

PERFORMANCE ANALYSIS OF JAVA APIS FOR XML PROCESSING

Bruno Oliveira¹, Vasco Santos¹ and Orlando Belo²

¹CIICESI, School of Management and Technology, Polytechnic of Porto
Felgueiras, PORTUGAL

{bmo,vsantos}@estgf.ipp.pt

²Algoritmi R&D Centre, University of Minho
4710-057 Braga, PORTUGAL
obelo@di.uminho.pt

ABSTRACT

Over time, XML markup language has acquired a considerable importance in applications development, standards definition and in the representation of large volumes of data, such as databases. Today, processing XML documents in a short period of time is a critical activity in a large range of applications, which imposes choosing the most appropriate mechanism to parse XML documents quickly and efficiently. When using a programming language for XML processing, such as Java, it becomes necessary to use effective mechanisms, e.g. APIs, which allow reading and processing of large documents in appropriated manners. This paper presents a performance study of the main existing Java APIs that deal with XML documents, in order to identify the most suitable one for processing large XML files.

KEYWORDS

XML, XML Markup languages, XML Documents, Java API, Performance Analysis.

1 INTRODUCTION

Due to the simplicity of its hierarchical structure, XML (*Extensible Markup Language*) is widely used for data representation in many applications. As a result of its portability, XML is used to ensure data interchanging among systems with high heterogeneous natures, facilitating data communication and sharing, it's platform independent, which makes it quite attractive for the majority of applications. Associated with the XML format there are other languages that complement the application area of this format, such as XSD, XSLT or XQuery. Currently, XML format is used

in the development of several types of software, including web pages, web services, network applications, and fully based XML databases.

Access and modification operations are essential to XML files manipulation once they are affected by any increasing amount of data, by the complexity of those operations, and by shorter periods of time needed to process them. Coupled with this data growing, XML documents can reach large number of megabytes (or even gigabytes), limiting and conditioning the technology used for development of applications appealing for XML data processing. Also coupled with the concept of portability, Java programming language provides a set of interfaces allowing for the manipulation of structured documents according to the XML format. Due to their portability, Java and XML are commonly used in application development.

The main focus of this paper was to conduct a study of the various parsing models and APIs (*Application Programming Interface*) for XML processing using Java programming language, with the purpose to supply a refresh benchmark to the available representation models, identifying which is the most suitable for access and transformation of large XML documents. We also refer the main advantages identified for each representation model, always keeping the performance factor in mind. The next section (2) examines some memory and streaming representation models, identifying how documents are processed according to each parsing model. Section 3 and 4 present some memory-based APIs and streaming-based APIs and their features. Section 5 compares performance and memory consumption of

