ScaleDOM
A lazy-loading DOM implementation for processing large XML documents based on Apache Xerces

DOMINIK RAUCH, 0825084, Technical University of Vienna

XML is the defacto standard for human-and-machine-readable markup languages. Loads of document formats, data exchange formats and even database-alike documents are based on XML. While processing average-sized documents is straightforward, handling XML documents of huge size is a challenge - especially if document access should be based on the well-known Document Object Model standard which preserves the tree-nature of XML. ScaleDOM proposes a new Document Object Model implementation for processing large XML documents based on the industry-proven Apache Xerces library, minimizing memory consumption by utilizing transparent lazy-loading and unloading.

Categories and Subject Descriptors: I.7.2 [**Document and Text Processing**]: Document Preparation—*Markup languages*; E.2 [**Data Storage Representations**]

Additional Key Words and Phrases: XML, DOM, Document Object Model, lazy loading

1. INTRODUCTION

Since its introduction in 1996 the Extensible Markup Language (short: XML) has become the defacto standard for human-and-machine-readable markup langauges. It has been a worldwide success, it is simple and yet a general enough specification to be used as a base for a variety of applications. The official specification is published by the W3 consortium. Hundreds of document formats are using XML, many of those are rather small in average size and used as data exchange formats, e.g. nowadays mostly for Internet services. Several document formats based on XML on the other hand are considerably bigger or even huge in size and bring up formidable resource-consumption challanges for XML processors.

While huge documents are often better processed by sequential access parsers like SAX (Simple API for XML), it is often required or at least beneficial to use the officially W3C-provided Document Object Model for XML access. In comparison to SAX where the application processor is only receiving a stream of events, DOM is able to provide applications a full representation of XML's underlying tree structure and far more advanced analysis features.

As indicated in the first paragraph of the introduction, processing huge XML documents with Document Object Model-based APIs can be a challenge for modern computer systems, especially in terms of main memory consumption. This paper gives you a short introduction to the Document Object Model and points out where and why resource problems may occur in current implementations. It will present ongoing research in the field of processing huge XML files and afterwards presents ScaleDOM, a new approach to load huge Document Object Models, clarifying advantages and disadvantages and pointing out possible future development.

2. DOCUMENT OBJECT MODEL

According to the W3C the Document Object Model (DOM) is a platform and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents [Hors et al. 2004]. The Document Object Model is the most common way to access XML documents programmatically, challenged only by the Simple API for XML (SAX).

Nowadays most programming languages offer native implementations of the DOM as part of their standard library. It allows to access the XML document in a structured way. Two programming languages even have official API recommendations by the W3 consortium: JavaScript and Java. While the former is used mainly to manipulate client-side HTML, the latter is one of the currently most-used languages for enterprise software, an area requiring to deal a lot with XML. Furthermore Java itself bases most of its configuration files on XML. In earlier days more Java libraries for accessing XML in a tree-based manner have existed, e.g. the then-popular JDOM library. However, since the standardization of the Java API for XML Parsing (JAXP) with JSR 5 "XML Parsing Specification", which incooperates the official W3C Java API recommendation for the Document Object Model, most of the alternative libraries are not in use anymore.

The best known implementations of the DOM specification are: libxml2 for C which provides bindings for C, C++, Python, Ruby, PHP and many more languages, Microsoft's MSXML which is exposed as a COM interface, Apache Xerces for C++ and its Java counterpart Apache Xerces for Java. Apache Xerces for Java is the only competible implementation of the JAXP standard. Even Sun's (now Oracle's) own implementation included in the Java Development Kit is a repackaging of Apache Xerces.

However, all of those DOM implementations are most concerned with their performance in terms of speed. Comparisons are commonly based on the time libraries spend to load XML documents, not at the memory resources they squander, which results in problems when loading huge XML documents into memory. Although easily compressable, implementations do not make use of such space-savvy ideas and optimize for speed only. In restricted areas this causes memory problems immediately, however, even on enterprise servers memory can become a concern as soon as the XML document reaches a critical size of a few 100 Megabytes.

3. RELATED WORK

There are some approaches to offer Document Object Model support for processing arbitrarily large XML documents, however, many do not support full random document access and are therefore out of scope of this paper.

