# Virtual DOM: an Efficient Virtual Memory Representation for Large XML Documents

Giuseppe Psaila

*Università degli Studi di Bergamo*
*Facoltà di Ingegneria*
*Viale Marconi 5, I-24044 Dalmine (BG), Italy*
psaila@unibg.it

***Abstract*— Standardized main memory representation methods (*DOM*) are quite inadequate to represent large XML Documents, both in terms of space (necessary amount of memory) and, consequently, in time: it easily happens that it is not possible to load large XML documents in main memory.**

**In this paper, we present *Virtual DOM*, a Java package that provides an efficient representation technique for large XML documents. It adopts a specifically designed *virtual memory technique*: memory blocks allocated to represent the document are swapped, when necessary, by skipping the operating system swapping mechanism; this way, the actual main memory needs are kept under control, the thrashing phenomenon is avoided even for large documents.**

## I. Introduction

Since its definition, XML has gained an important role in may application domains and information systems, both classical and web based. In fact, XML is nowadays used for representing any kind of document and information, for example data sets for data mining and the results of data mining tasks (see, for instance, PMML, the Predictive Modeling Mark-up Language [1]), metadata for data warehouse systems (such as the Common Warehouse Metamodel [2]), ontologies, etc.. In particular, several large data sets are collected and managed by means of very large XML documents (such as DBLP XML Records, stored in an XML document greater than 400 MBytes), or very large collections of XML Documents (like the INEX data set). However, standardized main memory representation methods (*DOM*) are quite inefficient in terms of space (necessary amount of memory): it easily happens that it is not possible to load so large XML documents in main memory, thus limiting the possibility of dealing with these huge documents.

In this paper, we present the results of the development of *Virtual DONM*, a Java package aimed at representing large XML documents in a DOM fashion.

Within the package, we developed a context-specific *virtual memory technique*: memory blocks allocated to represent the document are swapped, when necessary, by skipping the operating system swapping mechanism, cause of the thrashing phenomenon. This solution allows to keep under control the actual maim memory needs; consequently, the thrashing phenomenon is avoided even for large documents, because it is possible to adopt specifically designed swapping strategies.

However, the results presented in this paper are not the final result of our research. In our mind, *Virtual DOM* is going to be the basic brick for the development of a *Persistent Storage Service* for collections of XML documents.[1]

The paper is organized as follows. Section II introduces the structure of the *Virtual DOM* package, describing in details the technical solutions adopted to design and develop the package. Section III reports the results we obtained by making comparative experiments with the standard *DOM* implementation. Finally, Section IV draws the conclusions.

**Related Work.** XML is now widely adopted within (networked) applications, and in the context of databases as well [3]. In effect, database vendors are extending their DBMS engines with a set of functionality that directly support XML. However, the exchange of XML fragments is made through standard channels (ODBC or JDBC): XML fragments are serialized and their string representation is parsed by the application (see, for a concise discussion, [4]). Even when a native XML DBMS is used (for instance, the Tamino DBMS [5]), usually *SAX* APIs or *DOM* APIS are used to exchange XML fragments between DBMS and application. Consequently, in case of large documents to exchange, these environments present the same problem addressed by this paper.

The work presented in this paper is also relevant for another research line, i.e., the extension of the Java language with native support for the type management management of XML documents. The main proposal in this sense is the *XJ* language [6], i.e., the Java language extended with new suitable constructs.

## II. The Virtual DOM Package

### A. Data Structure Architecture

The architecture of the data structure is shown in Figure 1. In our architecture, there are several sets of homogeneous elements. This sets are organized as *virtual vectors*; each element in the vector is identified by its position.

[1]The work was supported by the PRIN 2006 program of the Italian Ministry of Research, within project ‘ ‘Basi di dati crittografate” (2006099978).