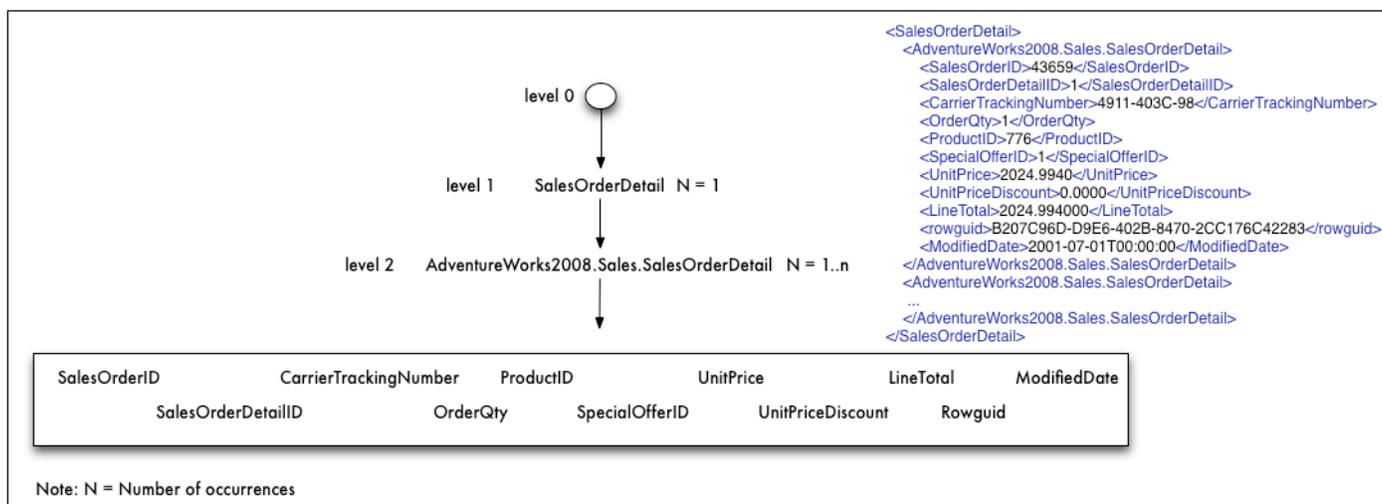


Figure 1. Example of a XML memory tree representation

memory-based APIs and streaming-based APIs. Next, in section 6, we compare modification performance of the best APIs studied previously, terminating the paper summarizing results in section 7.

2 MEMORY and STREAMING REPRESENTATION MODELS

Due to its complexity and importance, the parsing process is the most critical operation in XML processing, directly conditioning processing time and memory consumption. Several studies [1–6] have been conducted with the goal to test and improve representation models and APIs in XML processing [7]. However, some of them did not focus its context on Java programming and others are outdated. This is mainly caused by miscellaneous updates and improvements in the execution environment, particularly in the Java Virtual Machine, which affects, as we know, runtime and effectiveness of the operations.

In [8] the process of handling XML documents was described in four phases: *Parsing* that is considered a critical step in performance, *Access*, *Modification* and *Serialization*, whose performance is directly affected by the parsing models. As the most critical factor of performance, parsing is characterized by the conversion of characters, mainly related to the conversion of characters into a format that a programming

language understands, lexical analysis which is the process that identifies XML elements, e.g. start node, end node or characters, applying regular expressions defined by World Wide Web Consortium (W3C)¹. The last step of the parsing phase is the syntactic analysis of the document, where it is checked if the document complies with the rules of construction of an XML document. Finally, the API implements access and modification operations on the data resulted from the parsing process.

2.1 Memory-based representation models

Most memory-based APIs use a common model in data processing, where XML documents are entirely stored in memory in a tree format with multiple nodes, descending all from a single node representing the root of the tree. This kind of schema allows the use of different methods to locate and manipulate data contained inside the nodes. For each search, or other kind of manipulation, it is necessary to start the processing by the root element continuing in the structure hierarchy to access the remaining data (figure 1). Since all the information is available in memory, we can traverse the tree in random order, changing the positioning of the nodes and performing data transformations in a very simple and accessible way. Considering its memory structure

¹<http://www.w3.org/>

representation, these APIs facilitate application development, providing a wide range of search methods that allow you to easily perform operations on the constituent nodes of the tree. However memory-based APIs consume, in average, four to five times more memory than the document's size. For example, a 20 megabytes document needs, depending on the representation model, approximately 100 megabytes in order to be stored in memory, which may represent a problem in processing large documents.

2.2 Streaming-based models

Streaming-based APIs perform a sequential scan of the document using minimum memory resources. Typically, this type of APIs use the depth of the XML document (number of nested elements) and the maximum data stored in XML attributes on a single XML element. Both of these are always smaller than the size of the memory-based parsing tree approach. Then, a small portion of the document is extracted sequentially without the need to load the whole document structure. Usually, the parser reads the XML document calling a specific method for each type of event to process its object. Figure 2 presents the SAX (*Simple API for XML Processing*)² conceptual model for XML processing, which is similar to other streaming-based APIs.

