A Document Object Modeling Method to Retrieve Data from a Very Large XML Document

Seung Min Kim
School of Computer Science and Engineering
Seoul National University
Republic of Korea
mowgli@ailab.snu.ac.kr

*Suk I. Yoo
School of Computer Science and Engineering
Seoul National University
Republic of Korea
siyoo@ailab.snu.ac.kr

Eunji Hong
Department of Software Engineering
Sung-Kong-Hoe University
Republic of Korea
hong@skhu.ac.kr

Tae Gwon Kim
Department of Computer and Media Engineering
Kangnam University Republic of Korea
ktg@kangnam.ac.kr

Il Kon Kim
Graduate School of Computer Science
Kyungpook National University Republic of Korea
ikkim@knu.ac.kr

ABSTRACT

Document Object Modeling (DOM) is widely used approach for retrieving data from an XML document. If the size of the XML document is very large, however, using the DOM approach for retrieving data from the XML document may suffer from a lack of memory space for building the associated XML tree in the main memory. To alleviate this problem, we propose a method that allows the very large XML document to be split into small XML documents, retrieves data from the XML tree built from each of these small XML documents, and combines the results from all of the *n* XML trees to generate the final result. With this proposed approach, the memory space and processing time required to retrieve data from the very large XML document using DOM are reduced so that they can be managed by one single general-purpose personal computer.

Categories and Subject Descriptors
I.7.1 [**Document and Text Processing**]: Document management; F.2.2 [**Nonnumerical Algorithms and Problems**]: Computations on discrete structures

General Terms: Algorithms, Performance, Experimentation

Keywords: XML, DOM, DOM API, Very Large XML Documents

1. INTRODUCTION

XML is a W3C-recommended general-purpose markup language [1]. XML and its related technologies are being used widely as the standard methods for representing and exchanging information on Web environments due to their flexibility in information modeling. Information has been modeled using XML in the various fields including information retrieval, document exchange, document management, data mining and electronic publishing.

*Suk I.Yoo is a corresponding author.

One of the noteworthy trends happening with XML in recent years is that large XML documents, whose sizes range from several hundreds of MB to several GB, are now being generated. Usually, a large XML document is the aggregation of a number of relatively small XML data, which have a common tree structure. For instance, the sizes of XML files for protein sequences [2] range from 1.8~16 GB. But each of these large sizes is due to the aggregation number of much smaller XML data, whose size is about 1~10 KB.

XML parsers such as Apache Xerces [3] can be used for managing the contents of XML documents. The parsing methods of these XML parsers can be classified into two different groups with their methods for accessing XML documents: DOM [4] and SAX [5]. DOM models an XML document as a tree structure for every XML application so that the GET, INSERT, DELETE and UPDATE operations can be easily done using the predefined DOM API. However, it suffers from a lack of memory space for building the tree when the size of a document is very large. On the other hand, SAX parses the XML document sequentially from the beginning of the document whenever it is requested to find some data. Thus, to get some data from the document, each XML application using SAX first predefines the related data structure and stores the data (from parsing) into this data structure. Then, each XML application using SAX is not easily completed with the repetitive full file scanning and the work of predefining various data structures, when it needs many randomly located data with a different data type.

In this paper, we present a DOM method for retrieving data from a very large XML document with manageable memory space and processing time by a single general-purpose personal computer. A very large XML document is partitioned into *n* small documents, where *n* varies depending on the capacity of the given resource such as a personal computer. Each of the *n* small documents is then modified by a padding process to meet the well-formedness of the XML document. A data retrieval operation on the original large XML document, which is expressed with DOM API, is then executed sequentially on the small XML tree that is built from each of the modified *n* XML documents, and the results from all the *n* XML trees are combined to generate the final result. With this approach, the data retrieval operations on the very large XML document can be executed by a single general-purpose personal computer.

2. RELATED WORK

Several approaches for processing XML data efficiently have been published. Noga et al. [6] proposed a lazy XML processing approach, which consists of the preprocessing phase and the progressive parsing phase. In the preprocessing phase, they built the internal representation per each node of the tree by analyzing the document structure. In the progressive parsing phase, the internal representation of each node is transformed into its physical node of the tree when it is accessed for the first time. They claimed that the performance of their approach would be maximized if the number of transformed nodes does not exceed 80% of the number of total nodes. This approach is efficient in constructing the tree rapidly but is not efficient when 80% nodes of the XML tree are accessed during the execution of an operation. Further, the advantage of rapid construction of the tree may not be realized if the size of the given document is so large that the memory necessary for the internal representation exceeds the allowed free memory space.