1529-4188/08 $25.00 © 2008 IEEE
DOI 10.1109/DEXA.2008.117

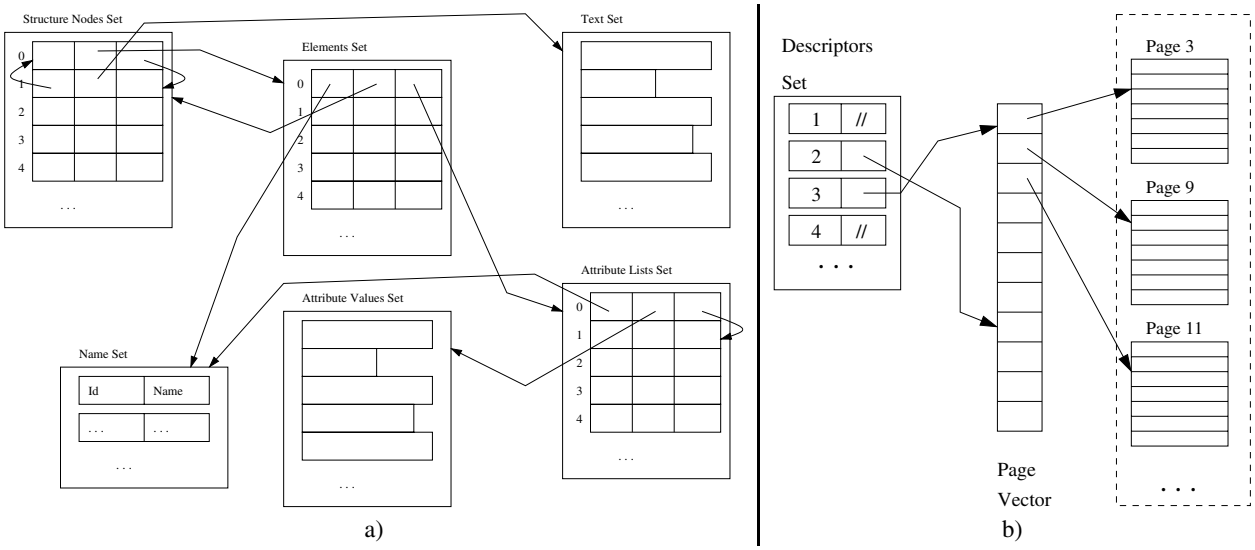


Fig. 1. a) Data Structure Architecture. b) Structure of Pagers

*a) Nodes:* The main XML concept is the *Element*. An element has a name and a set of attributes. The occurrence of an element is contained within a *start tag* and an *end tag*; If not empty, the content of an element occurrence is a mixed sequence of element occurrences and *text* occurrences.

Being inspired by *DOM* terminology, we call both element occurrences and text occurrences *structure nodes* or, for simplicity, *nodes* (the *DOM* concept of nodes is wider, and encompasses attributes, entities, comments, etc.). A text (node) is simply a string that appears in the content of an element node.

In our architecture, each node is described by a row in the *Structure Nodes Set* (that is a virtual vector); the row number plays the role of *node identifier*. Each element in the *Structure Nodes Set* is a triple with the following structure.

| Parent | Reference | NextBrother |
|---|---|---|

The field *Parent* is the identifier of the parent node the contains the node described by the row. Field *Reference* is the reference to the actual node type: if its value is greater than or equal to 0, this means that the node is an element and the value is the identifier of the element descriptor described in the *Elements Set* virtual vector; if its value is less than 0, this means that the node is a text node and the absolute value of the field is the offset of the text string in the *Text Set* virtual vector. Field *NextBrother* is the identifier of the next node in the parent's content; if this field is set to $-1$, this means that the current node is the last node in the parent's content.

*b) Elements Set:* When a node is an element node, the corresponding row in *Structural Nodes Set* refers to an element descriptor in the virtual vector named *Elements Set*. Each row in the virtual vector has the following structure.

| Name | FirstChild | FirstAttrbute |
|---|---|---|

Field *Names* is the identifier of the element's name, described by *Names Set*, a container of names. Field *FirstChild* is the identifier of the first node in the element content; if it is set to $-1$, this means that the content is empty. Field *FirstAttribute* is the identifier of the first attribute occurrence; if it is set to $-1$, this means that the node has no attribute.

*c) Names Set:* An important reason why *DOM* representation wastes memory space (up to 5 times than the size of the document, see [3]) is the fact that names (element names and attribute names) are replicated several times, once for each occurrence of tags and attributes. Furthermore, in Java implementations they are instance of the class *Object*, that contains specific data structures to deal with synchronization and multi-threading.