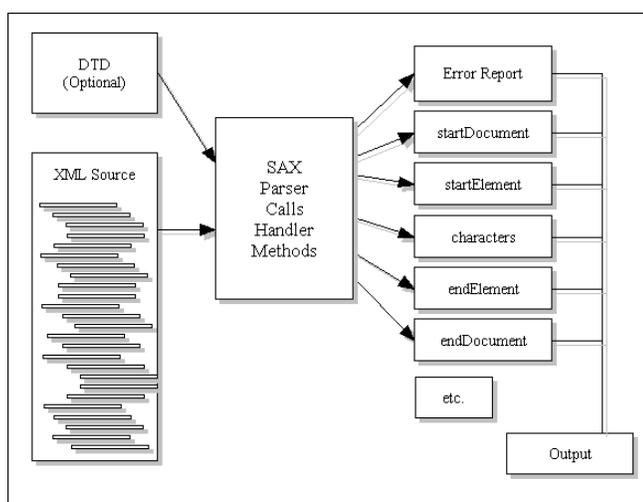


Figure 2. SAX parsing model

² <http://www.saxproject.org>

The parser is configured as an input source, which is associated with a set of content management methods that identify, for example, the beginning or the end of the document and elements of data that might contain errors that occurred during the parsing step. When the parser runs, event triggers are captured by content management methods. Each time the parser detects an important part of the XML document it triggers the appropriate method in order to read the respective data block. Streaming-based APIs are more suitable for processing large XML documents, because, in theory, they can process documents of infinite size.

3 MEMORY-BASED APIS

In order to get a memory-based API overview for XML processing in Java, we conducted a specific study to determine which APIs are more efficient in memory management and which is faster in XML processing. We covered the following APIs: DOM³, XOM⁴, OJXQI⁵, VTD⁶, JDOM⁷, dom4j⁸ and Xerces2⁹. Included in *JAXP package*, DOM API is a collection of classes that has a set of Java methods that allows XML processing in memory with a structure similar to figure 1. In several cases, the DOM API is the basis for the construction of new APIs that revise some of its characteristics, with the aim of serving specific requirements. For instance, the JDOM API allows the manipulation of XML documents with Java via a tree structure representation, thus being similar to DOM. However, this API has been developed specifically for Java language, making it much more intuitive for a typical Java programmer. For example, there is no *Text* class [9], since Java programming language provides its own class (*String* class). JDOM takes advantage of Java features such as: creating methods with the same name, *reflection*¹⁰, *weak references*¹¹, and the use

³ <http://download.oracle.com/javase/6/docs/technotes/guides/xml/>

⁴ <http://www.xom.nu>

⁵ <http://www.oracle.com/technetwork/database/features/xmlldb/index-087544.html>

⁶ <http://vtd-xml.sourceforge.net>

⁷ <http://www.jdom.org>

⁸ <http://dom4j.sourceforge.net/dom4j-1.6.1>

⁹ <http://xerces.apache.org/xerces2-j>

¹⁰ <http://java.sun.com/developer/technicalArticles/ALT/Reflection/>

of collections such as *List* and *Iterator* [10]. JDOM API differs from DOM API in the use of classes instead of interfaces, simplifying the API but limiting flexibility. For his part, the dom4j is an open-source API based on DOM and JDOM concepts, using an interface and abstract base class approach, with extensive use of the Collection classes. dom4j is a more complete solution than JDOM, which gives more emphasis to the use of the interfaces, adding more flexibility at the cost of a little added complexity [11, 12].

Inspired by DOM and JDOM, the XOM API was designed to be the best of both worlds. In Harold's presentation [13], XOM is classified as an easy to use API, fast and simple. XOM makes use of existing Java mechanisms (like JDOM), revealing a far more restricted API that does not allow creation of malformed documents, forcing validations through the use of inheritance. In such overview some disadvantages of JDOM were presented, namely the one that considers it inconsistent since there are several ways to accomplish the same tasks (like reading a child element) and due to some gaps in the use of Java convention (e.g. set methods not always return void). Another disadvantage listed, refers to elements of an XML document that are represented using objects, which produces small memory overheads. In addition, a comparison is also provided with the dom4j that uses interfaces instead of classes resulting in a more complex API. Briefly, we can say that dom4j is an API based on DOM (and extended), and the XOM API based on the principles of DOM with the main goal of simplifying XML processing. JDOM, dom4j and XOM have the advantage of being specifically developed for the Java language, unlike other APIs (like DOM), which were developed in a generic way for several programming languages [11].