Huang et al. [7] proposed a model that employs a prefilter to remove uninteresting fragments of an input XML document by approximately executing a user's queries. The XML document consisting of the candidate-set is then returned to the user's DOM- or SAX-based applications for further processing. The performance result of the sample XPath [8] query "/site/regions/asia" against XML documents, which are generated by XMark [9], showed that it could reduce such computational resources as CPU time and memory that are needed for parsing and data retrieving. However, this approach has a drawback in that it has to execute prefiltering whenever a new query to the XML document is given.

XDBMS- and RDBMS-based approaches provide another solution for processing XML data. Lu et al. [10] reported the result on benchmarking a set of XML database implementations using XMark and XMach [11] benchmarks. The selected implementations represent a wide range of approaches, including RDBMS-based systems with document-independent [12, 13, 14] and document-dependent XML-relational schema mapping approaches [15, 16], and XML native engines [10, 17] based on an Object-Oriented Model and the Document Object Model. In order to use XDBMS- and RDBMS-based approaches, however, XML data must be preprocessed and stored in DBMS and XPath queries must be translated into SQL queries. Furthermore, XDBMS- and RDBMS-based approaches are too expensive to be used in small-scale applications. Some researchers have also claimed that the performance of relational XML database degrades when dealing with huge XML data [18].

Kido et al. [18] proposed a scheme called PC cluster for the parallel processing of XML data using a group of personal computers. Each computer of the PC cluster runs the same version of RDBMS using [14]. To make parallel processing possible, they suggested a method of partitioning the XML data based on the subgraph decomposition of a schema graph and subset decomposition of XML partitions. To allocate XML partitions to cluster nodes, they give an algorithm for computing suboptimal assignment by applying a greedy method and a genetic algorithm. Although they could speed up the performance by using a PC cluster, the dependency of the PC cluster may be a barrier to an application that uses a stand-alone computing environment.

The approach of Wei Lu et al. [19] is the one that relates most to our method from the viewpoint of data partitioning. Here, they designed and implemented a 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 XML tree 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. They partition the XML document, based on 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 generated from the chunks are attached to the main XML tree when the parallel parsing phase is completed. This approach may be efficient in constructing the tree rapidly. However, it also suffers 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.

3. RETRIEVING DATA FROM A VERY LARGE XML DOUCMENT USING MULTI SMALL XML DOCUMENTS

In this section, we first discuss the difficulty faced when a very large XML document is parsed to build one huge XML tree in the main memory by a general-purpose XML parser supporting DOM (i.e. a DOM Parser). We then present a method to retrieve data from it in terms of multi-manageable small XML documents.

Through the use of DOM, an XML document can be modeled as a tree, called an XML tree in this paper, as shown in Figure 1. Each element of an XML document is mapped onto a node where the name of a node is the tag name of the corresponding start- and end-tag, and the parent-child relationship of two elements is mapped onto an edge between the two associated nodes of an XML tree. The root element of the XML document becomes the root of the XML tree.

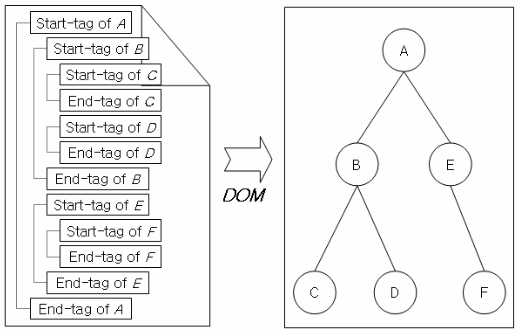


Figure 1. XML Document and XML Tree

Building the corresponding XML tree from an XML document is the core process of DOM parsing: a DOM parser scans the full contents of the XML document sequentially on a character-by-character basis from the beginning of the document until it reaches the end of the document. If an element is found in the scanning process, the DOM parser stores its structure and value into memory. All operations expressed with DOM API are managed on this XML tree.