In order to limit this phenomenon, in the *Virtual DOM* package all names are represented by a specific data structure, called *Names Set*. Each name is associated a numerical identifier, and names are referenced by means of this identifier.

The data structure provides a fast access to names, both by means of their identifier and by means of their name (so that it is possible to efficiently find previously defined names).

*d) Text:* A specific virtual vector, named *Text Set*, collects text appearing in the content of tags. *Text Set* is able to store only text. However, since text items are nodes, they are uniquely identified by the position of the row in *Structural Nodes Set* that describe the node and refers (field *Reference*) to the actual position of the text value in the *Text Set*.

*e) Attribute Lists:* Element nodes may have a set of attributes as well. Although the order of attributes within a tag is not relevant, nevertheless they form a list. A specific data structure called *Attribute Lists Set* stores attribute lists. Each attribute occurrence is identified by its position in the *Attribute Lists Set*. The structure of a row is the following.

| Name | Value | NextAttribute |
|---|---|---|

Field *Name* is the identifier of the attribute name; it refers to the *Names Set*. Field *Value* is the offset of the string value in

the *Attribute Values Set*. Field *NextAttribute* is the identifier of the next attribute occurrence belonging to the same element node; if it is set to $-1$, this is the last attribute occurrence of the element node.

The attribute value is stored in a specific virtual vector, named *Attribute Values Set* (see later).
If an element node has an associated attribute list, the descriptor in the *Elements Set* refers to the first element of the attribute list.

*f) Attribute Values Set:* Attribute values are strings of variable length, depending of the actual length of each attribute value occurrence; then, they are very similar to *Text*. Consequently, they are stored in a specific virtual vector, named *Attribute values Set*, that behaves as the *Text Set*.
The row describing an attribute occurrence in an attribute list (see *Attribute Lists Set*), refers to its value occurrence by means of its offset in the virtual vector.

### B. Pagers

A *Pager* is a key component of our architecture. Its role is to represent *virtual vectors*, i.e., very large sets of homogeneous XML concepts, in an efficient way. It is the responsibility of pagers to implement the virtual memory mechanism. Figure 1 shows the internal structure of a pager.

A pager is based on the concept of logical page: the whole set of homogeneous information is subdivided in partitions named *pages*; all pages in the same pager have the same size. A pager contains a number of preallocated *physical pages* (denoted in the figure within the dashed rectangle), and the vector of pointers named *page vector* points to each physical page. Since we rely on the concept of virtual memory, the number of logical pages might be greater than the physical ones: a swapping mechanism saves pages to disk, and reload them when necessary.

A descriptor is associated with each logical page. It is necessary to deal with logical pages not present in any physical page. Descriptors are collected in the *Descriptors Set*, implemented by means of a hash table, for fast access. A descriptor describes the logical number of the page, and the position of the physical page in the *page vector*: the non valid value `-1` denotes that the page is stored in the disk. Furthermore, it also describes the number of *disk page*, i.e., the offset in the swap file where the page has been saved for the first time (if set to $-1$, the pages has not been saved yet).

At the moment, we adopt two different strategies for choosing the page to swap. For pagers except the one representing structural nodes, the page less recently used is swapped to disk. For *Structural Node Set*, we adopt an integrated strategy that includes knowledge about tree navigation (Section II-D). *Text and Attribute Values* pagers store textual content and attribute values. The pager implements a virtual continuous vector of bytes; each text is terminated by an ASCII code 0 character, and identified by its offset in the virtual vector.
*Structural Nodes, Element Occurrences and Attribute Lists* are stored in pagers named *Structural Nodes Set*, *Elements Set* and *Attribute Lists Set*, respectively. As previously described, this is a virtual vectors of rows, where each row is composed of several fields. The pager always implements a continuous vector of bytes, but shows a row organization; thus, it allows to directly refer to a row by means of its absolute position.

### C. Cursors

In order to allow programs to navigate the document, the *Virtual DOM* package provides the concept of *Cursor*; a cursor provides methods to visit the tree structure, by moving up, down and to the side of the currently visited node; furthermore, it provides methods to jump to a specific node. Nevertheless, a cursor provides methods to get element names, attribute names, attribute values and texts.

The *Virtual DOM* package can deal with several active cursors at the same time: this allows programs to navigate the document tree in a complex way, by nesting recursive scans of the tree.