XQuery is a language for extracting data from an XML document that allows the creation of a high-level code for extraction of data, similar to what happens with SQL language for relational databases. This language will require native

support from the API that should interpret commands produced from XQuery language. OJXQI (*Oracle Java XQuery API*) is an API proposed by Oracle which is incorporated into its database with support for XQuery language, simplifying XML transformations through the use of a simple language, which is very similar in construction to SQL language. Oracle supports XQuery in two different levels: database and mid-tier. The first one applies queries in the database environment and the second one run queries on sources, which are not databases. Thus, it is possible to compile several clauses allowing XQuery execution, and consequently lead to a new set of results. Data from OJXQI API is entirely processed in memory, allowing the creation of DOM objects in order to represent the data.

The last API that was analyzed, representing XML data in an object tree structure, is named Xerces2¹², and consists in a set of parsers that use DOM and SAX data models. We tested the DOM implementation, which naturally follows the same guidelines in terms of architecture as the previous APIs presented. On the other hand, VTD (*Virtual Token Descriptor*) API uses a different approach, having the premise that the creation of objects is the main factor of low performance. VTD API implements arrays of integers based structure to represent data in memory, eliminating the cost of object creation resulting from the extraction process, through the use of arrays of 64-bit integers (*VTD records*). VTD is a binary format that specifies how to assign tokens (identification codes) in a non-extractive method. The concept of parsing "*non-extractively*" [14] means that XML text remains intact in memory while the tokens are represented exclusively by using ranges and sizes in bits (the contents of the string is not copied) [8].

The process contrasts with the method used by other extractive XML processing models (such as DOM and SAX), which allocate blocks of memory for document contents allocation, manipulating data directly. This manipulation can only be performed after the parsing process has finished

¹¹<http://weblogs.java.net/blog/2006/05/04/understanding-weak-references>

¹² <http://xerces.apache.org/xerces2-j/index.html>

with document size as the largest bottleneck in XML data access performance.

4 STREAMING-BASED APIS

Streaming-based APIs do not maintain long-lived structures of documents in memory. This type of APIs read data as a series of events representing them in a form of objects (like DOM API), using a small portion of memory to process the document in a sequential way. Objects are associated with different types of events and are not maintained too long in memory unlike the approach of memory-based APIs. The JSR (*Java Specification Request*) 173¹³ defines Streaming API for XML (StAX¹⁴), that allow parsing elements in streaming mode, and the extraction of information through events controlled by the application (*pull model*), differing from SAX API of JAXP package, that has a manager that takes events as convenience of the parser (*push model*). While StAX API allows you to discard information in the document's parsing as appropriate (invoking the *nextEvent* method), SAX parser extracts all elements even if you don't need them. In addition, StAX has two integrated APIs with different levels of abstraction: the *cursed-based* API, which is a lower-level API, focused on efficiency and simplicity of use, that works like a stream of events, and the *iterator based* API that offers a higher level of abstraction allowing pipelining, and representing the events through objects. This implementation allows the programmer to ask (*peek() method*) without reading the event. It is possible to skip the input of both the *Cursor* and *Event* approaches. In this study we tested *cursed-based* API because is the most efficient way to read XML data [15]. In addition to SAX and StAX, we also tested XOM API with *NodeFactory* implementation. *NodeFactory* allows parsing the XML document as Streaming like SAX and StAX. SAX, StAX and XOM (streaming mode implementation) allow access to data before the parsing process is completed. This feature allows memory consumption to remain low because processed data, and no longer in need, might be released

¹³ <http://jcp.org/en/jsr/detail?id=173>

¹⁴ <http://stax.codehaus.org/Home>

from memory, thus keeping memory usage low as the parsing process proceeds. Table 1 summarizes all APIs described before.

Table 1. APIs analysis summary

API	Parsing Model
JAXP: Sax	Streaming events: push model.
JAXP: StAX	Streaming events: pull model.
JAXP: DOM	Memory: tree object.
XOM	Memory: tree object.
OJXQI	Memory: tree object.
jDOM	Memory: tree object.
dom4j	Memory: tree object.
Xerces2	Memory: tree object.
VTD	Memory: array of integers.

