# LazyDOM
## *Transparent Partial DOM Loading and Unloading for Memory Restricted Environments*

Daniel Peintner, Richard Kuntschke, Jörg Heuer
*Siemens AG, Corporate Technology, Information & Communication, 81730 Munich, Germany*

Harald Kosch
*University of Passau, Chair of Distributed Information Systems, 94032 Passau, Germany*

Keywords: XML, DOM, EXI, LazyDOM.

Abstract: Processing XML documents using the Document Object Model (DOM) usually requires loading the entire document into an in-memory DOM prior to processing. Since the in-memory size of a DOM generally is a multiple of the original XML document size, the resulting DOM often consumes a lot of memory and might not even fit into the available memory on memory restricted devices. The LazyDOM approach presented in this paper divides a DOM into XML fragments and loads or unloads these fragments transparently on demand during processing. Thus, the LazyDOM only loads the parts of a DOM that are actually currently needed by an application and unloads them if they are no longer required and memory needs to be freed for other processing tasks. Besides enabling DOM-based processing of large XML documents on memory restricted devices, this approach is able to reduce the amount of memory required for DOM processing at any given time and can also increase the performance of DOM loading if only parts of a DOM are actually needed by an application.

## 1 INTRODUCTION

Many XML tools and XML-based applications directly or indirectly rely on the Document Object Model (DOM) (Hors and Hégaret, 2004). This includes, for example, the XML Path Language (XPath) (Clark and DeRose, 1999) for specifying and evaluating path expressions on XML document instances, XQuery and XSLT processors, XML Schema validators, XForms, and applications built on top of these tools. Many XML-based processors today work on an in-memory representation of the entire XML document instance, usually represented as a DOM. The in-memory representation of a document can become very large—even larger than the size of the corresponding textual XML file in the file system (about 400% of the file size according to (Marian and Siméon, 2003)). When the file contains a representation of the XML data in the W3C's Efficient XML Interchange (EXI) (Schneider and Kamiya, 2009) format, the discrepancy between file size and in-memory size of the DOM is even larger. Especially when dealing with larger XML document instances, the potentially excessive memory consumption constitutes a problem and may render traditional DOM processing infeasible on embedded and other resource limited devices such as cell phones and digital cameras.

Nevertheless, the flexibility and extensibility of XML makes using XML and XPath desirable even on such limited devices. Consider, for example, the use of SVG (Ferraiolo et al., 2003) for creating animated user interfaces on digital cameras. Enabling such use cases requires a means for loading an XML instance into a DOM and evaluating XPath expressions in the face of restricted memory boundaries. In this paper, we present an approach for dynamically loading and unloading DOM elements using EXI features (Section 3.1).

Since EXI and our solution are seamlessly integrated into the XML stack, our approach can be combined with any existing DOM-based XML technology without any additional modifications. For example, compared to previous solutions (Marian and Siméon, 2003), we do not need to analyze the XPath expression prior to loading the DOM. Instead, we can use the same DOM for several queries, dynamically adapting the loaded parts of the DOM not only between the evaluation of subsequent expressions but also on-the-fly during the evaluation of a single expression. The paper introduces the general concept

Peintner D., Kuntschke R., Heuer J. and Kosch H..
LazyDOM - Transparent Partial DOM Loading and Unloading for Memory Restricted Environments.
DOI: 10.5220/0003338100980105
In *Proceedings of the 7th International Conference on Web Information Systems and Technologies* (WEBIST-2011), pages 98-105
ISBN: 978-989-8425-51-5

of a *lazy* DOM (Section 2) combined with an indexing mechanism for referencing DOM elements to be loaded into memory and a simple but effective strategy for identifying DOM elements to be unloaded when memory becomes scarce (Section 3). Further, we present evaluation results comparing our approach to other DOM and XPath evaluation solutions in terms of memory consumption and query execution speed (Section 4), give guidelines how the solution can be tailored to given demands (Section 5), introduce a real word application (Section 6) and discuss related work (Section 7).

## 2 LAZY DOCUMENT OBJECT MODEL

The introduction shortly alluded to the idea behind our memory sensitive model and highlights its simplicity but also its effectiveness in the sense of reducing memory consumption and increasing processing speed. In the following, we describe the general concept and the requirements of our approach, while the technical realization is subsequently discussed in Section 3.