### D. Choosing the Page to Swap

For pages in the *Structural Nodes Set*, we implemented a specific strategy to choose the page to save to disk. When loading the document and when navigating the tree (by means of cursors) stacks are used to store the paths of visited nodes; we call these stacks *Navigation Stack*.
The strategy to choose the page to swap is the following: pages containing nodes in at least one *Navigation Stack* should have a low priority; the greater the distance from the root, the lower their priority. The other pages have higher priority than the previous ones, and they are assigned a priority based on the last time they were accessed, so that the page less recently used is the preferred.

Figure 2 shows the pseudo-code of the algorithm that implements the strategy.

Lines 1. to 4. denotes the input data structures of the algorithm. *PAGES* is the number of physical pages actually present in main memory.
*NodePages* is the hash table containing page descriptors; a page descriptor (as far as the algorithm is concerned) contains the page identifier, the number of physical page (set to $-1$ if the pages is not in main memory) and the timestamp of the last access to the page.
*Stacks* is a set of *Navigation Stacks*; each one is denoted as *NStack* and contains node identifiers. There is one stack for each active cursor, and one stack to deal with the loading phase.
*Weight* is an array of integers, that the algorithm uses to evaluate the weight of each page in main memory.

The loop at line 5. initializes the *Weight* array.

The nested loops beginning at lines 6 and 7. explore each *Navigation Stack* in *Stacks*, in order to know what pages must have low priority. For each node identifier in a navigation stack (line 8.), the number of the logical page that contains the corresponding descriptor is computed (line 9.) and from the set of page descriptors (*NodePages*) the number of physical page that contains the requested logical page is obtained (by means of method *FindPage*); the number of physical page is

**Algorithm ChoosePage**

1. *PAGES*: number of physical pages
2. *NodePages*: HashTable of (*id*, *physicalPage*, *lastAccess*)
3. *Stacks*: set of *NStack*, Stack of (*nodeId*)
4. *Weight*: Array of *Integer*, with *PAGES* elements
5. **for** $i := 1$ **to** $PAGES$ **do**
   **begin**
   *Weight*[$i$] := 0;
   **end**
6. **for each** *NStack* **in** *Stacks* **do**
   **begin**
7. **for** $i := 1$ **to** |*NStack*| **do**
   **begin**
8. *NodeId* := *NStack*[$i$].*nodeId*;
9. *PageId* := *NodeId* / *PAGE_DIM* + 1;
10. *position* := *NodePages*.*FindPages*(*PageId*);
11. **if** *position* > 0 **then**
12. *Weight*[*position*] := $i$;
    **end if**
   **end**
   **end**
13. *MaxTime* := *NodePages*.*MaxTime()*;
14. **for** $i := 1$ **to** |*NodePages*| **do**
   **begin**
15. *position* := *NodePages*[$i$].*physicalPage*;
16. **if** *position* > 0 **then**
17. **if** *Weight*[*position*] = 0 **then**
18. *Weight*[*position*] :=
        0 − (*MaxTime* − *NodePages*[$i$].*lastAccess*);
    **end if**
   **end if**
   **end**
19. *PageToSwap* :=
    *PositionWithMinimumValue*(*Weight*);

**End Algorithm**

Fig. 2. The Algorithm *ChoosePage*

stored in the variable *position* (line 10.).
If the physical page is in main memory (line 11.), i.e., the value of *position* is positive, then the corresponding element in array *Weight* is assigned the value of the loop counter $i$ (line 12., that denotes the distance from the root of the document. At the end of the loop, pages in main memory containing the descriptors of the traversed nodes have a positive weight.

Line 13. computes the maximum value of timestamps denoting the last access to a page.
This value (assigned to variable *MaxTime* is used to compute the weight for pages not considered in the previous loop.

The loop beginning at line 14. explores again the stack of page descriptors.
For each page, it takes the number of physical page, and assigns it to variable *position* (line 15.).
If the value of position is greater than 0 (line 16.), this means that the page is currently in main memory. Thus (line 17), if the page has not been considered by the loop at line 7., it is assigned a negative weight, such that pages are still ordered based on the value of time-stamp denoting the last access time.