A general-purpose DOM parser such as Xerces keeps requesting memory allocations until the complete XML is generated in the virtual memory of a computer. Whenever a DOM parser finds an element, it requests memory allocation to store the element into memory. Therefore, if we use DOM API to retrieve data from a very large XML document, we need sufficient virtual memory space during the lifetime of the parsing process.

We carried out some DOM parsing experiments on various-sized XML files in order to check the DOM parsing speed. We used 100 sample XML files, varying in size from 10 to 1,000 MB in increments of 10 MB. These XML files are generated by aggregating small XML data, which have a common tree structure. Using these XML files, we examined the memory and time required for DOM parsing. The hardware platform of this experiment was a personal computer with a Pentium 4 CPU (3.0 GHz, EM64T) and 10 GB of virtual memory (3 GB of physical memory plus 7 GB of swap space) and which ran Ubuntu 6.06 (Linux 2.6, 64 bit) as its operating system. The DOM Parser we chose was Xerces-J (implemented with Java, version 2.9.0). To restrict the overhead of to increase the heap size in the process of parsing, we set the value of –Xms and –Xmx options of JVM (Java Virtual Machine) to the same value, 8192m. The results of these experiments are shown in Figure 2, 3 and 4.

In Figure 2, when the size of a test file becomes large, the required memory space for storing elements in a tree structure increases linearly. This matches the result of the theoretical analysis of a DOM parsing algorithm. However, as we can see from Figure 3, the DOM parsing time of an XML document does not increase linearly with the size of the document. There are two jumping points of parsing time. In Figure 3, we see that there is a big jump of parsing time at around the 760 MB point. In Figure 4, which contains the same data as Figure 3 but with an X range narrowed to 100~760 MB, we see that there is another jump of parsing time at around the 390 MB point.

In every experiment, the jumping points are not exactly the same as before, but this pattern of having two jumping points has always been observed. When there are repeated requests for memory allocation in spite of the exhaustion of free space in the physical memory, swapping occurs frequently between the physical memory and swap area in a second memory device. This explains the first jumping point. If the size of memory, which is required for building the XML tree of an XML document, reaches the total size of virtual space, every process in the computer spends most of its time waiting for the completion of memory access operations. This explains the second jumping point. The inefficiency for handling large memory caused by the heap space managing algorithms of the JVM is also a critical factor of this pattern.

This result shows that the DOM method using one single XML tree is not a good approach for a very large XML document; the memory space and processing time are not manageable by one single general-purpose personal computer. Based on this analysis, we designed a DOM method that uses *n* manageable small XML trees instead of using one large XML tree to avoid the shortage of free memory size and low efficiency in parsing.

To execute a data retrieval operation on the *n* manageable small XML trees in terms of the original XML tree, the associated *n* XML documents are first generated from the original XML document. Our method generates the *n* XML documents in two steps, partitioning and padding. In the partitioning step, the original XML document is split into *n* nearly equal-sized documents where *n* is decided depending on the resource available, such as the capacity of the personal computer. In the padding step, each of the *n* documents is modified to meet the requirement of a well-formed XML document. The data retrieval operation is then performed sequentially on each of the *n* XML trees built from the *n*-associated XML documents in the retrieving step. The result of the operation from the first XML tree is then combined with the one from the second XML tree, which continues until the result from the last XML tree is combined. Partitioning, padding and retrieving are each explained in further detail below.

