DOM tree browsing of a very large XML document: Design and implementation

Seung Min Kim, Suk I. Yoo*

School of Computer Science and Engineering, Seoul National University, Seoul 151-742, Republic of Korea

A R T I C L E   I N F O

Article history:
Received 3 July 2008
Received in revised form 8 May 2009
Accepted 20 May 2009
Available online 2 June 2009

Keywords:
XML
DOM
DOM tree
Very large XML documents

A B S T R A C T

Browsing the DOM tree of an XML document is an act of following the links among the nodes of the DOM tree to find some desired nodes without any knowledge for search. When the structure of the XML document is not known to a user, browsing is the basic operation performed for referring the contents of the XML document. If the size of the XML document is very large, however, using a general-purpose XML parser for browsing the DOM tree of the XML document to access arbitrary node may suffer from the lack of memory space for constructing the large DOM tree. To alleviate this problem, we suggest a method to browse the DOM tree of a very large XML document by splitting the XML document into \( n \) small XML documents and generating sequentially the DOM tree of each of those small \( n \) XML documents. For later reference, the information of some nodes accessed from the DOM tree already generated has been also kept using the concept of their virtual nodes. With our suggested approach, the memory space necessary for browsing the DOM tree of a very large XML document is reduced such that it can be managed by a personal computer.

© 2009 Elsevier Inc. All rights reserved.

1. Introduction

Extensible Markup Language (XML) and its related technologies are used as the standard for representing and exchanging information on Web environments. In spite of the parsing overhead and the requirement for a considerably large storage space caused by the XML representation, many XML documents have been generated in various industry fields due to their flexibility in information modeling. The size of some XML documents which can be accessed easily on the Internet or any other information networks may not be very large. However, due to the high computing speeds and the high bandwidth of information networks, nowadays, a number of large XML documents with their size ranging from several hundreds of megabytes to several gigabytes have been generated and used. For instance, the size of XML files containing protein sequence data (UniProt) ranges from 1.8 GB to 16 GB.

An XML document contains the structured information. In an XML document, each individual datum is stored as the value of an element and the element begins from its start tag and ends with its end-tag. Between the two tags, there can be a number of elements generated recursively. Therefore, the basic operation for accessing the contents of an XML document is the retrieval of the values of the elements using the structure in which they are stored.

Document Object Model (DOM) is the standard method to model an XML document as a tree structure: each element of the XML document is mapped onto a node of a tree structure where the name of the node is the tag name of the corresponding start- and end-tag, and the containment relationship between two elements is mapped onto an edge between two nodes in which one is contained by the other. By following the links among the nodes of the tree structure, we can access the contents of an XML document. In this paper, we call this tree structure the DOM tree of the XML document and an act of following the links among the nodes of the DOM tree browsing.

Two approaches to constructing the DOM tree of an XML document have been introduced: one to constructing it in main memory and the other to constructing it in secondary storage. The approach to constructing the DOM tree in secondary storage has advantage that it is independent of the size of the XML document but disadvantage that it has the long computation time to find the location of the desired node in the DOM tree and also the longer access time to that node as compared to the one using the DOM tree constructed in main memory.

In this paper we present the method to browse the DOM tree of the very large XML document constructed in main memory. A general-purpose XML parser such as Apache Xerces constructs the DOM tree of an XML document in main memory as it keeps requesting for memory allocation until the construction of the DOM tree is completed: whenever the XML parser finds an element, it requests memory allocation for storing the element in the memory. Thus it suffers from a lack of memory space for constructing the DOM tree of an XML document when the size of the XML document becomes very large.

The approach presented in this paper is the extended version of our previous work in Kim et al. (2007) for browsing the DOM tree...
of a very large XML document. In the previous approach, we split a very large XML document \( D \) into \( n \) small XML documents \( \langle D_1, D_2, \ldots, D_n \rangle \), retrieved data from the DOM tree constructed from each of these \( n \) small XML documents, and combined the results from all \( n \) DOM trees to construct one unified DOM tree. The desired data is then retrieved from the unified DOM tree constructed in main memory. With this approach, however, accessing arbitrary data not contained in the unified tree is not easy. To access any arbitrary data for browsing, the new approach is presented which is based on not the explicitly constructed unified tree but the virtual nodes and their associated operations.

2. Related work

Noga et al. (2002) and Lu et al. (2006) suggested the approaches to constructing the DOM trees of the XML documents in main memory: Noga et al. (2002) proposed a lazy XML processing approach consisting of the preprocessing phase and the progressive parsing phase. In the preprocessing phase, an internal representation of each node of the DOM tree is created by analyzing the structure of the XML document. In the progressive parsing phase, the internal representation of each node is transformed into a physical node of the DOM tree when it is accessed for the first time. This may be efficient in that the outline of the DOM tree structure is constructed rapidly and its details is taken only when it is required. However, it becomes inefficient in the application accessing most of the nodes in the DOM tree such as tree navigation from the root to all leaf nodes. Further, even rapid construction of the tree outline may not be feasible when the size of the document is so large that the memory required for the internal representation exceeds the allowed free memory space. Lu et al. (2006) designed and implemented parallel XML parsing on a shared-memory computer. Their method consists of the pre-parsing phase and the parallel parsing phase. In the pre-parsing phase, a simplified tree structure of the XML document is constructed. This tree contains the logical structure of the XML document and the range information of each element in the XML document. The XML document is partitioned on the basis of the logical structure in order to parse each chunk in parallel. In the parallel parsing phase, each chunk from the partitioned XML document is allocated to a thread either statically or dynamically. All subtrees, which are generated from the chunks, are combined to generate the main tree structure when the parallel parsing phase is completed. This is an efficient parallel processing technique for constructing the tree rapidly. However, it also can suffer from a memory problem when the size of the document becomes very large since the required memory for internal representation may exceed the allowed free memory space.

Koch (2003), Zhang et al. (2004), and Wong et al. (2007) suggested the approaches to constructing the DOM trees of the XML documents in secondary storage. Koch (2003) proposed a storage model for constructing the XML tree of an XML document in secondary storage by separating the tree structure and its data into two files, named as .arb and .lab. The file .arb contains the structural information of the XML tree as a binary tree, and the file .lab contains the label names of each node in the XML tree. Each node \( e \) of the XML tree is stored as a fixed-size field of \( k \) bytes on disk: the two highest bits of \( k \) denote whether \( e \) has a first and/ or a second child and the remaining \( 8 \)’s \( - 2 \) bits are used to hold an integer denoting the label of \( e \), which is saved in the .lab file. In order to navigate adjacent nodes from a node, they scan the contents of the .arb file as a stack of size \( O \) (the depth of the XML tree) is used for calculating the ancestor-descendant relationship between nodes, and they also access .lab file in order to generate any node \( e \) of the XML tree. The main advantage of this method is that the main memory space for generating these two files is proportional to the depth of the XML tree (not to the size of the XML tree). However, this method needs two file scanning passes to generate the .arb and the .lab files, and the cost of calculating the parent-child relationships between nodes on a binary tree can be a significant performance bottleneck particularly when there are many nodes in the XML tree. Zhang et al. (2004) and Wong et al. (2007) proposed physical storage schemes, which also separate the structural information and the value information, in one sequential scan on an XML data. For storing the structural information of an XML data, Zhang et al. (2004) used the balanced parentheses encoding method (Knuth, 1997) by removing all open parentheses. In order to reconnect the two parts of information, they used Dewey ID (Guo et al., 2003) as the key of tree nodes. Wong et al. (2007) used the balanced parentheses encoding method too, but they claimed that all navigational operations and updating operations can be finished in near constant time on compressed XML data, which is compared to the linear time of Zhang et al. (2004). Native XML DBMSs, such as Apache Xindice and eXist store and index compressed XML documents into secondary storage by constructing performance-oriented storage structures for storing and querying the XML documents also. The main purpose of these native XML storage schemes is to organize XML data into secondary storage for rapid query processing using XPath or XQuery on XML data. Therefore, the all navigational operations on the DOM tree of an XML data proceed after the separation of the tree structure and its data is done completely. This strategy of focusing on rapid query processing by constructing a traversable data structure is efficient on frequently accessed XML data. However, the processing time for constructing this traversable data structure may be overload when (1) the size of an XML data is very large and (2) there are also no guarantees of repeated accessing the XML data.