The first related research field are space-efficient XML parsers. The in-memory DOM tree created from an XML document may be as large as ten times of the size of the original document [Wang et al. 2007]. Space-efficient XML parsers are therefore trading performance with memory consumption by compressing the in-memory DOM tree as far as possible. Two of the most modern parsers in this area are called DDOM and SEDOM. The first, Dictionary-based Document Object Model, uses a dictionary compression approach to reduce the amount of memory used when manipulating a document. Compared to other implemnetations it saves 30 to 80% of the memory [Neumüller 2002], which is a typcial value considering the amount of boilerplate in XML documents (e.g. repeated tags with same name, etc.) . SEDOM is a more advanced version utilizing new compression techniques and performing slightly better and even faster [Wang et al. 2007].

A completely different approach is persisting the DOM in some kind of database format. EMC's xDB is using binary XML and efficient backend structures to provice a standards compliant, non-memory-backed, DOM implementation. This is especially peformant if you want to query the same Document Object Model a lot and do not need to immediately output a little change in the Document Object Model to a new file. After initially parsing the XML document into the database format, xDB shows off with little in-memory usage and a huge performance boost [Probst 2010].

A third approach is to load only necessary data from the XML document (lazy approach). This is also the basis for the proposed ScaleDOM library, however, when implemented in its "pure" way, it does still load the whole document into memory - considering the worst case, where each node of the document is processed at least once. Some initial work in this area has been done by Universitt Karlsruhe [Noga et al. 2002].

VTD-XML [Zhang et al. 2004] does not support the Document Object Model, however, claims to be not only the fastest but also the world's most memory-efficient XML parser. It is a real alternative for many use cases, however, it still requires main memory of 1.3 to 1.5 times the size of the original document. In this paper we're looking into even more memory-savvy solutions which purposely trade performance for memory consumption.

4. SCALEDOM

ScaleDOM is an approach on loading very huge XML documents, providing full Document Object Model support, but using minimal memory resources. Especially extensive memory demand is inherent in all DOM libraries, especially the ones for the Java programming language.

The main idea of ScaleDOM is to load nodes, or levels within the XML tree *lazily* on demand. As implementing the whole DOM specification is a huge project, which would take considerable amount of time and would remove the focus from the ScaleDOM idea it was clear from the beginning that we need to customize an existing DOM implementation instead of reimplementing everything from scratch. We wanted to target the Java market, firstly because it is *the* language for enterprise software development in areas where huge XML document formats are in use and secondly because it is the Information Systems institute's default programming language at TU Vienna. This left us with not many choices, as up-to-date DOM implementations became rare since the introduction of JAXP (see chapter about DOM).

Therefore it was no hard decision to base ScaleDOM on Apache Xerces. In the end this combination has proven strong and ScaleDOM became a drop-in solution to replace Apache Xerces if huge XML file processing is required - an area where Apache Xerces certainly has its limits.

4.1 ScaleDOM Components

The architecture of ScaleDOM divides the system into components which can be exchanged to suit the user's requirements. The following components have been defined:

—I/O service
—XML parser
—Loading strategy
—Node cache

Furthermore the following, non-exchangeable, utiltiy components exist:

—Node location utility
—Low memory detection utility

In the following chapter we present how all those components are implemented and play together to form the whole ScaleDOM system.

4.2 I/O service

The I/O service component is defined by the ReaderFactory interface.

XML documents loaded by ScaleDOM reside on a hard disk and are not already in memory - otherwise memory resources would obviously already be sufficient. In order to load text data like XML from the hard disk Java provides the abstract Reader class, which serves as the data source for the XML parser component later on.

Java Readers are streams, i.e. they do not hold the whole file in-memory but only a small buffer for performance reasons. ReaderFactory implementations must provide two different kinds of Readers: one that reads the file as a whole and another, which returns only a specific part of the file.

In the end an implementaiton of the I/O service interface could provide the same Reader for both requests, however, for performance reasons the default implementation does not. ScaleDOM uses the most performant standard ways for its implementation: a BufferedReader-based approach for reading the whole file, and a RandomAccessFile-based approach for reading a specific part of the file.

A future, alternative implementation of the I/O service component could make use of memory mapping instead. This could be especially intersting for reading a specific part of the file.

4.3 XML parser

The XML parser component is defined by the XmlParser interface.