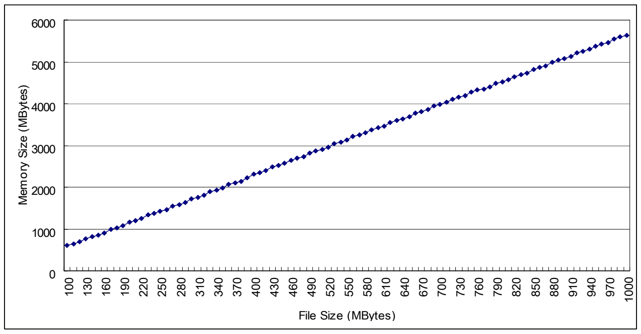


Figure 2. Memory Size required for DOM Parsing

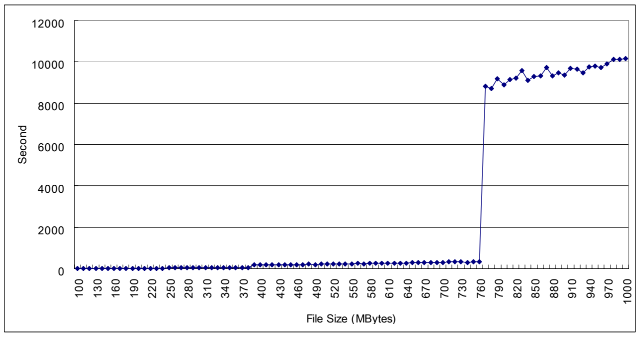


Figure 3. DOM Parsing Time of 100~1,000 MB XML files

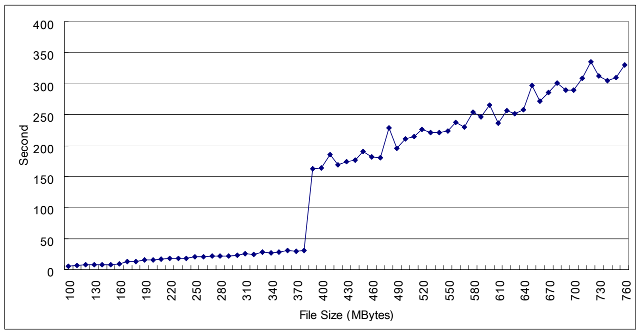


Figure 4. DOM Parsing Time of 100~760 MB XML files

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, *…, and* F_n, *is a* *subsequence of D such that the concatenation of F_1, F_2,…, and F_n is equal to D, then a sequence of the n documents, <F_1, F_2, …, F_n>, is a* ***partition of a document*** *D where each F_i, i=1,…,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 i-th character to j-th character of a document D is expressed as D[i, j]. Therefore, the contents of a 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_o be a given XML document and *n* be the expected number of fragments in the partition of D_o. First, the $\lfloor D_o.length / n \rfloor$ characters are scanned from the beginning of D_o, and around the location right after these characters, the range of width, *h*, is formed as shown in Figure 5.

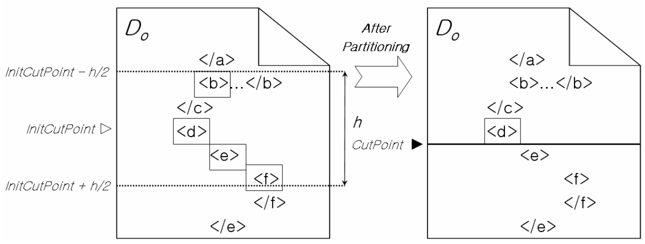


Figure 5. Finding a cut point

Next, we compute the length of the path from the root element of D_o to each of the elements whose start-tags are contained in this range. For computing the path without building the XML tree of D_o in memory, we use a stack containing strings. This stack keeps the list of element names from the beginning of D_o 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_o 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_o after the range. For the second cut point, the range of width, *h*, is again formed around the location right after $\lfloor D_o.length / n \rfloor$ 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 selected this policy to decide the cut points because we observed three major things from experiences with a very large XML document *D*. First, *D* is in general an aggregation of small XML documents X_i, *i=1,…, k,* which have a common structure and are almost the same size. Second, each X_i is directly or very closely connected to the root element of *D*. Third, the data retrieval operations on *D* can usually be applied on each X_i independently. Therefore, considering these characteristics, it is a good approach not to split each X_i into different fragments. Each cut point, which comes as near to the root element as possible in a given range, is selected for achieving this purpose. This is why we use an external variable, *h*, for defining the width of ranges. If the value of *h* is carefully selected, each fragment generated could be associated easily and rapidly with a set of these 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_o is less than *n*, or the positions of start tags of D_o lean severely toward some positions of D_o, 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. Besides, there are very small possibilities that the situations that prevent the work of our partitioning algorithm occur in real areas having very large XML documents.