Finally, Balmin et al. (2005) and Beyer et al. (2005) suggested the approaches to storing and querying XML data using relational databases. Balmin et al. (2005) provided a framework that organized and formalized a wide spectrum of XML schema-driven denormalized decompositions of the XML data into relational databases. In order to eliminate many join operations for enhancing query performance, they generated non-normal form tables from XML schema and BLOB data type was used to store pre-parsed XML data. Beyer et al. (2005) proposed a system architecture which stored XML data as a new dedicated XML data type: when an XML data is stored into a column typed XML data type, the XML data is constructed into a binary representation of a structured and type-annotated tree from the XML data. For indexing and query evaluation, each node of the tree structure is given a unique identifier. Commercial DBMSs, such as Oracle Database and Microsoft SQL Server, also support their own XML data types. These database-based approaches may be preferable if the associated relational database is given and the structure of XML data is well analyzed so that we know exactly which parts of the XML data are referred frequently. However, in the usual personal computing environment with no specific storage solutions but the default file system provided by the operating system, this approach is not easy to be applicable.

3. Browsing the DOM tree of a very large XML document

In this section, we first provide the partitioning and padding algorithms, followed by a representation scheme to keep the information generated during the execution of the partitioning and padding algorithms. We then present the concept of virtual nodes of some nodes already accessed from the associated small DOM tree, which includes the construction of virtual nodes having the information of the already accessed nodes, the operations on virtual nodes for browsing, and the storage management of virtual nodes.
3.1. Partitioning

The contents of any document, including XML documents, are considered to be a sequence of characters. In this paper, we define a partition of a document as follows.

**Definition.** Let \( D \) be a document given by a sequence of characters. If each of \( n \) documents, \( F_1, F_2, \ldots, F_n \), is a subsequence of \( D \) such that the concatenation of \( F_1, F_2, \ldots, F_n \) is equal to \( D \), where \( n \) is a predefined number by user, then a sequence of the \( n \) documents, \( \{F_1, F_2, \ldots, F_n\} \), is a partition of a document \( D \) where each \( F_i, i = 1, \ldots, n \), is called a fragment of \( D \).

In explaining how to partition the given XML document, we use some notations defined as follows:

1. The size of a document \( D \) is expressed as \( D.length \).
2. The \( i \)th character of a document \( D \) is expressed as \( D[i] \) (\( 1 \leq i \leq D.length \)).
3. The sequence of characters from \( ith \) character to \( jth \) character of a document \( D \) is expressed as \( D[i..j] \). Therefore, the contents of the document \( D \) are expressed as \( D \{1..D.length\} \).
4. The document generated by concatenating two documents \( D_1 \) and \( D_2 \) is expressed as \( D_1 \cdot D_2 \).

The crux of partitioning a document lay in the way to find the boundary between fragments, called the cut point. Let \( D \) be a given XML document and \( n \) be the expected number of fragments in the partition of \( D \). First, the \( \lceil D.length/n \rceil \) characters are scanned from the beginning of \( D \), and around the location right after these characters, the range of width, \( h \), is formed as shown in Fig. 1.

Next, we compute the length of the path from the root element of \( D \) to each of the elements whose start-tags are contained in this range. For computing the path without building the DOM tree of \( D \) in memory, we use a stack containing strings. This stack keeps the list of element names from the beginning of \( D \) to the current scan position: when a start-tag is found, we push the element name into this stack, and when an end-tag is found, we pop the latest pushed element name from the stack. Since an XML document has the well-formed structure of element definitions, the size of this stack is the length of the path from the root element of \( D \) to the current element being processed.

The first cut point is then selected to be the location right after the start-tag of the element that has the shortest path from the root, called the cut element. If there are no start-tags in this range, the cut point is selected to be the location right after a start-tag, which is retrieved first in the scanning of \( D \) after the range. For the second cut point, the range of width, \( h \), is again formed around the location right after \( \lceil D.length/n \rceil \) characters from the first cut point, and the length of the path from the root to each of those elements whose start-tags are in this range is computed. Like the selection of the first cut point, the second cut point is then selected again to be the location right after the start-tag of the second cut element, the element having the shortest path from the root. This process is repeated until the last cut point is set up.

We adopted this policy of deciding the cut points based on three factors observed from the data structure of and operations on a very large XML document \( D \): first, \( D \) is in general an aggregation of small XML documents \( X_i, i = 1, \ldots, k \), each of which has a common structure with the almost same size. Secondly, the data retrieval operations on \( D \) are applied to each \( X_i \) independently in most cases. Thirdly, each \( X_i \) is directly or very closely connected to the root element of \( D \). Therefore, it is desirable to have cut points not split each \( X_i \) into two or more fragments. To meet this condition, each cut point is selected so as to locate as near to the root element as possible in a given range and the range is defined by the external variable, \( h \). If the value of \( h \) is defined to be large enough to contain more than one \( X_i \), each fragment can be easily generated with a set of the associated \( X_i \)’s.

It is noted, however, that the actual number of fragments generated by this process may become less than \( n \), the expected number of fragments, due to its policy of deciding the cut points. For example, if the total number of start-tags of \( D \) is less than \( n \), or the positions of start-tags of \( D \) lean severely toward some positions of \( D \), then the actual number of fragments generated by this partitioning algorithm becomes less than \( n \).

In spite of the drawback described above, we selected this policy of deciding the cut points because we concluded that, in seeking to find the benefits of using small XML documents, dividing the contents of an XML document into almost the same-sized fragments without breaking the aggregation pattern is more important than just guaranteeing the number of fragments to be generated.

The partitioning process explained above is more formally described as follows.

**Algorithm** \( \text{PARTITION}(D, n, h) \)

**[input]**
1. \( D \): An XML document to be partitioned
2. \( n \): The expected number of fragments to be generated
3. \( h \): The width of a range for finding a cut point

**[output]**
1. \( \{F_1, F_2, \ldots, F_n\} \): A partition of \( D \) (\( 1 < n < n \))
2. \( \{C_1, C_2, \ldots, C_{n-1}\} \): A sequence of cut points
3. \( p \): The prolog in \( D \)

```
declare starttags as an empty stack of strings;
declare t_i, t_{new} as pointing to a start-tag or end-tag each;
targetsize = \lceil D.length/n \rceil;
p — the prolog in \( D \);
startpoint — the offset of the end of prolog in \( D \) + 1;
cutpoint — startpoint;
i = 1;
while (((startpoint < D.length) and (i < n)) do do
  initcutpoint — startpoint + targetsize;
  t_{new} — next start-tag;
  cutpoint — cutpoint + targetsize;
  t_i — last end-tag of \( \text{partition}_i \);
end while
```
3.2. Padding

Since each fragment generated in the partitioning step does not constitute a well-formed XML document, it is modified into a well-formed XML document in the padding step.