In order to test memory usage and execution time for each API, we used two different families of XML documents:

- 1) one representing sales orders of a particular company (*SalesOrderDetail*), which was taken from the Microsoft Data Warehouse samples: *Adventure Works*¹⁵;
- 2) an other generated by *xmlgen*¹⁶ tool which aims to represent information about a bidding web site, from an *e-commerce*¹⁷ typical application.

5 PERFORMANCE ANALYSES OF APIS

Table 2 presents the size of the documents and the properties used on tests for each API. We used three instances of different sizes for each document type in order to test not only the size of in-memory representation, but also the elapsed time of parsing each document.

5.1 Memory-based APIs

The study consisted in measurements of memory consumption in megabytes (MB) - (figure 3), and

¹⁵ <http://msftdbprodsamples.codeplex.com/>

¹⁶ <http://www.xml-benchmark.org>

¹⁷ Tests realized in 2.53 Ghz Intel Core 2 Duo, 4 GB 1067 Ghz DDR3, Mac OS X 10.6.4, hard drive with 5400 RPM, 1.6.0_20 – Open JDK Runtime Environment with 455 megabytes of memory available

execution time in milliseconds (ms) – (figure 4) used by each memory-based API.

Table 2. Documents used on tests

File	File size	Number of nodes ¹⁸
SalesOrderDetail1	9,9 MB	20213
SalesOrderDetail2	60,8 MB	121317
SalesOrderDetail3	145,5 MB	304688
AuctionWebSite1	11,7 MB	2175
AuctionWebSite2	58,0 MB	10875
AuctionWebSite3	163,4 MB	30444

Results are based on an arithmetic average resulted from five executions for each API for each document (without considering the time of the first execution). The results shows the gain of VTD in relation to other memory-based APIs, either in terms of memory usage or at runtime, showing that VTD representation model of data is much superior than other APIs representation.

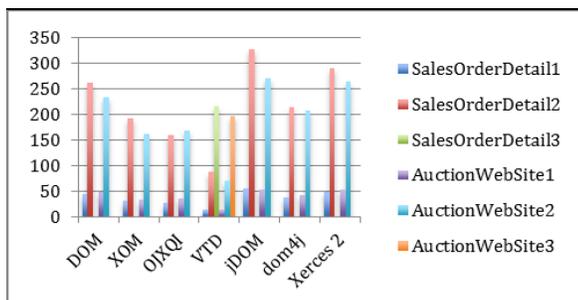


Figure 3. Memory consumption in megabytes of memory-based APIs

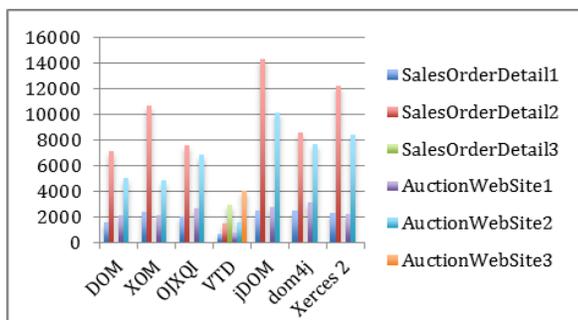


Figure 4. Execution time in milliseconds of memory-based APIs

¹⁸ In this particular scenario, a node represents a data record. For example, in the *SalesOrderDetail* document, one node represents one sales record.

With the exception of VTD, no other memory-based API was able to perform the parsing of the biggest documents with the amount of memory available on Java Virtual Machine (*SalesOrderDetail3* - green bar and *AuctionWebSite3* – orange bar). Noteworthy is the good performance in parsing time of DOM in relation to other memory-based APIs. Although the representation of a DOM document in memory is higher than the XOM and OJXQI representation. When large XML files are used, the memory-based approach is not feasible due to inherent memory limitations.

5.2 Streaming-based APIs

Once memory consumption of streaming-based APIs is reduced, not representing a critical point in terms of processing, we only tested parsing speed in milliseconds for each API: SAX, StAX (was deemed the *cursor-based* API) and XOM (streaming-based approach) (Fig 5) for each of the documents presented earlier.