### 2.1 LazyDOM Concept

Lazy evaluation is a well-known technique in the area of programming languages. Delaying a computation until its result is required can increase performance by avoiding unnecessary calculations. A similar concept can be adopted to operate on an in-memory model of an XML document. The Document Object Model (DOM) is a W3C specification and can be seen as an in-memory representation of an XML information set (Cowan and Tobin, 2004), providing at any time the full information of the actual XML instance to navigate through or operate on.

An XML instance is generally transformed entirely into a DOM while multiple interactions, such as look-up operations, may tackle only certain elements of the XML tree. The possibility of working with such independent XML fragments paves the way for efficient processing and saving runtime memory. Hence, at a given time, only a fragment of the document is required. In the following, we use the term *LazyDOM* for referring to a DOM that partially loads XML elements and unloads elements that are not needed anymore.

Figure 1 depicts a fragmentation example of an XML document that represents an arbitrarily subdivided document. The root element *site* and its children build the basic structure. The third XML level consists of *ghost* elements. *Loaded* elements are present in memory while *ghost* elements indicate that there is no subtree available in memory, unless it has

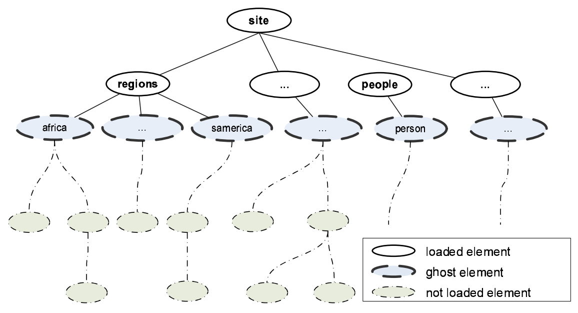


Figure 1: Lazy Document Object Model containing loaded and not loaded elements.

been loaded on demand. Children of *ghost* elements, labeled as *not loaded* element, are parsed and built only if required and can be unloaded later if necessary. This fragmentation can be done in a nested fashion, meaning that a *not loaded* element may contain further *ghost* elements.

### 2.2 LazyDOM Requirements

From the practical point of view, the LazyDOM solution has to face certain requirements to fulfill the described concept. A *ghost* element needs to be parsed independently from the rest of the document. To do so, the basic document needs to allow partial loading while also allowing random access on an element basis.

In XML terms this means that an XML instance is only partially loaded into a DOM and that currently unnecessary elements are skipped. Those skipped portions of a document (*ghost* elements) need to be accessible and indexable without any dependencies concerning previous data (e.g., XML namespace declarations).

The stated requirements could possibly be fulfilled using plain XML, but only with a fair amount of work and complexity. Hence we choose the upcoming EXI format, that is identical with XML on the basis of the Information Set (Cowan and Tobin, 2004) and already offers some of the required capabilities.

## 3 TECHNICAL REALIZATION

This section describes the technical realization of the required features for the LazyDOM. First, we introduce the EXI format and highlight format features we base our work on. Second, we describe the indexing mechanism that enables an efficient look-up of *ghost* elements.

## 3.1 Efficient XML Interchange (EXI)

The EXI format (Schneider and Kamiya, 2009) is a very compact representation of the XML Information Set (Cowan and Tobin, 2004) that is intended to simultaneously optimize performance and the utilization of computational resources. The W3C published the candidate recommendation in late 2009 and is expected to produce the final recommendation by the beginning of 2011.

The EXI format uses a relatively simple grammar-driven approach that achieves very efficient encodings (EXI streams) for a broad range of use-cases. Due to a straightforward encoding algorithm and a small set of data types, EXI processors can be implemented on devices with limited capacity. Besides other relevant properties such as encodings with and even without schema information, as well as schema deviations or partial schemas, the EXI format offers a variety of additional useful features. In this paper, we focus mainly on an aspect called *selfContained* element.

In EXI terms, a *selfContained* element is a portion of an XML document that may be read independently from the rest of the EXI document, meaning that it offers random access on XML element level. The EXI specification itself does not restrain the use of *selfContained* elements to a certain mechanism. An application is free to make use of this capability in a convenient way. The LazyDOM proposal uses this feature to realize the concept of a *ghost* element. Hence, both terms, *ghost* elements and *selfContained* elements, are used as synonyms throughout the paper.

## 3.2 Indexing Mechanism

Subsequently, we introduce a simple, yet powerful indexing mechanism. Based on the previously introduced requirements (Section 2.2) a LazyDOM indexing solution needs to provide three information items.