All the fragments have three characteristics, which are given by the algorithm PARTITION \((D, n, h)\). First, every fragment, excluding the last one, ends with the start-tag of a cut element. Second, every fragment, excluding the first one in this time, starts without the tag of a cut element. Finally, there are no elements that can play the role of the root element in each fragment. Therefore, in this step, we make well-formed XML documents from fragments by padding missing tags to both the beginning and end of the fragments.

Before describing our padding algorithm, we define front and back pads as follows.

**Definition.** Let \(e_1\) and \(e_n\) be two nodes of n DOM tree where \(e_1\) is an ancestor of \(e_n\). If the sequence of nodes on the path from \(e_1\) to \(e_n\) is given by \(e_1e_2\ldots e_{(n-1)}e_n\), then

1. The **front pad** from \(e_1\) to \(e_n\) is a string given by
   \[
   \langle e_1 \rangle \langle e_2 \rangle \ldots \langle e_{(n-1)} \rangle
   \]
2. The **back pad** from \(e_1\) to \(e_n\) is a string given by
   \[
   \langle e_{(n-1)} \rangle \ldots \langle e_2 \rangle \langle e_1 \rangle
   \]

Some other notations used for explaining the padding algorithm are defined as follows:

1. The cut element, split at the position of \(C_i\), among two fragments, is expressed as \(CutElement(C_i)\).
2. The first start-tag of a fragment \(F_i\) is expressed as \(F_{i.first}\).
3. The last end-tag of a fragment \(F_i\) is expressed as \(F_{i.last}\).
4. The root element of an XML document \(D\) is expressed as \(D.root\).
5. The front pad from \(e_1\) to \(e_n\) is expressed as \(FPad(e_1, e_n)\).
6. The back pad from \(e_1\) to \(e_n\) is expressed as \(BPad(e_1, e_n)\).

Let \(D\) be a given very large XML document and \(F_1, \ldots, F_n\) be \(n\) fragments generated by the partitioning step as shown in Fig. 2.

To make each fragment \(F_i\) into a well-formed XML document \(D_i\), the padding algorithm consists of two sub steps. First, for each fragment \(F_i\) ending with the start-tag \((e_i), i = 1, \ldots, n - 1\), the end-tag \(\langle e_i \rangle\) is added to the end of \(F_i\) and the copied start-tag \((e_i)\) is added to the beginning of \(F_{i+1}\). Then, we add two attributes to each original start-tag \((e_i)\) : an attribute known as a cut attribute for marking the cut point, and an attribute known as an ID attribute for identifying this cut point uniquely. The value of this ID attribute is given by the value of the file offset right before the start-tag of its associated original element in \(D_i\), when counted from the beginning of \(D_i\). For differentiating the copied start-tag from the original start-tag, an attribute known as a dummy attribute is added to each copied start-tag \((e_i)\) with the same ID attribute of the original start-tag \((e_i)\).

Let \(F_i', i = 1, \ldots, n\) be the modified fragments generated by the above step. Next, the element of \(D.root\) is set to be the root element of each of \(F_i\). This can be done by concatenating \(FPad(D.root, CutElement(C_{i-1}))\), the contents of \(F_i'\), and \(BPad(D.root, CutElement(C_i))\). Let \(F_i'\) be the newly generated fragments by this concatenation. For differentiating the elements, which are generated by this concatenation, from original elements, a dummy attribute and an ID attribute are added to each start-tag forming \(FPad(D.root, CutElement(C_{i-1}))\). Note that the element that has a dummy attribute added start-tag would be called the dummy element, and each of dummy or cut elements keeps all attributes of the corresponding original element in order to forward XML Namespace information of \(D\) to each \(F_i'\).

The last operation of the padding step is to generate each \(D_i\) by attaching the prolog of \(D\) to the head of each \(F_i'\). This is for preserving the version and encoding information of \(D\). If \(D\) has a number of XML DTDs (Document Type Declaration), these DTDs are also preserved in each of \(D_i\).

The padding step explained above can be summarized more formally as follows.

**Algorithm PAD** \((D, F_1, F_2, \ldots, F_n, (C_1, C_2, \ldots, C_n), p)\)

*input*
1. \(D\): An XML document
2. \((F_1, F_2, \ldots, F_n)\): A partition of \(D\)
3. \((C_1, C_2, \ldots, C_n)\): A sequence of cut points
4. \(p\): The prolog of \(D\)

*output*
1. \((D_1, D_2, \ldots, D_n)\): A sequence of XML documents

* for each \(F_i\) from \(i = 1\) to \(n\)
  * if \((F_i, F_1, F_2, \ldots, F_n)\) is not the first fragment
    * start tag -- duplicate the start-tag of \(CutElement(C_{i-1})\);
    * add a dummy attribute to start-tag;
    * add an identification attribute to start-tag;
    * \(F_i' \leftarrow \text{start tag -} F_i\)
else
  \( F'_i \leftarrow F_i \);
end if
if \( F_i \) is not the last fragment of \( \{F_1, F_2, \ldots, F_n\} \)
  \text{endtag} \leftarrow \text{generate end-tag of CutElement} \( C_i \);
  \( F'_i \leftarrow F'_i \); endtag;
  \text{add a cut attribute to the corresponding start-tag of } F'_i \text{;}
  \text{add an identification attribute to the corresponding start-tag of } F'_i \text{;}
end if
if \( F_i \) is not the first fragment of \( \{F_1, F_2, \ldots, F_n\} \)
  \text{front} \leftarrow \text{compute } F\text{Pad}(D.\text{root}, \text{CutElement} \( C_{i-1} \));
  \text{for each start-tag in Front}
    \text{add an identification attribute to the start-tag;}
    \text{add a dummy attribute to the start-tag;}
  \text{end for}
end if
if \( F_i \) is not the last fragment of \( \{F_1, F_2, \ldots, F_n\} \)
  \text{rear} \leftarrow \text{compute } B\text{Pad}(D.\text{root}, \text{CutElement} \( C_i \));
  \( F_{00} \leftarrow \text{front-}F'_i \text{; rear-}F'_i \);
end if
end for
for each \( F'_{0i} \) from \( i = 1 \) to \( n \)
  \( D_i \leftarrow p \cdot F'_{0i} \);
end for
return \( \{D_1, D_2, \ldots, D_n\} \);

By applying the partitioning and padding algorithms, we can generate a sequence of XML documents \( \langle D_1, D_2, \ldots, D_n \rangle \) from an XML document \( D \). In this paper, we call \( \langle D_1, D_2, \ldots, D_n \rangle \) the \textbf{multi-small XML documents} of \( D \).

Fig. 3 shows an example of the generation of multi-small XML documents \( \langle D_1, D_2 \rangle \) from an XML document \( D \). In the process of partitioning \( D \) into two fragments, the location immediately after the start-tag of the node \( f \) of \( T \) is chosen as the position to split \( D \) into two fragments \( F_1 \) and \( F_2 \). The well-formed XML documents, \( D_1 \) and \( D_2 \), are then generated from \( F_1 \) and \( F_2 \) as follows:

Generation of \( D_1 \): The end-tags of the node \( f \) and all its ancestors, the node \( a \) and the node \( c \), are copied. By attaching these copied end-tags to the end of \( F_1 \), the \( D_1 \) is generated.