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

Algorithm ***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. n' : The actual number of fragments generated
  2. <$F_1$ $F_2$ …$F_{n'}$> : A partition of D ($1 \leq n' \leq n$)
  3. <$C_1$ $C_2$ …$C_{n'-1}$> : A sequence of cut points
  4. P : The prolog of D
```

```
Declare StackOfStartTags as a stack of strings;
Declare $T_s$, $T_{new}$ as variables pointing to start- or end-tags;
TargetSize := $\lfloor D.length / n \rfloor$;
Initialize StackOfStartTags;
Prolog := the prolog of D;
StartPoint := the offset of the end of prolog of D + 1;
CutPoint := StartPoint;
i := 1;
While ((StartPoint < D.length) and (i < n)) do
  InitCutPoint := StartPoint + TargetSize;
  From := CutPoint;
  While (there is a tag in [From, D.length] of D) do
    Scan a tag and set $T_s$ to point to the tag retrieved;
    If ($T_s$ points to a start-tag)
      Push the name of the pointed start-tag by $T_s$ into StackOfStartTags;
      If (the start-tag pointed by $T_s$ is in [InitCutPoint – h/2, D.length] of D)
        jump FindCutPoint;
    End if
    If ($T_s$ points to an end-tag)
      Pop the last item from StackOfStartTags;
    From := (the offset of the end of the pointed tag by $T_s$)+1;
  End while
FindCutPoint:
  minimalDistance := the number of items in StackOfStartTags;
  CutPoint:= (the offset of the end of the pointed tag by $T_s$) + 1;
  Scan a tag and set $T_{new}$ to point to the tag retrieved;
  While (the tag pointed by $T_{new}$ is in [CutPoint, InitCutPoint + h/2] of D) do
    If ($T_{new}$ points to a start-tag)
      Push the name of the pointed tag by $T_{new}$ into StackOfStartTags;
      Distance := the number of items in StackOfStartTags;
      If (minimalDistance > Distance)
        minimalDistance:= Distance;
        $T_s$:= $T_{new}$;
        CutPoint:= (the offset of the end of the pointed tag by $T_s$)+1;
      End if
    End if
    If ($T_{new}$ points to an end-tag)
      Pop the last item from StackOfStartTags;
    Scan a tag and set $T_{new}$ to point to the tag retrieved;
  End while
  $F_i$ := D[StartPoint, CutPoint];
  StartPoint := CutPoint + 1;
  $C_i$ := CutPoint;
  i := i + 1;
End while
$F_i$ := D[StartPoint, $D_o$.length];
n':= i;
Return n', <$F_1$ $F_2$ ... $F_{n'}$>, <$C_1$ $C_2$ ... $C_{n'}$> and Prolog;
```

3.2 Padding

Since each fragment generated in the partitioning step does not comprise 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 start-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_{i1} and e_{in} be two nodes of an XML tree where e_{i1} is an ancestor of e_{in}. If the sequence of nodes on the path from e_{i1} to e_{in} is given by $e_{i1}e_{i2}...e_{i(n-1)}e_{in}$, then*

*(1) The **front pad** from e_{i1} to e_{in} is a string given by*

$$<e_{i1}><e_{i2}>...<e_{i(n-1)}>$$

*(2) The **back pad** from e_{i1} to e_{in} is a string given by*

$$</e_{i(n-1)}>...</e_{i2}></e_{i1}>$$

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