At this point, since pages currently in main memory are assigned weights in such a way the highest values denotes the lowest priorities for swapping, line 18. looks for the position in the *Weight* array containing the minimum value; this value, assigned to variable *PageToSwap*, is the number of physical page that contains the page to swap to disk.

| *Exp.* | *Elements* | *Size (Bytes)* |
|---|---|---|
| D1 | 1,000,000 | 44,170,838 |
| D2 | 3,000,000 | 132,237,606 |
| D3 | 5,000,000 | 220,322,509 |
| D4 | 7,000,000 | 304,122,615 |
| D5 | 9,000,000 | 384,270,837 |

Fig. 3. Characteristics of Large Documents

## III. Experiments

In order to test performance[2], we prepared several data sets. Since we wanted to have a uniform structure for all data sets, we generated several subsets of the DBLP XML Records [7], by taking the first $n$ elements from the beginning. We performed experiments on large documents (greater than 1000000 elements).

Pagers in the *Virtual DOM* package were set as follows: 400 pages with $5,000$ rows ($60,000$ Bytes) for *Structural Nodes Set*; 200 pages with $5,000$ rows ($60,000$ Bytes) for *Elements Set*; 200 pages with $1,000$ rows ($12,000$ Bytes) for *Attribute Lists Set*; 400 pages with $100,000$ Bytes for *Text Set* and *Attribute Values Set*. With these settings, the amount of memory allocated by the process is limited to 248 MBytes.

Table in Figure 4 shows the results of experiments: in particular, the column *DOM* reports the execution times for loading the document to main memory by a standard *DOM* library (we used the library provided together with the Xerces Java Parser see [8]); column *VDOM (Load)* reports the execution rimes for loading the document with the *Virtual DOM* package; column *VDOM (Scan)* reports the execution times needed to scan the document, once loaded by *Virtual DOM*. Figure 5 graphically depicts the results (the solid line corresponds to standard *DOM*, the dotted line corresponds to loading time for *Virtual DOM*, while the dashed line corresponds to the sum of loading and scanning times obtained with *Virtual DOM*).

We start by comparing *DOM* and *Virtual DOM* in the loading phase. The reader can see that execution times shown by *DOM* are around 3 times faster for the document with $1,000,000$ elements: this ratio is constant when the number of allocated pages is sufficient to represent the whole document

[2]Experiments were conducted on a PC powered by an Intel Pentium 4 641 3.2 GHZ processor, equipped with 1 GByte RAM (of type DDR2 PC2-4200 SYNCH DRAM NON-ECC), a 256 GByte Hard Disk (Serial ATA II). The installed operating system is Linux Fedora 6 Core Distribution (kernel version 2.6.20-1.2952.fc6). Java classes were compiled with JDK version 1.6.0_03. Classes were executed using the Java Runtime Environment JRE1.6.0_03.

| *Exp.* | *DOM* | *VDOM* (Load) | *VDOM* (Scan) |
|---|---|---|---|
| D1 | 3.726” | 12.144” | 11.070” |
| D2 | 9.423” | 57.723” | 43.166” |
| D3 | 1’33.959” | 1’52.003” | 58.616” |
| D4 | hours | 3’11.424” | 1’29.385” |
| D5 | hours | 4’55.650” | 2’58.179’ |

Fig. 4. Experiments with Large Documents, with Different Settings

(it is due to the complexity of our data structure, but we think it is possible to change some auxiliary structure, to obtain better performance closer to *DOM* performance); in fact, with document $D1$ only a very limited number of pages are swapped to disk by *Virtual DOM*.
When working on the document with $3,000,000$ elements, *DOM* is $6$ times faster: in fact, *Virtual DOM* starts swapping pages, while the amount of memory allocated by *DOM* is still contained in main memory; however, *DOM* uses more than $450$ MBytes, while *Virtual DOM* always uses $248$ MBytes.
With the document containing $5,000,000$ elements, *DOM* allocates more than $800$ MBytes; consequently, the operating system is forced to swap memory blocks, and the phenomenon of thrashing starts having some effects. This is shown by the fact that the two execution times are now comparable.
With the document containing $7,000,000$ elements, *DOM* needs around $1.5$ GBytes, and its execution time grows dramatically (in our system, several hours were necessary to finish). With the document containing $9,000,000$ elements, the situation is worst, since *DOM* needs about $1.9$ GByte memory; thrashing makes impossible to use *DOM*.