Generation of \( D_2 \): The start-tag of the node \( f \) and the start-tags of the node \( a \) and the node \( c \), the ancestors of the node \( f \), are copied. By attaching these copied start-tags of \( a, c, \) and \( f \) to the beginning of \( F_2 \), the \( D_2 \) is generated.

Let \( T_1 \) and \( T_2 \) be the DOM trees of \( D_1 \) and \( D_2 \), respectively. We then call the node \( f \) of \( T_1 \) the \textbf{cut node} and the three nodes \( a, c, \) and \( f \) of \( T_2 \) \textbf{dummy nodes}.

\subsection{3.3. Representation of multi-small XML documents}

Multi-small XML documents may have many cut and dummy elements. In order to browse virtually the DOM tree of an original XML document using the multi-small XML documents, how to
access the associated original node from the given dummy node and conversely how to access the associated dummy nodes from the given original node should be presented.

Let \( \{D_1,D_2,\ldots,D_n\} \) be the multi-small XML documents with the information about their associated cut nodes and dummy nodes. Each small XML document, \( D_i, i=1,\ldots,n \), is then represented by a table, called a summary table of \( D_i \), having five fields of \( \text{nr}, \text{from}, \text{to}, \text{dummy}, \text{and cut} \): the \( \text{nr} \) field is given by the value of \( i \) from \( D_i \). The from is given by the value of file offset, when counted from the beginning of the original document \( D_i \), on which the fragment \( F \) starts, while the to is given by the one on which \( F \) ends. The dummy containing the information about the dummy elements in \( D_i \) consists of three subfields, name, id, and level where Name and ID are given by the tag-name and the ID attribute value of a dummy element. Finally, the cut containing the information about the cut elements in \( D_i \) consists of two subfields, name and id, given by the tag-name and the ID attribute value of a dummy element. The following Algorithm \text{FindOriginalElement} \( D_1,D_2,\ldots,D_n \) describes this procedure more formally.

![Fig. 4. Representation of multi-small XML documents.](image)

```
Algorithm \text{FindOriginalElement} \( D_1,D_2,\ldots,D_n \) \( (id) \)
\begin{enumerate}
\item \( id \): The value of an ID attribute of a dummy node
\item \( T_i \): The DOM tree of \( D_i \)
\item \( e \): A node of \( T_i \)
\item \( z_1,z_2,\ldots,z_{\ell} \): A sequence of positive integer numbers
\item \( i \): The document number of \( D_i \) where \( (D_i,\text{from} \leq id \leq \text{to}) \) in the summary tables of \( \{D_1,D_2,\ldots,D_n\} \)
\end{enumerate}
\begin{itemize}
\item if (DOM tree of \( D_i \) is not constructed in memory)
\item Construct the DOM tree \( T_i \) of \( D_i \)
\item \( e \): the root node of \( T_i \)
\item level: the index value of \( id \) in the summary tables of \( \{D_1,D_2,\ldots,D_n\} \)
\item \( i \): \( 1 \)
\item \( \text{while} \ (\text{level} > 0) \ \text{do} \)
\item \hspace{1em} append the number of child nodes of \( e \) at the end of \( rp \)
\item \hspace{1em} \( z_\text{c} \): the number of child nodes of \( e \)
\item \hspace{1em} \( e \): the last child node of \( e \)
\item \hspace{1em} \( \text{level} \): \( \text{level} - 1 \)
\item \hspace{1em} \( i \): \( i + 1 \)
\item \( \text{end while} \)
\item return \( T_i,e \) and \( (z_1,z_2,\ldots,z_{\ell - 1}) \)
\end{itemize}
```

Next, the converse procedure to find any associated dummy nodes, if exist, from the given original node \( e \) on the DOM tree \( T_i \) is as follows: if the node \( e \) of \( T_i \) had a cut node as its descendant, there would be at least one dummy node associated with \( e \) on the DOM tree \( T_{i+1} \). The information about the associated dummy node such as its ID value is then taken from the summary table of \( D_{i+1} \) using the value of the level of \( e \) in \( T_i \). If more than one dummy node exists, then the information about these nodes is then taken from their summary tables. The following Algorithm \text{GetDummyElementLocation} describes this procedure more formally.
Algorithm GetDummyElementLocation \( T_i, e \)
[input]
1. \( T_i \): The DOM tree of \( D_i \),
2. \( e \): A node of \( T_i \).
[output]
1. \( (\text{from}, \text{to}) \): A pair of two positive integer values with holding from \( \leq \) to \( \geq \).
2. \( \text{id} \): The value of an ID attribute.

if \( e \) has a cut node as one of its descendants

level— the level of \( e \) in \( T_i \);

\( \text{id} \) — the level-\( h \) value in the summary table of \( D_{i+1} \) in \( (D_1, D_2, \ldots, D_n) \);

\( \text{from} \) — \( i + 1 \);

\( \text{to} \) — the biggest summary table number where the value of \( \text{id} \) is recorded in dummy field;

end if;
return \((\text{from}, \text{to})\) and \( \text{id} \);

3.4. Construction of virtual nodes

To keep the information of some node on the DOM tree already generated and processed, its virtual node is constructed. Let \( e \) and \( e^r \) be the original node on the DOM tree and its virtual node, respectively. The \( e^r \) is then constructed to have the values of six fields, \( \text{nr}, \text{int}, \text{type}, \text{cont}, \text{ad}, \) and \( \text{tm} \). The \( \text{nr} \) value of \( e^r \), denoted by \( e^r \).\( \text{nr} \), is given by \( k \) if the node \( e \) exists on the \( k \)th DOM tree \( T_k \) from the \( k \)th document \( D_k \) when the original document is split into \( n \) small documents. The \( \text{int} \) value of \( e^r \) is given by a finite sequence of positive integers, \( d_1d_2 \ldots d_i \), if each node on the path, except the root, from the root of \( T_k \) to \( e \) is the \( d \)th child of its parent when counted from the left. For example, if the path is \( e_0e_1e_2e \) on \( T_k \) where \( e_1 \) is the second child of the root \( e_0 \), \( e_2 \) is the first child of \( e_1 \), and \( e \) is the fourth child of \( e_2 \), then \( e^r \).\( \text{int} \) would be 214. The type value of \( e^r \) is given by either \( \text{ELEMENT} \) if the node \( e \) is an element node or \( \text{TEXT} \) if the \( e \) is a text node. The \( \text{ad} \) value of \( e^r \) is given by its physical address and finally the \( \text{tm} \) value of \( e^r \) is given by time when the DOM tree \( T_i \) is generated in memory.

Fig. 5 shows one example of generating two virtual nodes \( b^v \) and \( h^v \) keeping the information of their associated nodes \( b \) of the DOM tree \( T_1 \) and \( h \) of the DOM tree \( T_2 \): the node \( b \) is the element node with its name of \( b \), the node \( h \) is the text node with its value of “SC123”, the construction time of \( T_1 \) is 141 and the construction time of \( T_2 \) is 145.