This component takes a Reader provided by the I/O service component as input and creates events for each XML event occured in the stream. The events are published back to the document.

4.4 Loading strategy

The loading strategy component is defined by the LazyLoadingStrategy interface.

In its simplest and most memory-sparing form the ScaleDOM architecture could really just load one single node at a time, when demanded. However, the structure of XML documents (with start and end of the node possibly far away from each other) would drastically increase parsing times if only one node after the other would be loaded.

Therefore a strategy implementation is queried by the loading process for each received XML event if the event should be dropped or kept. The strategy also has access to context information and can e.g. behave differently during initial loading or at different “tree deepness levels”.

This component is the most likely one to be customized for specific XML document formats as it creates a compromise between memory consumption and performance. The default implementation is loading two levels of nodes during the initial loading, i.e. the root node and the next level. If demanded, the next level for a parent is loaded, if even deeper nodes are required three levels are loaded at once.

4.5 Node cache

The node cache component is defined by the NodeCacheManager interface.

When demanded, nodes are loaded into memory and put inside the DOM tree for processing. However, in the end ScaleDOM would require just as much memory as any other library if nodes would never be unloaded and a document fully processed. Therefore it is important to unload nodes from the DOM in a smart way to keep the memory consumption at a minimum level.

ScaleDOM makes use of Java java.lang.ref.SoftReference inside its DOM to allow Java to free nodes at any point in time if memory is required. The JVM waits to clear SoftReferences only immediately before the virtual machine would run into out-of-memory errors. This has been enough for our proof-of-concept, however, of course in production use it would be more interesting to control which nodes are unloaded in case of memory shortage. This is where the node cache component comes into play.

The node cache components holds strong references instead of soft references to nodes in order to prevent Java from unloading them. When detecting a memory shortage the node cache component removes strong references from its cache list using a specific strategy (e.g. least recently used node). Removing the strong reference leaves the Java VM only with the SoftReference in the DOM, which may be cleared for more available memory.

At the moment a strong reference to the root node of the XML document is always hold in memory and never removed.

4.6 Node location utility

To load nodes on demand it is required to know where those nodes reside in the source file. This is particularly difficult as a Java Reader is only providing us with character positions which does not coincide with its byte offset in the source file as many character encodings nowadays are either mutlibyte character sets or even variable length encodings. The future of StAX parsers (Streaming API for XML) are planning to support this fact by providing byte and character offsets to the populated XML events during parsing, currently ScaleDOM supports only fixed-length encodings. In case you need to process a variable-length encoding file you have to first convert it to a fixed-length encoding. The current implementation of the node location utility is aware of the source file's encoding and converts fixed-length character offsets to file locations and vice-versa.

4.7 Low memory detection utility

The low memory detection utility makes use of Java Management Extensions (JMX) to detect current memory usage and alert the system, especially the node caching system in case of exceeding memory thresholds.

4.8 DOM modifications

DOM modifications are not yet implemented, however, we already know how to. An initial implementation could just prevent all modified nodes from unloading (as the modification is not yet backed by the file). More advanced implementations could save modifications in some persistent store, or even the original file, modifying the reload-locations to enable reloading.

5. CURRENT LIMITATIONS

ScaleDOM has been developed as a proof-of-concept library and therefore faces some limitations which are going to be fixed in future versions. Some of them are just due to limited developer resources others require future research and are discussed in the future outlook section.

—The underlying XML parser supports only 2 GB file size as a maximum.

—Apache Xerces is currently not repackaged within ScaleDOM, therefore you can use either Xerces or ScaleDOM but not both at the moment, as ScaleDOM replaces some internal classes of Xerces' DOM implementation with own classes.

—DTD is not parsed and DTD nodes are simply dropped. Also EntityReference nodes are dropped, however, most parsers encourage you to expand them during parsing anyways. To support EntityReferences one would have to hold the DTD in-memory at all times for reloading purposes.

Unfortunately due to the internal structure of Apache Xerces its XPath implementation transforms the whole DOM into its internal DTM format, therefore XPath is demanding all DOM nodes at once and makes the use of ScaleDOM needless. In the future we think about replacing XPath's DTM format with a lazy adapter, only loading nodes which are required by XPath at the very moment.