1. *The cut element, split at the position of C_i into two fragments, is expressed as CutElement(C_i).*
2. *The first start-tag of a fragment F is expressed as F.first .*
3. *The last end-tag of a fragment F is expressed as F.last.*
4. *The root of an XML document D is expressed as D.root.*
5. *The front pad from e_{i1} to e_{in} is expressed as FPad(e_{i1}, e_{in}).*
6. *The back pad from e_{i1} to e_{in} is expressed as BPad(e_{i1}, e_{in}).*

Let D_o be a given very large XML document and $F_1,...,F_n$ be n fragments generated by the partitioning step as shown in Figure 6.

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 $<e_i>$, $i=1, ..., n-1$, the end-tag $</e_i>$ is added to the end of F_i and the start-tag $<e_i>$ is added to the beginning of F_{i+1}. For differentiating the cut elements from original elements, an attribute known as a cut attribute is added to each start-tag $<e_i>$ additionally. In Figure 7, the dotted rectangles show the added start-tags or end-tags. The shaded rectangles represent the start-tags of cut elements.

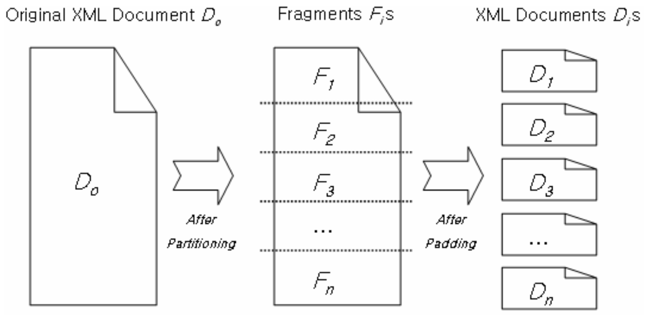


Figure 6. Well-formed XML Documents generated by Padding

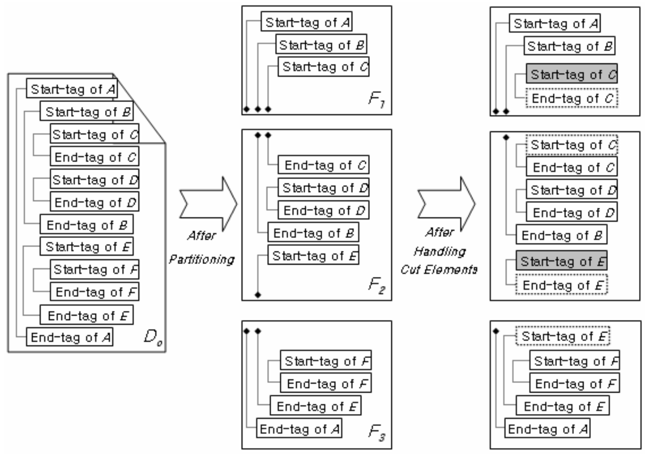


Figure 7. Generated start- and end-tags of cut elements

Let F'_i, $i=1,...,n$, be the modified fragments generated by the above step. Next, the element of $D_o.root$ is set to be the root element of each of F'_i, $i=1,...,n$. This can be done by concatenating *FPad(D_o.root, CutElement(C_{i-1}))*, the contents of F'_i, and *BPad(D_o.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, an attribute known as a dummy attribute is added to each start-tag forming *FPad(D_o.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 [23] information of D_o to each F''_i.

The last operation of the padding step is to generate each D_i by attaching the prolog of D_o to the head of each F''_i. This is for preserving the version and encoding information of D_o. If D_o has a number of XML DTDs (Document Type Declaration) [1], these DTDs are also preserved in each of D_i's.

Figure 8 shows the well-formed XML document D_1, D_2, D_3 and Figure 9 shows the tree structures of T_1, T_2 and T_3, which are generated from the given XML document D_o with a tree structure of T_o by the partitioning and padding steps. In Figure 9, the shaded ellipses represent the cut elements, and the ellipses filled with lines represent the dummy elements.

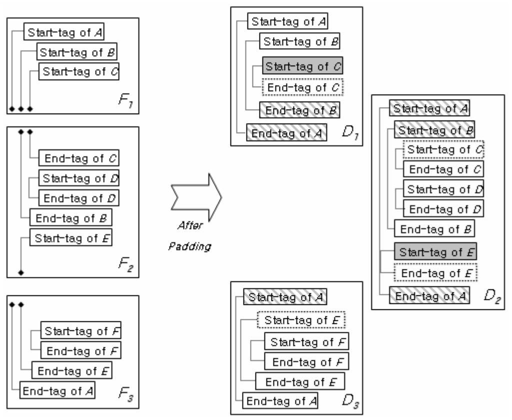


Figure 8. Applying *PAD* algorithm

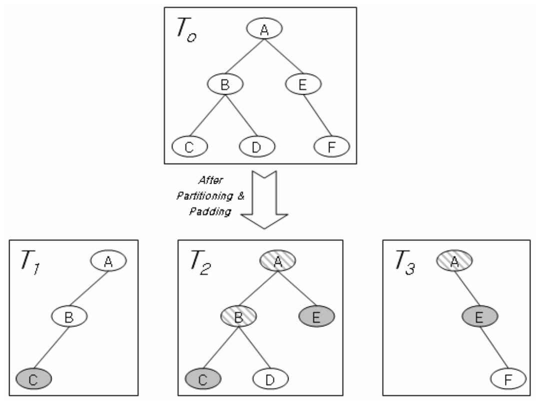


Figure 9. XML Trees generated by *PARTITION* and *PAD* algorithm

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

Algorithm *PAD(D, <F_1 F_2 …F_n>, <C_1 C_2 …C_n>, P)*
<Input>
1. *D* : An XML document
2. <F_1 F_2 …F_n> : A partition of *D*
3. <C_1 C_2 …C_n> : A sequence of cut points
4. *P* : The prolog of *D*

<Output>
1. <D_1 D_2 …D_n> : A sequence of XML documents