First of all, a *selfContained* element needs to be uniquely identifiable. The mechanism we are proposing is inspired by XPath (Clark and DeRose, 1999), a language for selecting nodes in XML documents. Let's assume the following XPath expression, composed of three XPath nodes:

`/site[1]/people[1]/person[2]`

This expression uniquely identifies the second `person` node of the first `people` node of the first `site` node. Second, the byte offset of a given *selfContained* element is needed to randomly access the element. Third, the length of each *selfContained* element needs to be known to support skipping of such elements.

Figure 2 depicts the outcome of the indexing format expressed in XML Schema notation. Multiple indices are glued together to a set of indices, depicted as `scIndices`.

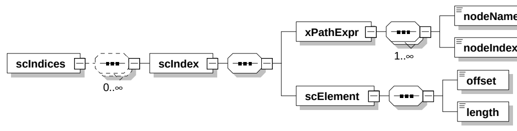


Figure 2: XML Schema for index structure.

A *selfContained* element index (`scIndex`) is composed of an XPath expression (`xPathExpr`) and a *selfContained* element information set (`scElement`). An XPath expression in turn has one to many `nodeName` and `nodeIndex` tuples while the *selfContained* element portion comprises the byte `offset` for random access and the *selfContained* element `length` in bytes.

Listing 1: Index Example Document.

```
<?xml version="1.0" encoding="UTF-8"?>
<scIndices>
    <scIndex>
        <xPathExpr>
            <nodeName>site</nodeName>
            <nodeIndex>1</nodeIndex>
            <nodeName>people</nodeName>
            <nodeIndex>1</nodeIndex>
            <nodeName>person</nodeName>
            <nodeIndex>2</nodeIndex>
        </xPathExpr>
        <scElement>
            <offset>233</offset>
            <length>81</length>
        </scElement>
    </scIndex>
    <!-- more scIndex elements -->
</scIndices>
```

The index example in Listing 1 maps exemplarily the previously used XPath expression to the byte offset (233) in an EXI document. This means that one needs to skip 233 bytes from the start of the EXI stream before reaching the relevant *selfContained* element `person` while the length of the selfContained element is 81 bytes.

The use of XML technologies (e.g., XML Schema) to describe the index has the advantage of being able to use the EXI format to efficiently store the index in an easy consumable way without introducing new requirements or technologies. EXI uses local-names to keep the overhead at a minimum. Instead of coding the element name as a character sequence over and over again, EXI uses compact identifiers that are already part of EXI string tables. Measurements presented in Section 4 take a closer look at the effectiveness of the presented index structure and present numbers for real world data.

### 3.3 LazyDOM Applicability

As stated in the introduction, our goal is to offer the same functionality as any other Document Object Model realization, with the benefit of having just the relevant parts of an XML tree in memory. Hence we propose the LazyDOM solution that uses the described indexing mechanism and initially forms a skeletal structure, meaning that the root element and the entire substructure up to any indexed element is resolved and loaded into memory (see Figure 1).

An XPath engine or any application using a DOM does not see any difference to a conventional DOM implementation. A DOM node or element has to provide capabilities for navigating the tree. Each node has a link to the parent, the previous and the next sibling, as well as a complete list of child-nodes. The LazyDOM solution provides exactly the functionality required by the W3C DOM specification without any restrictions.

## 4 MEASUREMENTS & EVALUATION

In this section, we provide test results showing that the memory sensitive approach presented in this paper works well for real test data. For comparison with other projects, such as Projecting XML Documents (Marian and Siméon, 2003), as well as for traceability, we use test-data generated with the *XMark Benchmark Project*[1]. In addition, we use Java's Management Extensions (JMX)[2], a Java technology that supplies tools for managing and monitoring Java applications, e.g., in terms of memory consumption.

### 4.1 Test Data

XMark provides a data generator that we used to produce XML documents with sizes varying from 1 to 116 MB (e.g. the document xmark-f-0-10.xml was produced using the XMark factor 0.10, see Table 1).

Our comparison is split into two steps. First of all, we produce EXI encoded streams, which constitute counterparts to the XMark generated textual XML documents. Second, we test those generated XML / EXI documents applying several XPath queries.

Table 1 shows the different documents and their sizes on disk. EXI stands for the EXI encoded document with *selfContained* elements. We decided to introduce *selfContained* elements for all elements appearing on the third XML tree level. The last column *Idx* shows the overhead of the index structure by mentioning the size of the index and quoting the number of indices (#).