Another implementation note is about why we replaced some of Apache Xerces' internal classes instead of providing transparent proxy classes: again, the reason is unfortunately the internal structure of Apache Xerces. Sometimes the Xerces DOM implementation casts to internal types and accesses member fields directly without using an accessor method (e.g. access to the child collection), a type of call which can't be intercepted with a proxy. Furthermore we saved some additional memory as we could remove unused fields and replace calls to getChildNodes() with either getCurrentlyLoadedChildNodes() or getAllChildNodes(), optimizing performance.

A different solution to the problem would be the usage of Aspect-oriented programming (AOP), however, AOP costs a lot of performance and in some cases it has been found that adapting the original classes has been muche easier than AOP would have been.

6. PERFORMANCE EVALUATION

Our initial performance evaluation shows very positive results for accessing huge XML documents via Document Object Model in low memory constraint areas. Our first measurement compared ScaleDOM to Apache Xerces in low- memory environments: 16 MB, 64 MB and 256 MB of max memory are given to the Java VM for opening and fully traversing different XML documents. The more memory we provide to the VM the less re-loading has to be done by ScaleDOM and the more performant it can be.

XML doc size	ScaleDOM (with 16 MB)	Apache Xerces (with 16 MB)	ScaleDOM (with 64MB)	Apache Xerces (with 64MB)	ScaleDOM (with 256MB)	Apache Xerces (with 256MB)
actors20000.xml (14 MB)	41s	NEM	38s	NEM	3s	159ms
treebank_e.xml (84 MB)	NEM	NEM	4m	NEM	1m	NEM
psd7003.xml (683 MB)	NEM	NEM	NEM	NEM	28min	NEM

NEM means that the process exited with an OutOfMemoryError. Each test has been run on the same machine (Intel Core i7 with 3.4 GHz, 8 GB of RAM and an OCZ Vertex II SSD).

The table clearly points out that in many scenarios ScaleDOM gives us the full feature richness of the Document Object Model (even for very large files) where the default Apache Xerces implementation fails. Although Apache Xerces is fast compared to ScaleDOM (given enough memory) Apache Xerces required more than 1 GB (!) to load and traverse the treebank_e.xml example, psd7003.xml is even unparsable on a 32bit machine due to address space limitation. Better lazy loading strategies which adapt to the available memory will improve the speed performance of ScaleDOM in the near future.

7. CONCLUSION & FUTURE OUTLOOK

When handling large files it is still far more convinient to rely on the proven DOM API instead of using a stream-based API like SAX and building all required in-memory structures yourself. Speed is not much behind either and can be tweaked with suitable loading strategies. ScaleDOM's base, Apache Xerces, is an industry-proven framework which delivers us the necessary reliability and robustness. We're planning to release a more in-depth performance evaluation of the framework soon.

In the future we are interested into overcoming the current limitations of ScaleDOM by further enhancing the library and producing an enterprise-ready framework. Another interesting option is to combine our approach with related work, e.g. use compressing algorithms for the part of the XML document currently hold in-memory.

REFERENCES

Arnaud Le Hors, Philippe Le Hgaret, and Lauren Woodand Gavin Nicol. 2004. Document Object Model (DOM) Level 3 Core Specification. (March 2004). Retrieved August 9, 2013 from http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/

Mathias Neumüller. 2002. Compact data structures for querying XML. *EDBT 2002 PhD Worshop* (2002), 127–130.

Markus L. Noga, Steffen Schott, and Welf Löwe. 2002. Lazy XML processing. In *Proceedings of the 2002 ACM symposium on Document engineering (DocEng '02)*. ACM, New York, NY, USA, 88–94. DOI:http://dx.doi.org/10.1145/585058.585075

Martin Probst. 2010. *Processing Arbitrarily Large XML using a Persistent DOM*. Mulberry Technologies, Inc. http://dx.doi.org/10.4242/BalisageVol5.Probst01

Fangju Wang, Jing Li, and Hooman Homayounfar. 2007. A space efficient XML DOM parser. *Data Knowl. Eng.* 60, 1 (Jan. 2007), 185–207. DOI:http://dx.doi.org/10.1016/j.datak.2006.01.002

J Zhang and others. 2004. VTD-XML: The Future of XML Processing. (2004).