Given an XML document \( D \) with its multi-small documents \( (D_1, D_2, \ldots, D_n) \) generated, virtual browsing of the DOM tree of \( D \) starts from constructing the DOM tree \( T_1 \) from the document \( D_1 \). The virtual root node of all \( n \) DOM trees, \( (T_1, T_2, \ldots, T_n) \), is then defined as follows

**Definition.** Let \( (D_1, D_2, \ldots, D_n) \) be the multi-small XML documents that are generated by applying the partitioning and padding steps to an XML document \( D \). We define the virtual root node, \( r^v \), of all the DOM trees \( (T_1, T_2, \ldots, T_n) \) of \( (D_1, D_2, \ldots, D_n) \) to be

\[
\begin{align*}
&\text{\( r^v \).nr} = 1; \\
&\text{\( r^v \).int} = \text{EMPTY}; \\
&\text{\( r^v \).type} = \text{ELEMENT}; \\
&\text{\( r^v \).cont} = \text{the node name of the root node of} T_1; \\
&\text{\( r^v \).ad} = \text{the physical address of the root node of} T_1; \\
&\text{\( r^v \).tm} = \text{the construction time of} T_1.
\end{align*}
\]

3.5. Operations on virtual nodes

In the specification of DOM API [5], there are four operations to access its adjacent nodes from a node \( e \), i.e. getParentNode \( (e) \) to access the parent node of \( e \), getChildNodes \( (e) \) to access the child nodes of \( e \), getPreviousSibling \( (e) \) to access the node immediately preceding \( e \), and getNextSibling \( (e) \) to access the node immediately following \( e \). These four operations enable to browse the DOM tree of an XML document. In this paper, we call these operations the **navigational operations**.

If the DOM tree of an XML document is generated in memory, each of these navigational operations can be directly implemented by returning any existing nodes locating at the position specified in
the operation. In the case of multi-small XML documents, however, these navigational operations cannot be implemented in this manner. This is because there are dummy and cut nodes on the DOM tree generated from each of multi-small XML documents which should be not visible in browsing the DOM tree of the original XML document virtually.

Thus we present the newly defined operations to manage cut and dummy nodes and also achieve the navigational operations of DOM API. We define four operations on virtual nodes \( e^d \) from the multi-small XML documents \( (D_1, D_2, \ldots, D_n) \):

\[ \text{DOWN}_{D_1, D_2, \ldots, D_n}(e^d) \]

1. Access the virtual child nodes of \( e^d \).

\[ \text{LEFT}_{D_1, D_2, \ldots, D_n}(e^d) \]

1. Access the virtual node immediately preceding \( e^d \).

\[ \text{RIGHT}_{D_1, D_2, \ldots, D_n}(e^d) \]

1. Access the virtual node immediately following \( e^d \).

\[ \text{UP}_{D_1, D_2, \ldots, D_n}(e^d) \]

1. Access the virtual parent node of \( e^d \).

The operation, \( \text{DOWN}_{D_1, D_2, \ldots, D_n}(e^d) \), finding all virtual child nodes of \( e^d \) begins by accessing the DOM tree \( T_i \), where \( i = e^d\.nr \). If \( T_i \) does not exist in memory, \( T_i \) is generated first by using a general-purpose XML parser and then the operation of \( \text{getChildNodes} \) of DOM API is executed by the XML parser.

Then, it is checked further if \( e \) has a cut node as either of itself and its descendent in \( T_i \). If yes, the additional operation of \( \text{getChildNodes} \) for every dummy node \( e_{\text{dummy}} \) of \( e \) in \( (T_1, T_2, \ldots, T_s) \) is executed. The location of \( e_{\text{dummy}} \) can be found from the summary tables of \( (D_1, D_2, \ldots, D_n) \) using the ID value of \( e \). The following Algorithm \( \text{DOWN} \) describes more formally the above procedure:

**Algorithm \( \text{DOWN}_{D_1, D_2, \ldots, D_n}(e^d) \)**

```plaintext
1. \( e^d \): A virtual node
2. \( e_1^d, e_2^d, \ldots, e_k^d \): A sequence of virtual nodes

\r
- \( i \) -- the value of \( e^d\.nr \);
- \( \text{if} \) (DOM tree of \( D_i \) is not constructed in memory)
  - construct the DOM tree \( T_i \) of \( D_i \);
- \( \text{end if} \)
- \( \text{if} \) (the construction-time of \( T_i \) is equal to the time saved in \( e^d\.tm \))
  - \( e \) -- the node saved at \( e^d\.ad \);
- \( \text{else} \)
  - \( e \) -- the node of \( T_i \) identified by the value of \( e^d\.int \);
- \( \text{end if} \)
- \( E \) -- the sequence of child node of \( e \) on \( T_i \);
- \( \text{for each} \ e_i \ of \ E \)
  - \( e_i^.nr \) -- \( i \);
  - \( e_i^.int \) -- add \( k \) at the end of \( e_i^.int \);
  - \( e_i^.type \) -- the node type of \( e_i \);
  - \( \text{if} \) (the value of \( e_i^.type \) is \( \text{TEXT} \))
    - \( e_i^.cont \) -- the value of \( e_i \);
  - \( \text{else if} \) (the value of \( e_i^.type \) is \( \text{ELEMENT} \))
    - \( e_i^.cont \) -- the name of \( e_i \);
  - \( \text{end if} \)
  - \( e_i^.ad \) -- the physical address of \( e_i \);
  - \( e_i^.tm \) -- the construction-time of \( T_i \);
  - add \( e_i \) to the tail of \( V \);
- \( \text{end for} \)
- \( \text{if} \) (\( e \) is a cut node or \( e \) has a cut node among its descendent nodes)
  - \( ((\text{From}, \text{To}), \text{ID}) \)-- call GetDummyElementLocation\( _{D_1, D_2, \ldots, D_n}(T_i, e) \);
  - \( \text{RP} \) -- generate an empty sequence of positive integers;
  - level -- the level of \( e \) in \( T_i \);
  - Add `1` at the end of \( \text{RP} \) level times;
- \( \text{for each document} \ D_j \) from \( \text{From} \) to \( j=\text{To} \)
  - construct DOM tree \( T_j \) of \( D_j \);
- \( e_j \) -- the dummy node of \( e \) in \( T_j \);
- \( E_j \) -- the sequence of child node of \( e_j \) of \( T_j \);
- \( \text{return} \ V \);
```

The operation \( \text{LEFT}_{D_1, D_2, \ldots, D_n}(e^d) \) to find the virtual node immediately preceding \( e^d \) is done by executing first the operation \( \text{getPreviousSibling} \) of \( e \). If the returned node from \( \text{getPreviousSibling} \) is an empty node, then \( e \) is the first child node of the parent node of \( e \) and thus \( \text{LEFT}_{D_1, D_2, \ldots, D_n}(e^d) \) returns an empty virtual node. Otherwise, it is checked further if the returned node \( e_1 \) has a dummy attribute. If yes, its original node is found by executing the operation, \( \text{FindOriginalElement}_{D_1, D_2, \ldots, D_n}(\text{id}) \), where \( \text{id} \) is given by the value of the ID attribute of \( e_1 \). The virtual node of the found original node is then generated and returned as the result of \( \text{LEFT}_{D_1, D_2, \ldots, D_n}(e^d) \). Otherwise, its virtual node \( e_2^d \) is generated and returned. This is described in details by the following Algorithm \( \text{LEFT} \):
Similarly, the operation $\text{RIGHT}_{D_1, D_2, ..., D_n}(e^*)$ to find the virtual node immediately following $e^*$ is done by executing first the operation $\text{getNextSibling}(e)$. If the returned node is a nonempty node $e_R$, then the $e_R$ is the node immediately following $e$ and thus its virtual node $e_R^*$ is generated and returned as the result of $\text{RIGHT}_{D_1, D_2, ..., D_n}(e^*)$. Otherwise, it is checked further if $e$ has a cut node as either of itself and its descendant. If not, the $e$ is the last child node and thus the $\text{RIGHT}_{D_1, D_2, ..., D_n}(e^*)$ returns the empty node. Otherwise, the additional operation $\text{getNextSibling}(e_{\text{dummy}})$ for the last dummy node $e_{\text{dummy}}$ of $e$ is executed. If this additional operation returns a nonempty node $e_R$, its virtual node $e_R^*$ is generated and returned as the result of $\text{RIGHT}_{D_1, D_2, ..., D_n}(e^*)$. Otherwise, $e$ is the last child node and thus the $\text{RIGHT}_{D_1, D_2, ..., D_n}(e^*)$ returns an empty virtual node. The following algorithm $\text{RIGHT}$ describes this procedure more formally:

**Algorithm $\text{RIGHT}_{D_1, D_2, ..., D_n}(e^*)$**

**input**
1. $e^*$: A virtual node

**output**
1. $e_R^*$: A virtual node

```plaintext
i ← the value of $e^*.nr$;
if (DOM tree $T_i$ is not constructed in memory)
    construct $T_i$, of $D_i$;
end if
if (the construction-time of $T_i$ is equal to the time saved in $e^*.tm$)
    $e$ ← the node saved at $e^*.ad$;
else
    $e_R$ ← the node immediately following the node $e$ on $T_i$;
    if ($e_R$ exists)
        $e_R^*.nr ← i$;
        $e_R^*.int ←$ increase the last number of $e^*.int$ by 1;
        $e_R^*.type ←$ the node type of $e_R$;
        if (the value of $e_R^*.type$ is TEXT)
            $e_R^*.tm ←$ the construction-time of $T_i$;
        end if
        else if (the value of $e_R^*.type$ is ELEMENT)
            $e_R^*.cont ←$ the name of $e_R$;
        end if
        $e_R^*.ad ←$ the physical address of $e_R$;
        $e_R^*.tm ←$ the construction-time of $T_i$;
        return $e_R^*$;
    else if ($e$ is a cut node or $e$ has a cut node among its descendant nodes)
        ([From, To], ID) ← call GetDummyElementLocation$_{D_1, D_2, ..., D_n}(T_i, e_R)$;
        $k ← To$;
        construct the DOM tree $T_k$, of $D_k$;
        $e ←$ the dummy node of $T_k$ identified by ID;
        $e_R ←$ the node immediately following the node $e$ on $T_k$;
        if ($e_R$ exists)
            $e_R^*.nr ← k$;
            $e_R^*.int ←$ increase the last number of $e^*.int$ by 1;
            $e_R^*.type ←$ the node type of $e_R$;
        end if
    end if
end if
```

The last operation $\text{UP}_{D_1, D_2, ..., D_n}(e^*)$ to get the virtual parent node of $e^*$ starts by executing the operation of $\text{getParentNode}(e)$. If the returned node $e_P$ from $\text{getParentNode}(e)$ is not a dummy node, its virtual node $e_P^*$ is generated and returned as the result of $\text{UP}_{D_1, D_2, ..., D_n}(e^*)$. Otherwise, its original node of $e_P$ is found by calling $\text{FindOriginalElement}_{D_1, D_2, ..., D_n}(id)$ where $id$ is given by the value of the ID attribute of $e_P$. The virtual node of the found original node is then generated and returned as the result of $\text{UP}_{D_1, D_2, ..., D_n}(e^*)$. This is more formally described in the following algorithm $\text{UP}$:

**Algorithm $\text{UP}_{D_1, D_2, ..., D_n}(e^*)$**

**input**
1. $e^*$: A virtual node

**output**
1. $e_P^*$: A virtual node

```plaintext
i ← the value of $e^*.nr$;
if (DOM tree $T_i$ of $D_i$ is not constructed in memory)
    construct $T_i$, of $D_i$;
end if
if (the construction-time of $T_i$ is equal to the time saved in $e^*.tm$)
    $e ←$ the node saved at $e^*.ad$;
else
    $e ←$ the node of $T_i$ identified by the value of $e^*.int$;
end if
$e_P ←$ the parent node of $e$ on $T_i$;
$e_P^*.type ←$ ELEMENT;
$e_P^*.cont ←$ the name of $e_P$;
if ($e_P$ is not a dummy node)
    $e_P^*.nr ← i$;
    $e_P^*.int ←$ Remove the last number of $e^*.int$;
    $e_P^*.ad ←$ the physical address of $e_P$;
    $e_P^*.tm ←$ the construction-time of $T_i$;
else
    ID ← the ID of $e_P$;
    ([Tj, ej, RP] ← call $\text{FindOriginalElement}_{D_1, D_2, ..., D_n}(ID)$;
    $e_P^*.nr ← j$;
    $e_P^*.int ←$ RP;
    $e_P^*.ad ←$ the physical address of $e_P$;
    $e_P^*.tm ←$ the construction-time of $T_i$;
end if
return $e_P^*$;
```

For example, each of the above four operations is illustrated in Fig. 6: Let $T_1$ and $T_2$ be two DOM trees of $D_1$ and $D_2$ respectively, and let $c^*$ be the virtual node of the node $c$ on $T_1$. The upper-left part of Fig. 6 shows the result of $\text{UP}_{D_1, D_2}(c^*)$. The parent node of the node $c$ on $T_1$ is the node $a$ on $T_1$. Since the node $a$ is not a dummy node, the $\text{UP}_{D_1, D_2}(c^*)$ returns the virtual node $a^*$ of the node $a$ on $T_1$. In the upper-right part of Fig. 6 showing the result of $\text{DOWN}_{D_1, D_2}(c^*)$, a dummy node of the node $c$ on $T_1$ exists on $T_2$. The last operation $\text{UP}_{D_1, D_2, ..., D_n}(e^*)$ to get the virtual parent node of $e^*$ starts by executing the operation of $\text{getParentNode}(e)$. If the returned node $e_P$ from $\text{getParentNode}(e)$ is not a dummy node, its virtual node $e_P^*$ is generated and returned as the result of $\text{UP}_{D_1, D_2, ..., D_n}(e^*)$. Otherwise, its original node of $e_P$ is found by calling $\text{FindOriginalElement}_{D_1, D_2, ..., D_n}(id)$ where $id$ is given by the value of the ID attribute of $e_P$. The virtual node of the found original node is then generated and returned as the result of $\text{UP}_{D_1, D_2, ..., D_n}(e^*)$. This is more formally described in the following algorithm $\text{UP}$:
Each child node of the node \( c \) on \( T_1 \) and that of the dummy node of \( c \) on \( T_2 \) are then merged to return the virtual child nodes \( \{f^v, g^v\} \) of the node \( c^v \). In merging, the dummy child nodes like \( f \) on \( T_2 \) are removed because their associated original child nodes like \( f \) on \( T_1 \) already exist. In the lower-left part of Fig. 6 showing the result of \( LEFT_{D1,D2}(c^v) \), the node immediately preceding the node \( c \) on \( T_1 \) is the node \( b \) on \( T_1 \). Since the node \( b \) is not a dummy node, the \( LEFT_{D1,D2}(c^v) \) returns the virtual node \( b^v \) of the node \( b \) on \( T_1 \). Finally, the lower-right part of Fig. 6 shows the result of \( RIGHT_{D1,D2}(c^v) \). Although there is no node immediately following the node \( c \) of \( T_1 \), there exists the node \( d \) immediately following its dummy node of the \( c \) on \( T_2 \). Thus, the \( RIGHT_{D1,D2}(c^v) \) returns the virtual node \( d^v \) of the node \( d \) on \( T_2 \).