[1]http://www.xml-benchmark.org/

[2]http://java.sun.com/products/JavaManagement/

Table 1: Document sizes in kB.

| TestCase | **XML** | **EXI** | **Idx** | (#) |
|---|---|---|---|---|
| xmark-f-0-01.xml | 1155 | 812 | 7 | (498) |
| xmark-f-0-10.xml | 11597 | 8076 | 72 | (4931) |
| xmark-f-1-00.xml | 115775 | 79133 | 731 | (49256) |

It should further be noted that the size of the index structure compared to the documents itself is less than 1%. This seems like a reasonable overhead in the sense of storage and parsing. Please note also that EXI documents are usually about ten or more times smaller than semantically equivalent XML documents (see EXI Evaluation Note (Bournez, 2009)). Due to the fact that the XMark generator produces instances with string datatypes only and the XML structure is minimal compared to the actual content data, the EXI format can not properly show its expected benefit. Nevertheless, this paper is not about showing EXI's compression efficiency. Instead we focus on a small subset of EXI format features that facilitate building the LazyDOM solution.

### 4.2 Query Set

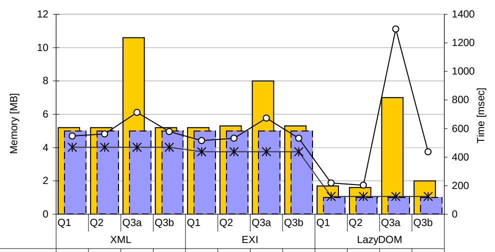
To show the effectiveness of the LazyDOM approach, we introduce three XPath queries. These serve as a basis for the subsequently presented measurements regarding memory consumption and processing time.

**Query Q1.** $/site/people/person[2]/name$
describes a very precise request looking for a person's name with a position index.

**Query Q2.** $/site/regions/*$
returns a node list of regions such as `asia` and `europe`.

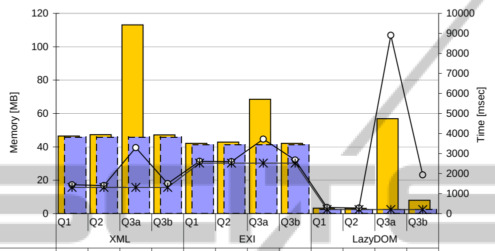
**Query Q3a.** $//closed\_auction[date ='04/16/2000']$
tackles essentially the entire document. The request is stated in such a generic way that an XPath engine is required to traverse the entire XML tree and filter `closed_auction`'s with a certain date.

**Query Q3b.** $/site/auctions/closed\_auction[date ='04/16/2000']$
is semantically equivalent to Query Q3a according to the XMark schema. The improvement is though that the request uses absolute paths to identify the node. We will get back to the significant difference between Q3a and Q3b further below.



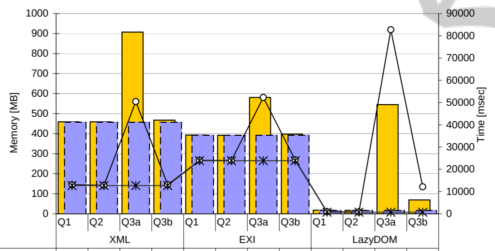
| | XML Q1 | XML Q2 | XML Q3a | XML Q3b | EXI Q1 | EXI Q2 | EXI Q3a | EXI Q3b | LazyDOM Q1 | LazyDOM Q2 | LazyDOM Q3a | LazyDOM Q3b |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Memory [Query] | 5,2 | 5,2 | 10,6 | 5,2 | 5,2 | 5,3 | 8,0 | 5,3 | 1,7 | 1,6 | 7,0 | 2,0 |
| Memory [DOM] | 5,0 | 5,0 | 5,0 | 5,0 | 5,0 | 5,0 | 5,0 | 5,0 | 1,0 | 1,0 | 1,0 | 1,0 |
| Time [Query] | 548 | 562 | 713 | 579 | 516 | 532 | 673 | 532 | 219 | 203 | 1297 | 437 |
| Time [DOM] | 469 | 469 | 469 | 469 | 438 | 438 | 438 | 438 | 125 | 125 | 125 | 125 |

(a) xmark-f-0-01.xml (ca. 1 MB of XML).



