_2, Q_3 and Q_4 is similar; the same partitions have to be loaded and unloaded for these cases. As observed in Figure 23, the execution of queries Q_4 and Q_6 do not require the parsing of any new partitions, since all the necessary partitions are already in main memory. The performance of queries Q_{14} and Q_{17} is also related to the behavior in the previous figure. The wildcards and descendant operators require a large number of partitions to be parsed and with this, a large number of partitions to be unloaded as well.

We can also see that both the *FIFO* and *LRU* strategies behave similarly in terms of re-loaded partitions. In terms of total execution time, *LRU* is penalized by the reordering of the partitions in the internal data structures of the strategy.

8 Related Work

Nicola and John [41] have identified the XML parsing process as a bottleneck to enterprise applications. Their study compares XML parsing in several application domains to similar applications that use relational databases as their backend. Operations such as shredding XML documents into relational entities, XPath expression evaluation and XSLT [3,16] processing are often determined by the performance of the underlying XML parser [41], limiting

the massive embracement of native XML databases into large-scale enterprise applications.

Noga, Schott, and Löwe [42] present the idea of Lazy Parsing as presented in the Section 1. The virtual document tree can potentially be stored on disk to avoid the pre-parsing stage; however, the virtual document tree has to still be read from disk. Schott and Noga apply these ideas to the XSL transformations [45]. Kenji and Hiroyuki [36] 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.

Lu et al. [39] present a parallel approach to XML parsing, which initially pre-parses the document to extract the structure of the XML tree to then perform a parallel full parse. This parallel parsing is achieved by assigning the parsing of each segment of the document to a different thread that can exploit the multi-core capabilities of contemporary CPU's. Their pre-parsing phase is more relaxed than the one proposed by [42] and that we use throughout our work; this relaxed pre-parsing only extracts the tree shape without additional information, and is used to decide where to partition the tree to assign the parsing sub-tasks to the threads. This partitioning scheme differs from ours since it is performed after the pre-parsing phase is executed, whereas ours is performed a priori, with the objective of optimizing such pre-parsing stage.

There have been efforts in developing XML pull parsers [15] for both SAX and DOM interfaces. Also, [14] presents a new API built just one level on top of the XML tokenizer, hence claiming to be the simplest, quickest, and most efficient engine for processing XML.

Huang et al. [32,33] present a pre-filtering framework to improve the efficiency of XPath processing over large XML documents with the existing DOM and SAX models. Their framework utilizes an inverted index and a tiny search engine that locates the useful fragments that may be candidates to satisfy the input XPath query, and only these fragments are submitted to the XML parser. In contrast to our approach which is minimally invasive and is compatible with current XML parsers and standards, they use specialized proprietary storage and processing mechanisms.

Van Lunteren et al. [48] 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, Miklau, Onizuka and Suciu [29], to lazily convert an XPath query into a Deterministic Finite Automata (DFA). After this conversion is performed, 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 [37] presents techniques to use functional programming to construct better XML Parsers.

Kanne and Moerkotte [35] have worked on tree partitioning algorithms, but their techniques are more oriented to low-level disk placement, mapping each partition to a single block on the disk drive to be further exploited by native XML data stores like Natix [40].

Several works have been proposed in the area of XML compression. Some of these works [25,38] require the document to be decompressed before any query or navigation can be performed over the XML data. Some others, considered *query-friendly* [22], only require a small subset of the document to be decompressed. Some recent works [23,47,49] can support navigation in the compressed document. SDOM [23] proposes a succinct way of representing XML documents in order to reduce their memory fingerprint and allow efficient navigation. However, SDOM still incurs the pre-parsing cost. Furthermore, their representation is not backwards compatible with current XML parsers. [22,25,38,47,49] have similar limitations. These XML compression and parsing techniques could be viewed as complementary to our work since we mainly optimize the pre-parsing stage with a slight optimization of the progressive parsing stage and they mainly optimize the latter one.

9 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 optimize the pre-parsing phase during which the whole document is read. In this paper, we address this problem by enabling laziness in the pre-parsing phase as well. 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. Additionally, we implemented a dynamic partition unloading mechanism that can enables parsing in memory-limited systems, allowing us to parse and navigate large documents under conditions wherein other parsers typically fail. To aid partitioning decisions, we have proposed a theoretical model for the processing of partitioned documents and presented methods to compute optimal partition sizes. We have experimentally showed that 2LP outperforms other deferred evaluation techniques such as Xerces Java Parser.