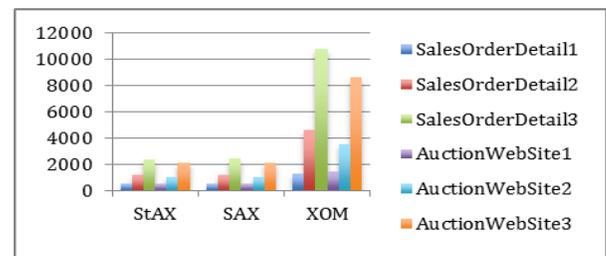


Figure 5. Parsing time in milliseconds from streaming-based APIs

SAX and StAX are very similar in time consumption, which is easily expected, since the main point that distinguishes these two APIs is how the parser handles the events processed. Considering the entire document, the results are quite similar, nevertheless XOM has a much lower performance compared to other streaming-based APIs.

5.3 Comparative analysis of two types of APIs

Memory-based APIs are widely used due to the fact that, in most cases, documents being processed are small enough to fit in memory. However, in cases where memory availability is limited, or the size of the XML document to be

processed is large, streaming-based APIs are the most suitable. Project requirements are crucial to determine the most suitable type of API used. The need to apply document transformation is also a considerable factor for API selection, once memory-based APIs are much more suitable for this type of operation, while streaming-based APIs are more used for forward-only applications.

In order to test API performance in document transformations we considered *SalesOrderDetail* documents for the following APIs: SAX, StAX and VTD. Two operations were developed for each API:

- *Selection*: an operation that selects a set of elements based on a given predicate, representing forward-only access to data.
- *Difference*: an operation that removes from the first document all the elements that are in common with the second document, representing a random access to data.

A *selection* operation, based on a predicate, selects all elements where *SalesOrderID* has a value of 43,659, producing a new document. The *difference* operation checks if an element, immediately below the root node of a document R, exists in a document S thus disregarding it and keeping it only if he doesn't exist in document S. For the *difference* operation we considered *SalesOrderDetail* for both arguments in order to produce an empty document so we could extensively use the algorithm and disregard the size of the result document, since it will be *null*.

In memory-based APIs, documents are fully loaded into memory allowing access to the whole XML structure. In our tests the result is immediately written to disk without creating an in-memory structure. For streaming-based APIs, transformations are performed in a sequential way; i.e. as data is read from, changes are reflected in the outcome document. According to results (table 3 and table 4) we can see that StAX is the API that has the better performance, followed by VTD.

Table 3. Performance test in milliseconds for selection operation

Operation/API	StAX	SAX	VTD
SalesOrderDetail1	884 ms	895 ms	861 ms
SalesOrderDetail2	1900 ms	2494 ms	1971 ms
SalesOrderDetail3	3590 ms	5763 ms	4288 ms

Table 4. Performance test for the difference operation in minutes (m)

Operation/API	StAX	VTD
SalesOrderDetail0	4,1 m	1,5 m
SalesOrderDetail1	72,2 m	14,7 m

However, VTD consumes a considerable amount of memory. Memory consumption can be a bottleneck for environments that provide limited capabilities. We used a new document: *SalesOrderDetail0* with 2,9 megabytes in order to reduce execution time of the test. Considering the selection operation, StAX is slightly faster, with the advantage of lower memory consumption compared to VTD. This increase in memory usage occurs mainly due the cost of rebuilding the entire structure of document in memory, which also implies a higher execution time. Only after the correct representation of the document in memory the processing phase starts. Streaming-based APIs do not have this procedure, starting transformation immediately, obtaining results faster and with less computational resources.

For the difference operation, memory-based APIs are faster than streaming-based APIs. The difference operation requires that for each element of R, a verification process be done that uses multiple comparisons in order to verify if it exists in document S. With streaming-based APIs it is necessary to perform a large number of I/O (*input/output*) operations, because for every element of R it might parse the entire document S (at worst). In case of memory-based APIs, since both documents are fully represented in memory, the comparisons do not have to do any I/O thus reducing execution time. Due to memory limitations, if we need to work over several documents at the same time their size is even more

restricted since they all need to be in memory to be processed.