| | XML Q1 | XML Q2 | XML Q3a | XML Q3b | EXI Q1 | EXI Q2 | EXI Q3a | EXI Q3b | LazyDOM Q1 | LazyDOM Q2 | LazyDOM Q3a | LazyDOM Q3b |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Memory [Query] | 46,5 | 47,3 | 113,1 | 47,2 | 42,1 | 42,9 | 68,5 | 42,1 | 3,3 | 3,2 | 56,9 | 8,0 |
| Memory [DOM] | 45,8 | 45,8 | 45,8 | 45,8 | 41,2 | 41,2 | 41,2 | 41,2 | 2,5 | 2,5 | 2,5 | 2,5 |
| Time [Query] | 1437 | 1405 | 3297 | 1500 | 2613 | 2597 | 3723 | 2674 | 312 | 281 | 8907 | 1937 |
| Time [DOM] | 1312 | 1312 | 1312 | 1312 | 2519 | 2519 | 2519 | 2519 | 203 | 203 | 203 | 203 |

(b) xmark-f-0-10.xml (ca. 12 MB of XML).



| | XML Q1 | XML Q2 | XML Q3a | XML Q3b | EXI Q1 | EXI Q2 | EXI Q3a | EXI Q3b | LazyDOM Q1 | LazyDOM Q2 | LazyDOM Q3a | LazyDOM Q3b |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Memory [Query] | 459,0 | 459,1 | 907,0 | 468,0 | 393,0 | 392,0 | 581,0 | 398,0 | 18,2 | 17,4 | 545,0 | 69,0 |
| Memory [DOM] | 457,0 | 457,0 | 457,0 | 457,0 | 391,0 | 391,0 | 391,0 | 391,0 | 16,9 | 16,9 | 16,9 | 16,9 |
| Time [Query] | 13075 | 12773 | 50465 | 13094 | 24045 | 23952 | 52313 | 24170 | 953 | 828 | 82704 | 12125 |
| Time [DOM] | 12672 | 12672 | 12672 | 12672 | 23858 | 23858 | 23858 | 23858 | 750 | 750 | 750 | 750 |

(c) xmark-f-1-00.xml (ca. 116 MB of XML).

Figure 3: Memory consumption and query execution times.

## 4.3 Performance Measurements

Our testbed is composed of a notebook running Windows XP with 1.66 Ghz and 2 Gigabytes of RAM. We used the latest Java Virtual Machine 1.6 (with default options). From the many possible XPath engines, we selected the widely used and well established Jaxen[3]engine.

Figure 3 shows a digest of our memory consumption measurements and the according processing times. The bar diagram presents the peak size in MB of the Java heap during execution (e.g., the JVM option `-Xmx128m` sets the maximum heap size to 128 MB) while the line diagram respectively shows query loading/execution times in milliseconds.

[3]http://jaxen.codehaus.org/

The set of queries (Q1, Q2, Q3a and Q3b) groups the three candidates, XML, EXI, and LazyDOM. XML designates parsing and loading the DOM from a textual XML document. EXI in turn parses and loads an EXI document. The LazyDOM, in contrary, parses an EXI document but only loads required portions of the document into the DOM.

To demonstrate the difference between initially loading the XML information set to a DOM and the additional overhead for processing queries we split both measurements. Figure 3 shows memory consumption (Memory) and processing times (Time) for both, DOM loading only (DOM) and DOM loading together with subsequent query execution (Query). Three XMark generated XML test documents (a), (b), and (c) varying in size from 1 MB to 116 MB demonstrate the applicability of the LazyDOM in diverse memory magnitudes.

### 4.3.1 Memory Consumption

Figure 3 illustrates that the memory consumption and processing times for DOM loading is similar for XML and EXI. The reason for the slight difference in memory size is due to the fact that the EXI format prunes insignificant whitespaces.

Our proposed LazyDOM solution shows a huge memory benefit especially for queries Q1, Q2 and Q3b, where only a portion of the entire document is of interest. If we deal with inefficient XPath queries (e.g. Q3a // closed_auction [ date ='04/16/2000' ] ) the entire tree has to be visited. We implemented an approach that unloads *selfContained* elements that are no longer required if memory becomes scarce. We keep track of a list of least recently used *selfContained* elements and remove the least recently used *selfContained* element each time an additional element is to be loaded and the number of loaded elements has reached a configurable number $N$ (e.g., in our test cases $N = 10$). Nevertheless, the gain compared to conventional engines is not as outstanding as expected (see query Q3a in Figure 3). The reason is the implemented caching strategy of many XPath engines. The less an XPath engine makes use of caches, the more memory can be freed by Java's garbage collector.