10 Acknowledgements

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

References

[1] Apache Xerces2 Java Parser. http://xml.apache.org/xerces2-j/, 2008.

[2] Document Object Model (DOM). http://www.w3.org/DOM/, 2008.

[3] Extensible Stylesheet Language (XSL). http://www.w3.org/TR/xsl/, 2008.

[4] Galax. *http://www.galaxquery.org*, 2008.

[5] Geography Markup Language. http://opengis.net/gml/, 2008.

[6] Health Level Seven XML. http://www.hl7.org/special/Committees/xml/index.cfm, 2008.

[7] Medical Markup Language. http://www.ncbi.nlm.nih.gov/pubmed/10984873?dopt=Abstract, 2008.

[8] OpenDocument Specification v1.0. http://www.oasis-open.org/committees/download.php/12572/OpenDocument-v1.0-os.pdf, 2008.

[9] OpenOffice XML File Format v1.0. http://xml.openoffice.org/xml_specification.pdf, 2008.

[10] Simple API for XML (SAX). http://www.saxproject.org/, 2008.

[11] Xalan-Java. http://xml.apache.org/xalan-j/, 2008.

[12] XML Inclusion. http://www.w3.org/TR/xinclude/, 2008.

[13] XML Pointer Language (XPointer) Version 1.0. http://www.w3.org/TR/WD-xptr, 2008.

[14] XML Pull Parser (XPP). http://www.extreme.indiana.edu/xgws/xsoap/xpp/, 2008.

[15] XML Pull Parsing. http://www.xmlpull.org/index.shtml, 2008.

[16] XSL Transformations. http://www.w3.org/TR/xslt, 2008.

[17] XT. http://www.blnz.com/xt/index.html, 2008.

[18] S. Abramsky. The Lazy Lambda Calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Welsey, Reading, MA, 1990.

[19] T. Bernes-Lee. Universal Resources Identifiers. http://www.w3.org/designissues/axioms.html, 2008.

[20] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible Markup Language (XML) 1.1. W3C Recommendation, World Wide Web Consortium. http://www.w3.org/TR/xml11/, 2006.

[21] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java.* Prentice Hall, Englewood Cliffs, NJ, second edition, September 2003.

[22] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Efficient memory representation of xml documents. In *DBPL*, pages 199–216, 2005.

[23] O'Neil Delpratt, Rajeev Raman, and Naila Rahman. Engineering succinct dom. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 49–60, New York, NY, USA, 2008. ACM.

[24] Zoran Dimitrijevic and Raju Rangaswami. Quality of Service Support for Real-time Storage Systems. *Proceedings of International IPSI Conference*, October 2003.

[25] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching xml data via two zips. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 751–760, New York, NY, USA, 2006. ACM.

[26] Massimo Franceschet. *XPathMark: An XPath Benchmark for the XMark Generated Data.* 2005.

[27] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, 1997.

[28] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. Hong Kong, 2002.

[29] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML Streams with Deterministic Automata. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, pages 173–189, London, UK, 2002. Springer-Verlag.

[30] Object Management Group. UML Resource Page. http://www.uml.org/, 2008.

[31] Torsten Grust. Accelerating XPath Location Steps. In *SIGMOD Conference*, 2002.

[32] Chia-Hsin Huang, Tyng-Ruey Chuang, and Hahn-Ming Lee. Prefiltering techniques for efficient XML document processing. In *DocEng '05: Proceedings of the 2005 ACM symposium on Document engineering*, pages 149–158, New York, NY, USA, 2005. ACM.

[33] Chia-Hsin Huang, Tyng-Ruey Chuang, James J. Lu, and Hahn-Ming Lee. XML Evolution: a two-phase XML processing model using XML prefiltering techniques. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1215–1218. VLDB Endowment, 2006.

[34] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A Native XML Database. *The VLDB Journal*, 11(4):274–291, 2002.

[35] Carl-Christian Kanne and Guido Moerkotte. A linear time algorithm for optimal tree sibling partitioning and approximation algorithms in Natix. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 91–102. VLDB Endowment, 2006.

[36] Manaka Kenji and Sato Hiroyuki. Static optimization of XSLT stylesheets: template instantiation optimization and lazy XML parsing. In *DocEng '05: Proceedings of the 2005 ACM symposium on Document engineering*, pages 55–57, New York, NY, USA, 2005. ACM.