It was also found that the first run of the operations is slower than subsequent runs. Therefore, we conducted a study (table 5) for the selection operation with StAX and VTD with documents: *SalesOrderDetail1*, *SalesOrderDetail2* and *SalesOrderDetail3* in order to evaluate the impact of the first run. The first-run impact has more emphasis on VTD, and speed execution increases considerably as the size of documents increases, influencing runtime speed between StAX and VTD.

Table 5. Elapsed time in milliseconds (ms) for the selection operation first run

Operation/API	StAX	VTD
SalesOrderDetail1	1129 ms	1164 ms
SalesOrderDetail2	1917 ms	3052 ms
SalesOrderDetail3	3993 ms	6761 ms

6 MODIFYING PERFORMANCE

An important feature that appeared in the APIs analysis, was the ability to manipulate elements of an XML document, i.e., insert, delete or update information. Streaming-based APIs are not adequate to this kind of operations because they process documents in a sequential way, which complicates the implementation of the previous operations without apparent benefit since transformations are not performed by the order of elements presented in document. In this case, it would be necessary to perform multiple I/O operations.

For memory-based APIs, we tested DOM and VTD, mainly because almost all other APIs tested are based on the same model of DOM and the performance differential between them is not very relevant. VTD uses much less memory than DOM and performs document parsing in less time. The cost of object creation in DOM API is the main factor for the different performance. VTD is immune to this cost due to its inherent representation structure. However, tree structured manipulation for DOM allows a fairly trivial

manipulation of data, since adding or removing a node in the tree is done by a manipulation of pointers between nodes. On the other hand, VTD needs to rebuild *VTD records* for processing next update. We built a test scenario that changes the content of *AuctionWebSite* documents. The structure of such documents consists in the following elements: *regions*, *categories*, *catgraph*, *people*, *open_auctions*, and *closed_auctions*. Each of these elements contains a set of lines with information relating to an auction site. The tests change data on *persons* and consist of three steps:

- 1) adding an element *nationalidnumber* with *unknown* content;
- 2) renaming *creditcard* element for *cc*; and changing *gender* element content of each *person*,
- 3) replacing *male* for *M* and *female* for *F*.

In both APIs, documents are loaded into memory and scanned in order to scroll through the contents of each person, making the modifications at the same time. After performing all transformations, the document is written to a file using DOM *Transformer* class and VTD *XMLModifier* class respectively. For performance analysis we measured APIs with four smaller *AuctionWebSite* documents. Each document contains the following number of *persons*:

- *AuctionWebSite1* – 2550 persons
- *AuctionWebSite2* – 7649 persons
- *AuctionWebSite3* – 12750 persons
- *AuctionWebSite4* – 20400 persons

Table 6. Execution time in milliseconds of each API

Document/API	DOM	VTD
AuctionWebSite1	4203 ms	903 ms
AuctionWebSite2	6292 ms	1819 ms
AuctionWebSite3	10433 ms	3143 ms
AuctionWebSite4	14774 ms	4541 ms

In table 6 we can see the test results for each of the documents processed. Note that for large documents we had to increase the Java Virtual

Machine memory available in order to process them. Results show a clear superiority of VTD for data insertions and updates. For this scenario, object manipulation of DOM has no advantages in relation to the array of integers' structure used by VTD.

7 CONCLUSIONS

The use of structured documents in XML has a wide area of application in different types of fields. In many cases it is necessary to process documents of a considerable size, where runtime is relevant and the execution window is clearly limited. As we saw, there are two types of XML APIs: memory-based APIs and streaming-based APIs. Memory-based XML APIs maintain a long lived structural data in memory and only when the parsing process is finished modifications are allowed, while streaming-based APIs use small memory footprint, allocating and freeing memory constantly, allowing the process of infinite size XML documents (in theory).