Conversely, *Virtual DOM* is able to deal with the largest documents we considered in our experiments, finishing in around $3$ minutes when it reads the $7,000,000$ elements document, and finishing in about $4.5$ minutes when it reads the $9,000,000$ elements document: this is due to the virtual memory mechanism implemented in the *Virtual DOM* package, that skips the virtual memory mechanism provided by the operating system; thus it is able to deal with such large documents. It is notable that this result is achieved by allocating only $280$ MBytes of main memory.

Furthermore, we can see that *Virtual DOM* shows execution times that substantially grows linearly in the reading phase. This is due to the fact that the cost of writing virtual pages on disk grows with the size of the input document.

To complete our experiments, we evaluated the scanning time of the loaded document. A cursor is used to explore the document tree. The right hand column of Table in Figure 4 shows the scanning times, while the dashed line in Figure 5 shows the sum of loading and scanning times. The reader can see that the scanning time grows progressively, but never becomes greater than the loading time.

Thus, the main result, in this sense, is that *Virtual DOM* is able to manage large and very large documents, both for reading and for scanning. We think that its performance can be further improved; in fact, we are aware about several sources of inefficiency that still remain in the code of *Virtual DOM*, on which we are working at the moment the paper is written.

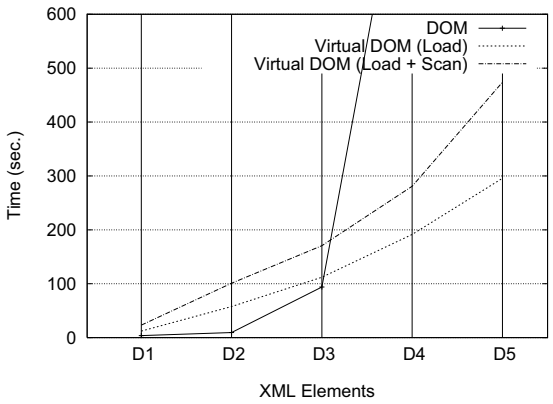


Fig. 5. Comparison for Large Documents

## IV. Conclusions

In this paper, we presented the *Virtual DOM* package, designed to manage large XML documents in a *DOM* fashion, overcoming the problem of representing large documents in main memory. We presented the architecture of the package, by showing the technical solutions we adopted. Finally, we performed comparative experiments, where we compared *Virtual DOM* with a standard *DOM* package.

Experiments show that the approach followed to design *Virtual DOM* is effective for large and very large documents, because it avoids the thrashing phenomenon caused by the virtual memory mechanism provided by the operating system, not optimized for the specific problem.

The long term perspective of this work is the use of *Virtual DOM* to realize *Persistent Collections of XML Documents*. This is our final goal, that we can achieve by passing through several steps: the first one is the implementation of an *XPath* interpreter, together with the development of suitable functionality to make persistent the *Virtual DOM* data structure image and reload it successively.

## References


[1] *Predictive Modeling Mark-up Language (PMML)*. http://www.dmg.org/: DMG Data Mining Group.

[2] *Common warehouse metamodel (CWM) specification*. http://www.omg.org/cwm: OMG Object Management Group.

[3] M. Nicola and J. John, “XML parsing: a threat to database performance,” in *the 2003 ACM CIKM International Conference on Information and Knowledge Management*, New Orleans, Louisiana (USA), 2003, pp. 175–178.

[4] H. Schuhart and V. Linnemann, “Valid updates for persistent XML objects,” in *Business, Technologie und Web BTW-2005*, Karlsruhe (Germany), March 2005.

[5] *Tamino XML Database*. http://www.softwareag.com/tamino: Software AG.

[6] M. Harren, M. Raghavachari, O. Shmueli, M. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar, “Xj : Facilitating XML processing in java,” in *the 14th World Wide Web Conference WWW-2005*, Chiba (Japan), 2005.

[7] *Database and Logic Programming Bibliography: XML Records*. http://dblp.uni-trier.de/xml/: University of Trier - Germany.

[8] *XERCES 2 Java XML Parser*. http://xerces.apache.org/xerces2-j/: The Apache Software Foundation.