### 4.3.2 Execution Time

Loading the information set into a DOM shows advantages for the LazyDOM given that only a small amount of data needs to be parsed. In the second measurement step, query processing takes effect. The additional querying process does not indicate any noteworthy execution time difference between the test candidates XML and EXI. This matches our expectations given that there is no difference between a DOM created from an XML or an EXI document.

In terms of processing efficiency, the LazyDOM shows improved runtime performance for XPath expressions requiring the loading or unloading of no or only few *selfContained* elements. When many load or unload operations take place, e. g., because of heavily restricted memory availability, processing efficiency suffers. However, evaluating the query will still succeed instead of failing with an out-of-memory error.

The measurements show that the developed solution represents a well-behaved DOM strategy, loading sub-elements of a document only when the actual need occurs. In any case, the overhead is kept at a minimum. Furthermore, both the granularity of indexed *selfContained* elements in EXI as well as the number *N* of buffered *selfContained* elements are tunable. Hence, we offer a *transparent* mechanism to reduce the overall memory consumption for devices with limited resources as well as for powerful servers.

## 5 DESIGN GUIDELINES

The LazyDOM offers a tunable means to potentially reduce the memory consumption and at the same time improve the performance of DOM-based XML processing. As with many optimization techniques, the amount of improvement that can be achieved depends on the actual use case. The configuration of the LazyDOM and the components interacting with it should be carefully tailored to the demands of the actual application. In this section, we describe some design guidelines that illustrate how the LazyDOM could and should be used in practice to achieve best results.

### 5.1 LazyDOM: Where and When

It should be pointed out that there are use cases where the LazyDOM might not be the best choice. In an environment that is always able to load the entire DOM into memory and that always accesses all or at least most of the data in the DOM, dynamically loading the DOM has little or no benefit in terms of memory consumption but can incur a performance penalty. Consider Query Q3a in Figure 3 as an example. Since the query uses the XPath descendant-or-self axis, all of the DOM is traversed during query evaluation. Thus, the savings in memory consumption when using the LazyDOM is limited compared to a traditional DOM. But the processing time is increased due to the overhead of indexing and dynamic loading.

However, if only parts of the DOM are actually needed, dynamically loading only these parts will not only save memory but will also lead to faster DOM loading since much less data needs to be loaded overall. This can be seen, for example, from the performance results for Query Q3b in Figure 3. The query avoids using the XPath descendant-or-self axis and directly queries only parts of the DOM. Thus, memory consumption and processing time are heavily decreased.

Furthermore, in an environment that is not able to always load the entire DOM into memory due to memory restrictions, LazyDOM is enabling the use of DOM-based XML processing in the first place. If, in this case, only parts of the DOM are actually required by an application, the reduced memory consumption is again combined with improved processing performance as described above. If all or most of the data in the DOM is referenced by an application, processing performance can be worse than when loading the entire DOM into memory up front since repeated loading and unloading of *selfContained* elements will occur. Still, since loading the entire DOM into memory is not possible in memory restricted environments, LazyDOM enables the use of DOM-based XML processing at the cost of potentially increased processing times.

### 5.2 Granularity of *selfContained* Elements

A major means of configuring the LazyDOM is the choice of granularity of *selfContained* elements. Identifying more and smaller *selfContained* elements—e. g., at lower levels of the XML tree—allows more fine grained control over memory consumption since smaller parts of the DOM can be loaded and unloaded. On the other hand, this approach increases indexing overhead and reduces EXI compactness.

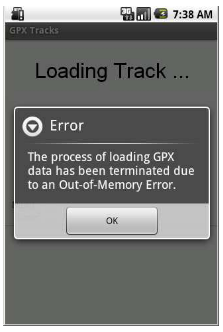
Choosing less and larger *selfContained* elements—e. g., at higher levels of the XML tree—leads to less fine grained control over memory consumption but reduces indexing overhead and improves EXI compactness.

## 6 REAL WORLD APPLICATION

Many memory-restricted device classes such as navigation systems or cell phones run applications that require an in memory model of the data set. A prominent data format is GPX. GPX, or GPS eXchange format is as a common GPS data format for software applications that can be used to describe waypoints, tracks, and routes in XML. We use an Android[4] application that loads GPX instances, shows the track on the display and adds additional user relevant information along the track.

The application demands loading the GPX data into an in-memory model to be able to execute queries

[4]Android is an operating system for mobile devices such as cellular phones, http://www.android.com/



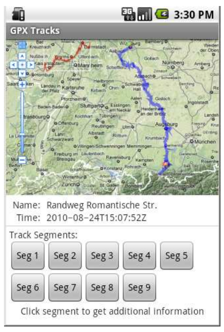
Loading Track ...