### 3.6. Virtual nodes management

When a very large XML document \( D \) is split into \( n \) small XML documents such that \( D = \{D_1, D_2, \ldots, D_n\} \), the four operations on virtual nodes discussed in the previous section are used for browsing the DOM tree of each of those \( n \) small documents sequentially. Since the sequential browsing of each DOM tree continues replacing the current DOM tree in memory by the next DOM tree generated until the given condition is satisfied, the information of some nodes on the replaced DOM tree is kept for later reference in their virtual nodes as explained in Section 3.3. As browsing continues, however, the number of virtual nodes generated becomes larger requiring the larger memory space to be stored in. To maintain the browsing efficiency, thus, the number of virtual nodes stored is limited and also the way to manipulate the stored virtual nodes efficiently not exceeding the limited number is suggested: our experiment in the next section shows that the number of stored virtual nodes set to be equal to the average number of elements contained in each small XML document makes it feasible to browse the DOM trees without difficulty in all the examples.

Next, to manipulate the stored virtual nodes efficiently, one data structure, called Virtual Nodes Registry, is constructed: the Virtual Nodes Registry contains the hash table in which each virtual node is stored with its own key generated by the hashing function. Otherwise, one of virtual nodes already stored in the hash table is removed and this new one is then stored. The removed node would be the oldest one defined as follows: let \( e^v_{\text{rem}} \) be the new virtual node and let \( e^v_{\text{rem}} \) be the removed one. Then,

\[
e^v_{\text{old}} = \max\{e^v - e^v_{\text{rem}} | nr\} \quad \text{where} \quad e^v \in \text{the hash table}
\]

\[
e^v_{\text{rem}} = \min\{e^v_{\text{old}} | e^v_{\text{old}}.tm\}
\]

Fig. 7 shows one example of how the Virtual Nodes Registry is used when the \( DOWN \) operation is executed finding all \( m \) virtual child nodes \( \{e^v_1, e^v_2, \ldots, e^v_m\} \) of \( e^v \). The numbered arrows represent the sequence of control flow of this operation.

Using the four operations, \( UP \), \( DOWN \), \( LEFT \), and \( RIGHT \), defined on virtual nodes, an operation to browse the DOM tree of a very large XML document can be done as follows:

**Algorithm BROWSE** \((D, n, c)\)

**[input]**
1. \( D \): A very large XML document
2. \( n \): A positive integer
3. \( c \): A condition to stop browsing

**[output]**
1. \( V \): A sequence of virtual nodes

**declare** \( DP \) as a general-purpose XML Parser;
generate a sequence of \( n \) small XML documents \( \{D_1, D_2, \ldots, D_n\} \) by applying Partitioning and Padding algorithms to \( D \);
construct the DOM tree of \( D_i \) in memory by using \( DP \);
generate the virtual root node \( r^v \);
set the capacity of Virtual Node Registry to be \( k \);
\( V \) = call \( DOWN_{D1,D2,\ldots,Dn}(r^v) \);
add each virtual node of \( V \) into Virtual Node Registry;
**while** (the condition is not satisfied)
**do**
\( e^v_{\text{from}} \leftarrow \text{select a virtual node from} \ V \text{ as a starting point to navigate};
\)
**Dir** = select a navigating direction from \( e^v_{\text{from}} \);
**if** (\( \text{Dir is up} \))
\( e^v_{\text{t}} \leftarrow \text{Call} \ UP_{D1,D2,\ldots,Dn}(e^v_{\text{from}});\)
add \( e^v_{\text{t}} \) to the tail of \( V \);
**else if** (\( \text{Dir is down} \))

4. Experiments

In this section, we present the performance of our browsing method for various sizes of original very large XML documents and their partitioned small XML documents.

A prototype of our method has been implemented using Java language with JDK 1.6. The experiments have been conducted on a personal computer with an Intel Core Duo CPU (T2500, 2.0 GHz), one 5400 rpm SATA hard disk, and 2 GB of physical memory running Microsoft Windows XP (Home Edition, 32 bit) as the operating system. Among general-purpose XML parsers, Xerces-J (implemented with Java, version 2.9.0) has been chosen for constructing each of the DOM trees of multi-small XML documents.

In order to generate various-sized XML documents for the experiments, xmlgen (XMark) has been used because this software is used in many XML-related researches as a tool for generating XML files of different sizes while preserving the same structure. Using xmlgen, four XML documents with size varying from 118 MB to 1.19 GB have been generated. In addition to these artificial XML documents, we have downloaded an XML document named uniref50.xml, which contains protein sequence data, from UniProt Web site as an instance of very large XML documents in real world. Table 1 shows the size of each XML document with the number of elements contained in it.

The maximum heap size of JVM was set to 1024 MB, each experiment on a test XML document was repeated 10 times under the same condition, and the results from these repeated experiments were averaged for the final result.

First, to see the maximum size of XML document running on the given platform, each of five documents given in Table 1 was tried to browse using one single DOM tree. As shown in Table 2, five of them except the smallest one were failed to browse because the available memory is not sufficient enough to generate the associated DOM tree.

Next, our suggested method for browsing a large XML document is applied to each of these five XML documents, varying the number of small documents generated from the original XML document and also the number of virtual nodes stored. As shown in Figs. 8–12, browsing was feasible in all the cases where total time

---

```
(e_1^v, e_2^v, ..., e_m^v) ← Call DOWN(D_1, D_2, ..., D_m); add each of e_i^v to the tail of V sequentially;
else if (DIR is left)
    e_L^v ← Call LEFT(D_1, D_2, ..., D_m); add e_L^v to the tail of V;
else if (DIR is right)
    e_R^v ← Call RIGHT(D_1, D_2, ..., D_m); add e_R^v to the tail of V;
end if
add each virtual node of V into Virtual Node Registry;
end while;
return V;
```

---

**Table 1**

XML documents generated for test.

<table>
<thead>
<tr>
<th>File name</th>
<th>Size (bytes)</th>
<th>No. of elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.xml</td>
<td>118,552,713</td>
<td>1,666,315</td>
</tr>
<tr>
<td>2.xml</td>
<td>237,804,960</td>
<td>3,337,649</td>
</tr>
<tr>
<td>3.xml</td>
<td>595,691,747</td>
<td>8,353,141</td>
</tr>
<tr>
<td>4.xml</td>
<td>1,192,676,224</td>
<td>16,703,050</td>
</tr>
<tr>
<td>uniref50.xml</td>
<td>5,057,364,420</td>
<td>80,265,630</td>
</tr>
</tbody>
</table>

**Table 2**

Result from using one DOM tree.

<table>
<thead>
<tr>
<th>File name</th>
<th>Total execution time (sec)</th>
<th>Waiting time</th>
<th>DOM tree constructing time</th>
<th>DOM tree browsing time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.xml</td>
<td>12.5</td>
<td>6.97</td>
<td>4.46</td>
<td>1.07</td>
</tr>
<tr>
<td>2.xml</td>
<td>Failed due to the shortage of memory space</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.xml</td>
<td></td>
<td>6.97</td>
<td>4.46</td>
<td>1.07</td>
</tr>
<tr>
<td>4.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uniref50.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Fig. 7.** Example of virtual nodes management by using Virtual Nodes Registry.
on the left column is the average time to access every node of all the DOM trees of the \( n \) small XML documents starting from the root. The 100% on the right column means the number of stored virtual nodes is set to the average number of elements contained in each of multi-small XML documents.