[37] Oleg Kiselyov. A Better XML Parser through Functional Programming. *Lecture Notes in Computer Science*, 2257:209+, 2001.

[38] Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data. pages 153–164, 2000.

[39] Wei Lu, Kenneth Chiu, and Yinfei Pan. A Parallel Approach to XML Parsing. *The 7th IEEE/ACM International Conference on Grid Computing (Grid2006)*, Barcelona, Spain, September 28-29, 2006.

[40] Natix. http://www.dataexmachina.de/. 2008.

[41] M. Nicola and J. John. XML Parsing: a Threat to Database Performance. *CIKM*, 2003.

[42] Markus L. Noga, Steffen Schott, and Welf Löwe. Lazy XML processing. In *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*, pages 88–94, New York, NY, USA, 2002. ACM.

[43] Christ Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *Computer*, 2:17–28, 1994.

[44] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. *VLDB*, 2002.

[45] Steffen Schott and Markus L. Noga. Lazy XSL transformations. In *DocEng '03: Proceedings of the 2003 ACM symposium on Document engineering*, pages 9–18, New York, NY, USA, 2003. ACM.

[46] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts 7th Edition with Java 7th Edition.* John Wiley & Sons, 2006.

[47] Pankaj Tolani and Jayant R. Haritsa. XGRIND: A query-friendly XML compressor. In *ICDE*, 2002.

[48] J. van Lunteren, T. Engbersen, J. Bostian, B. Carey, and C. Larsson. XML Accelerator Engine. *First International Workshop on High Performance XML Processing*, 2004.

[49] Raymond K. Wong, Franky Lam, and William M. Shui. Querying and maintaining a compact xml storage. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 1073–1082, New York, NY, USA, 2007. ACM.

A Disk Drive Modeling

As mentioned in Section 2.2, we base our disk drive modeling on the work of [43]. In their model, seek, rotation, and transfer times, combine the features presented in Section 2.2.

Table A.1 presents the disk drive *transfer time* (t_{transf}) and *random access time* (t_{rand}), required to transfer and access a disk block respectively, for the four hard drive disks we utilize for our theoretical model and experimental section. The values presented in this table were gathered from the manufacturers data sheets.

Table A.1
Hard Drive Modeling Parameters

Disk Model	Maxtor 6L020J1	Quantum Fireball+ KX27.3	Seagate Cheetah 15K.4	Hitachi UltraStar 10K300
Formatted capacity (GB)	20	27.3	36.7	73.4
Heads	1	16	2	3
Rotational Speed (RPM)	7200	7200	15000	10025
Stroke (ms)	17.8	15	7.9	10
Transfer (MBps)	54.2	66.6	200	134.375
Block count	40,132,503	54,600,000	71,687,372	143,374,804
Cylinders	16 383	16 383	50 864	65 494
Avg. seek	8.5	8.5	3.5	4.3
Track switch	0.8	0.8	0.2	0.4
Full Stroke	17.8	15	7.9	10

Notice that according to their model definition, the typical seek times are the average *seek*, *track-to-track seek*, and *full stroke*. We also consider the analysis of the average seek distance, utilizing one third of the full stroke as the average distance seek.

The equations in Table A.2 describe the Gamma function that models the head positioning effects as stated in [43], approximating the measured seek-time profile for the different disk drives.

Table A.2
Gamma Function: Seek Curve Modeling

seek distance	γ (distance) (ms)
$< 1/3$ Cylinders	$a + b \cdot \sqrt{distance}$
$\geq 1/3$ Cylinders	$c + d \cdot distance$

As stated in A.2 the average seek distance will be less than one third of the cylinders, we use the first equation to calculate γ. Table A.3 summarizes the values for the four parameters a, b, c and d, as well as the Gamma function value and the final Transfer Time and Random Access Time.

Table A.3
Gamma Values, Transfer and Random-access Time

Disk Model	Maxtor 6L020J1	Quantum Fireball+ KX27.3	Seagate Cheetah 15K.4	Hitachi UltraStar 10K300
a	0.694374	0.694374	0.174460	0.373425
b	0.105626	0.105626	0.025540	0.026575
c	3.850000	5.250000	1.300000	1.450000
d	0.000851	0.000595	0.000130	0.000131
$\gamma(1/3\,cylinders)$	1.275209	1.192346	0.359615	0.528011
t_{transf}	0.009446	0.007688	0.002560	0.003810
t_{rand}	5.441876	5.359013	2.359615	3.520530