Error

The process of loading GPX data has been terminated due to an Out-of-Memory Error.

OK

(a) Without LazyDOM - Out-of-Memory Error.



Name: Randweg Romantische Str.
Time: 2010-08-24T15:07:52Z

Track Segments:

Seg 1 Seg 2 Seg 3 Seg 4 Seg 5
Seg 6 Seg 7 Seg 8 Seg 9

Click segment to get additional information

(b) With LazyDOM - Successfully loaded.

Figure 4: GPX Tracks - Demonstration Tool.

on the data set. Internal tests, run on a widely used cellular phone, highlighted possible memory restrictions. For example, the HTC Hero phone, running Android 2.1 with 288 MB of RAM, ran into an out-of-memory error when trying to load larger tracks into a DOM (see Figure 4(a)). Listing 2 shows a snippet of an example GPX track (with 1.7 MB of GPX data) causing such a problem. We have retrieved the GPX data from GPSies.com, a virtual community (e.g., for sharing bike tracks or trekking routes).

Listing 2: GPX Track Example.

```
<?xml version="1.0" encoding="UTF-8"?>
<gpx xmlns="http://www.topografix.com/
     GPX/1/1">
  <metadata>
    <name>Randweg Romantische Str</name
         >
    <copyright author="GPSies.com" />
    <link href="http://www.gpsies.com/
         map.do?fileId=ewyawkysxmfjwlpp"
           />
    <time>2010-08-24T15:07:52Z</time>
  </metadata>
  <trk>
  + <trkseg> [5469 lines]
  + <trkseg> [4705 lines]
  + <trkseg> [7933 lines]
  + <trkseg> [9225 lines]
  + <trkseg> [5505 lines]
  + <trkseg> [7429 lines]
  + <trkseg> [6065 lines]
  + <trkseg> [5245 lines]
  + <trkseg> [2305 lines]
  </trk>
</gpx>
```

We replaced the default Android document builder with the LazyDOM solution presented in this paper and created *selfContained* elements for each track segment `trkseq`. Hence, each track segment is loaded sequentially and we managed to load the same information set without running into memory problems (see Figure 4(b)).

This real world application shows the easy applicability of the LazyDOM in a memory-restricted environment where an in memory representation of an XML document is needed and available memory is scarce.

## 7 RELATED WORK

(Busatto et al., 2005) presented a technique that represents the tree structure of an XML document in an efficient way by exploiting the high regularity of XML documents. Repetitions of tree patterns are detected and removed. The used technique is a generalization of the approach of sharing common subtrees. If a subtree has occurred before it is represented by a pointer to its previous occurrence. This approach is orthogonal to the LazyDOM. Combining both solutions, the LazyDOM and the efficient representation of the XML tree structure, reduces overall memory consumption even further.

The approach of (Kim et al., 2007) relates to our technique in the sense that large XML documents are partitioned into smaller XML trees. When retrieval operations take place, in contrast, a unify operation is required to unify split child nodes.

The way DOM-based applications, such as XPath, XQuery, or XSLT processors, interact with the DOM can have significant impacts on the improvements that are achievable by combining these applications with the LazyDOM. One example to be mentioned here is the concept of schema-aware XPath processors (e.g., (Paparizos et al., 2007)). Considering Query Q3a in Section 4.2, a non-schema-aware XPath processor has to traverse all of the DOM to answer the query. Thus, the entire DOM needs to be loaded (and potentially unloaded) during the process. Query Q3b uses schema knowledge to eliminate the descendant-or-self axis in the XPath expression of Query Q3a to obtain the same result without having to traverse all of the DOM.

Many efforts have been made to reduce the memory requirements of XQuery and XPath processors. An approach for reducing the memory requirements has been implemented in the Galax XQuery processor (Marian and Siméon, 2003). This approach is based on an a priori analysis of the query to be evaluated. Only the parts of the document needed to correctly and completely process the query are subsequently loaded into an in-memory DOM. Depending on the query, this might require significantly less memory than loading the entire document. However, the analysis process needs to be repeated and the identified necessary parts of the XML document need to be reloaded for each new query. In contrast, the LazyDOM can use the same DOM for evaluating each XPath expression referencing the corresponding XML document. Also, when dealing with queries that need most of or even the entire document during processing, e. g., queries using the descendant-or-self axis on the document root, the a priori analysis of the query yields little or no benefit since all necessary parts of the document need to be loaded into memory as a whole. Our solution allows to dynamically load and unload DOM subtrees during XPath evaluation. Hence, even if the parts of the document necessary for evaluating a certain XPath expression do not fit into memory as a whole, we are still able to correctly process the document within the available memory boundaries.