From this result, the following may be claimed:

1. Our method makes it feasible to browse the DOM tree of the very large XML document.
2. Since our method browses the DOM trees of the \( n \) small XML documents \( D_1, D_2, \ldots, D_n \) sequentially, the time necessary for generating each DOM tree is traded off with the time necessary for generating total number of DOM trees so that
there is slight time difference between the large number of XML documents with small size and the small number of XML documents with large size partitioned from the original large XML document \( D \).

(3) The partitioning \( \langle F_1, F_2, \ldots, F_n \rangle \) from \( D \) may be optimal if our method working on \( \langle D_1, D_2, \ldots, D_n \rangle \) generated from \( \langle F_1, F_2, \ldots, F_n \rangle \) guarantees the best performance of browsing speed. As the size of the DOM tree constructed from each \( D_i \), \( i = 1, \ldots, n \), becomes closer to the size of the currently available main memory, the better browsing speed may be expected. To find the optimal size of each \( F_i \) (to find the value of \( n \) equivalently) in advance is however infeasible with the following two reasons: first, the size of the DOM tree from each \( F_i \) can be determined only after the size of the DOM tree of the original XML document \( D \) is calculated which is however not possible due to its very large size but the limited size of the main memory. Secondly, the size of the currently available main memory cannot be measured exactly because it constantly varies as the browsing operation is being done. Instead, as shown in Figs. 8–12, the size of \( D_i \) with around 1 MB resulted in good performance and its size larger than 1 MB not necessarily resulted in better performance. Thus, for the given very large XML document having its size larger than 100 MB, our method may start with around 1 MB size of \( D_i \) and increase its size to try the better browsing.

(4) In setting the number of stored virtual nodes, there is no big time difference between 100%, 300%, and 600%, which may be due to the characteristic of the application used in these examples that needs for later reference only small portion of nodes of the previous DOM tree already generated. Fig. 13 shows total running time, waiting time, partitioning and padding time, DOM tree construction time, and browsing time of each of five XML documents given in Table 1, where the average size of each \( D_i \), \( i = 1, \ldots, n \), varies from 1 MB to 2 MB. It is noted that the execution time of our method increases almost linearly to the size of a given XML document.

Finally, for the comparison with other approaches, we chose three different XML document management systems: Apache Xindice, eXist, and Microsoft SQL Server. Apache Xindice and eXist are the disk-based native XML storage approaches and Microsoft SQL Server is a database system for XML.

<table>
<thead>
<tr>
<th>Table 3</th>
<th>Result from browsing 10,000 protein entries.</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Execution time for formatting XML storage (sec)</td>
</tr>
<tr>
<td>Our method</td>
<td>13</td>
</tr>
<tr>
<td>SQL Server 2005</td>
<td>119</td>
</tr>
<tr>
<td>Xindice 1.1</td>
<td>151</td>
</tr>
<tr>
<td>eXist 1.2.4</td>
<td>182</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4</th>
<th>Result from browsing 100,000 protein entries.</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Execution time for formatting XML storage (sec)</td>
</tr>
<tr>
<td>Our method</td>
<td>122</td>
</tr>
<tr>
<td>SQL Server 2005</td>
<td>830</td>
</tr>
<tr>
<td>Xindice 1.1</td>
<td>1,202</td>
</tr>
<tr>
<td>eXist 1.2.4</td>
<td>1,811</td>
</tr>
</tbody>
</table>
Server is the relational database-based XML storage approach. As we inserted the protein entries in `uniref50.xml` into each of these systems, two execution time, one for formatting XML storage and the other for querying an XML path ‘/entry/name’ which returns the name of each protein entry on the saved protein sequences, were checked with the required storage size. In our system, the ‘/entry/name’ query was processed by the query program using the proposed four navigational operations in Section 3, while in three other systems, the query was processed by each of their own query processors.

Tables 3–6, show that our method needs less time in formatting XML storage to perform the required operations such as querying. This is because our method keeps the hierarchy structure of the original XML document almost identically while other systems such as Xindice and eXist restructure the XML document for rapid query processing. As the size of XML storages becomes larger by inserting XML data sequentially, this restructuring process needs much more time.

Fig. 14. Initial parameter setup for generating multi-small XML documents and restricting the number of virtual nodes.

Table 5
Result from browsing 1,000,000 protein entries.

<table>
<thead>
<tr>
<th>System</th>
<th>Execution time for formatting XML storage (sec)</th>
<th>Execution time for querying ‘/entry/name’ (sec)</th>
<th>XML storage size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our method</td>
<td>1,320 (22 min)</td>
<td>630</td>
<td>3,202</td>
</tr>
<tr>
<td>SQL Server 2005</td>
<td>16,860 (281 min)</td>
<td>698</td>
<td>4,000</td>
</tr>
<tr>
<td>Xindice 1.1</td>
<td>Not finished in 24 h</td>
<td>Not observable</td>
<td>Not observable</td>
</tr>
<tr>
<td>eXist 1.2.4</td>
<td>Not finished in 24 h</td>
<td>Not observable</td>
<td>Not observable</td>
</tr>
</tbody>
</table>

Table 6
Result from browsing 1,914,193 protein entries.

<table>
<thead>
<tr>
<th>System</th>
<th>Execution time for formatting XML storage (sec)</th>
<th>Execution time for querying ‘/entry/name’ (sec)</th>
<th>XML storage size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our method</td>
<td>2,100 (35 min)</td>
<td>1,131</td>
<td>5,060</td>
</tr>
<tr>
<td>SQL Server 2005</td>
<td>29,640 (494 min)</td>
<td>1,068</td>
<td>7,142</td>
</tr>
<tr>
<td>Xindice 1.1</td>
<td>Not finished in 24 h</td>
<td>Not observable</td>
<td>Not observable</td>
</tr>
<tr>
<td>eXist 1.2.4</td>
<td>Not finished in 24 h</td>
<td>Not observable</td>
<td>Not observable</td>
</tr>
</tbody>
</table>
5. Conclusions

In this paper, we presented a method to browse the DOM tree of a very large XML document by splitting the XML document into \( n \) small XML documents and constructing sequentially the DOM tree of each of these \( n \) small XML documents. In constructing the DOM trees sequentially, the information of some nodes of the DOM tree already constructed were kept using their virtual nodes for later reference. Four navigational operations necessary for browsing were also defined on virtual nodes. Finally, our method was illustrated for several very large XML documents in which the direct browsing using one DOM tree was not feasible. With our approach browsing the DOM tree of the very large XML document can be done in a general-purpose personal computer.

References


Seung Min Kim received a B.S. and M.S. from Computer Science at Kyungpook National University, Taegu, South Korea, in 1995 and 1997. He is a Ph.D candidate in Computer Science at Seoul National University, Seoul, Korea. His research area includes XML, GRID, WEB, and Distributed Systems.
Suk I. Yoo received a B.S. from Electrical Engineering at Seoul National University, Seoul, South Korea, in 1977, a M.S. from Electrical Engineering at Lehigh University, Bethlehem, PA, in 1980, and a Ph.D from Computer Engineering at University of Michigan, Ann Arbor, MI, 1985, respectively. Since he joined the faculty of Dept. of Computer Science, Seoul National University, in 1985, he has been an assistant professor, an associate professor, and a professor in School of Computer Science and Engineering, Seoul National University, Seoul, Korea. He is a member of ACM, IEEE, and SPIE. His research area includes computer vision, problem-solving, machine learning, XML, and GRID.