Generally, for XML handling, dom4j, and DOM are good choices, with the preference between them determined by Java-specific features or cross-language compatibility, depending on project requirements. Although less flexible in XML transformations, OJXQI is an excellent choice when you need to do standard modifications with good performance. VTD array of integers' structure proves to be the best model in almost all tests. It is a model that consumes less memory (compared to other memory-based APIs), the processing time is very fast and even their ability to update a document, maintaining its structure in memory, proved being far superior in relation to the other memory-based APIs (for tested scenario). The use of VTD API is more complex in comparison to other memory-based APIs, where it is necessary an additional effort to dominate the API's features.

For streaming-based APIs, StAX has proved to be an API with better overall performance compared to SAX and XOM. This kind of APIs do not maintain long-lived structural data in memory, so there are no advantages in using this type of API

when you need to perform a set of transformations that somehow change the order of elements in the XML hierarchy. Typically, this type of APIs are used only for forward-only applications or simple modifications using XSLT language.

Memory-based APIs maintain the structure of the whole document in memory, resulting in some overhead, however, for updates that somehow change the document structure, this type of APIs lead to some advantages over the streaming-based APIs since those need to perform increased I/O operations to do same transformation. Manipulating a document using memory-based APIs is much more accessible and quick, since for streaming-based APIs we need to constantly use temporary buffers to keep information in memory. In summary, we can conclude that choosing from the two approaches studied for processing XML documents depends mostly on project's requirements.

8 REFERENCES

1. J. Zhang, "Simplify XML Processing with VTD-XML," JavaWorld.com (2006).
2. S. C. Haw and G. S. V. R. K. Rao, "A Comparative Study and Benchmarking on XML Parsers," in *Advanced Communication Technology, The 9th International Conference* (2007).
3. L. L. B. Zhao, "Performance Evaluation and Acceleration for XML Data Parsing," in *In 9th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)* (2006).
4. Y. Oren, "SAX Parser Benchmarks," <http://piccolo.sourceforge.net/bench.html> (2002).
5. R. Alnaqeib, F. H. Alshammari, M.A. Zaidan, A.A. Zaidan, B.B. Zaidan, and Z. M. Hazza, "An Overview: Extensible Markup Language Technology," vol. 2, no. 6, pp. 177–181, *Journal of computing* (2010).
6. V. Sengirova, A. Oralbekova, and N. Shah, "An Empirical Evaluation of BFS, and DFS Search Algorithms on J2ME Platform, and SVG Tiny Parsing on J2ME Platform Using SAX, StAX, and DOM Parsers," *International Journal on Advanced Science*, vol. 2, no. 5, pp. 65–70, *Engineering and Information Technology* (2012).
7. F. Wang, J. Li, and H. Homayounfar, "A space efficient XML DOM parser", vol. 60, no. 1, pp. 185–207, *Data & Knowledge Engineering* (2007).
8. T. C. Lam, J. J. Ding, and J.-C. Liu, "XML Document Parsing: Operational and Performance Characteristics", vol. 41, no. 9, pp. 30–37, *Computing & Processing* (2008).

9. B. McLaughlin and J. Edelson, *Java and XML*, 3rd Edition. O'Reilly Media (2006), p. 480.
10. W. Biggs and H. Evans, "Simplify XML programming with JDOM," <http://www.ibm.com/developerworks/java/library/j-jdom/> (2001).
11. D. Sosnoski, "XML and Java technologies: Document models, Part 1: Performance," <http://www.ibm.com/developerworks/xml/library/x-injava/> (2001).
12. D. Sosnoski, "XML and Java technologies: Java document model usage," <http://www.ibm.com/developerworks/xml/library/x-injava2/> (2002).
13. E. Harold, "XOM presentation," <http://www.xom.nu/whatswrong/whatswrong.html> (2003).
14. J. Zhang, "VTD-XML: XML Processing for the Future (Part I)," <http://www.codeproject.com/Articles/23516/VTD-XML-XML-Processing-for-the-Future-Part-I> (2008).
15. J. W. S. P. Team, "Streaming APIs for XML Parsers," http://java.sun.com/performance/reference/whitepapers/StAX-1_0.pdf (2005).