Streaming Transformations for XML (STX) (Cimprich et al., 2007) allow the transformation of large, theoretically infinite XML documents or XML data streams with bounded memory requirements. STX processes the XML data in a streaming fashion. The limitation of memory consumption during the processing of XPath expressions results from a limitation of the allowed XPath expressions to a subset of XPath that is suitable for streaming evaluation. Thus, the evaluation does not require the buffering of a possibly infinitely large internal state. Compared to STX, our approach is aimed at supporting the full functionality of any DOM-based application.

Other optimization approaches in the context of XPath processing focus on processing performance instead of memory consumption. A special problem in this context is the quick and efficient evaluation of a large set of XPath expressions on a sequence of XML documents as needed in XML-based publish&subscribe systems.

## 8 CONCLUSIONS & OUTLOOK

In this paper, we introduced the LazyDOM as an approach to limit the memory requirements of DOM-based XML processing and to potentially increase the performance of DOM loading. The LazyDOM uses the concept of *selfContained* elements defined in the Efficient XML Interchange (EXI) format to divide a DOM into fragments and to load or unload these fragments on demand during DOM processing. The approach is transparent to DOM-based applications, i. e., no changes need to be made to applications to support the LazyDOM instead of a traditional DOM. Any DOM-based application such as XPath processors, XQuery processors, XSLT processors, XML Schema parsers and validators, etc. can be used with the LazyDOM. An indexing mechanism allows to efficiently jump to the parts of the EXI encoded XML document that need to be loaded.

Our measurement results show that the LazyDOM is able to drastically reduce memory consumption during DOM-based XML processing. The amount of memory needed for processing and the processing performance depend highly on the use case and the configuration of the LazyDOM and the DOM-based applications. We have outlined generic design guidelines that promise to yield good results when followed in practice.

Topics for future work include investigating various cache replacement strategies and their applicability for partial DOM unloading, further investigations concerning suitable LazyDOM configurations for specific use cases especially with respect to the identification of suitable *selfContained* elements in the DOM, and exploiting schema knowledge.

## REFERENCES

Bournez, C. (2009). Efficient XML Interchange Evaluation. http://www.w3.org/TR/2009/WD-exi-evaluation-20090407/. W3C Working Draft.

Busatto, G., Lohrey, M., and Maneth, S. (2005). Efficient memory representation of xml documents. In *Database Programming Languages, 10th International Symposium, Trondheim, Norway*, pages 199–216. Springer.

Cimprich, P., Becker, O., Nentwich, C., Jiroušek, H., Batsis, M., Brown, P., and Kay, M. (2007). Streaming Transformations for XML (STX) Version 1.0. http://stx.sourceforge.net/documents/spec-stx-20070427.html. Working Draft.

Clark, J. and DeRose, S. (1999). XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath. W3C Recommendation.

Cowan, J. and Tobin, R. (2004). XML Information Set (Second Edition). http://www.w3.org/TR/xml-infoset/. W3C Recommendation.

Ferraiolo, J., Jun, F., and Jackson, D. (2003). Scalable Vector Graphics (SVG) 1.1 Specification. http://www.w3.org/TR/2003/REC-SVG11-20030114/. W3C Recommendation.

Hors, A. L. and Hégaret, P. L. (2004). Document Object Model (DOM) Level 3 Core Specification. http://www.w3.org/TR/DOM-Level-3-Core/. W3C Recommendation.

Kim, S. M., Yoo, S. I., Hong, E., Kim, T. G., and Kim, I. K. (2007). A document object modeling method to retrieve data from a very large xml document. In King, P. R. and Simske, S. J., editors, *ACM Symposium on Document Engineering*, pages 59–68. ACM.

Marian, A. and Siméon, J. (2003). Projecting xml documents. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 213–224, Berlin, Germany.

Paparizos, S., Patel, J. M., and Jagadish, H. V. (2007). Sigopt: Using schema to optimize xml query processing. In *ICDE*, pages 1456–1460. IEEE.

Schneider, J. and Kamiya, T. (2009). Efficient XML Interchange (EXI) Format 1.0. http://www.w3.org/TR/2009/CR-exi-20091208/. W3C Candidate Recommendation 08 December 2009.