```
Declare <$F'_1$ $F'_2$ … $F'_n$>, <$F''_1$ $F''_2$ … $F''_n$> as sequences of documents;
For each $F_i$ from i=1 to n
  If ($F_i$ is not the first fragment of <$F_1$ $F_2$ …$F_n$>)
      StartTag :=Duplicate the start-tag of CutElement($C_{i-1}$);
      Add a cut attribute to StartTag;
      Add an identification attribute to StartTag;
      $F'_i$:= StartTag ·$F'_i$;
  Else
      $F'_i$:= $F_i$;
  End if
  If ($F_i$ is not the last fragment of <$F_1$ $F_2$ …$F_n$>)
     EndTag := Generate the end-tag of CutElement($C_i$);
     $F'_i$ := $F'_i$·EndTag;
     Add a cut attribute to the corresponding start-tag of $F'_i$.last;
     Add an identification attribute to the corresponding start-tag of $F'_i$.last;
  End if
  If ($F_i$ is not the first fragment of <$F_1$ $F_2$ …$F_n$>)
     Front := Compute FPad(D.root, CutElement($C_{i-1}$));
     For each start-tag in Front
       Add an identification attribute to the start-tag;
       Add a dummy attribute to the start-tag;
     End for
  End if
  If ($F_i$ is not the last fragment of <$F_1$ $F_2$ …$F_n$>)
     Rear := Compute BPad(D.root, CutElement($C_i$));
  $F''_i$ := Front · $F'_i$ · Rear;
End for
For each $F''_i$ from i=1 to n
  $D_i$:= P · $F''_i$;
End for
Return <$D_1$ $D_2$ … $D_n$>;
```

3.3 Retrieving

In this section, we explain how to execute a given data retrieval operation using a number of small XML documents, which are generated by the previous partitioning and padding steps. The data retrieval operation is expressed with DOM API, and this operation is given on the assumption that the XML tree of an original large XML document exists in main memory.

We selected following GET operations from DOM API as the representatives of data retrieval operations.

Table 1. Data Retrieval Operations in DOM API

Interface	Operation Name	Description
Document	*getElementsByTagName*	Returns a nodelist of all the elements in document order with a given tag name
Node	*getChildNodes*	Returns a nodelist that contains all children of this node
	getFirstChild	Returns the first child of this node.
	getLastChild	Returns the last child of this node.
Element	*getElementsByTagName*	Returns a nodelist of all descendant elements with a given tag name, in document order.

The operations in Table 1 have a difference in external forms. However, each of these GET operations can be considered to receive some nodes that satisfy the specified condition from the given XML tree, *T*. Thus, it suffices to show how to implement the operation of *GET(T,e,P,S)*, which returns the nodes that satisfy the condition *P* from a subtree, having *e* as its root, of *T*. The 4th parameter *S* specifies how to make the result of the operation from the retrieved data. The value of *S* can be one of three values, *ALL, FIRST and LAST.* Let N_r be the nodes satisfying the condition *P* from a subtree, having *e* as its root, of *T*. If the value of *S* is *ALL*, this GET operation returns N_r as a list of nodes. Yet, if the value of *S* parameter is *FIRST* or *LAST,* this GET operation returns the first or last node of N_r correspondingly.

For example, the operation of *getElementsByTagName(XXX)* with *Document* interface on an XML tree *T* can be given by *GET (T, the root of T, tagname=XXX, ALL)* and the operation of *getFirstChild()*

with *Node, e,* interface on an XML tree *T* can be given by *GET(T, e, child of e, FIRST)*.

Let T_o be the XML tree of a very large XML document D_o, and $T_1,...,T_n$ be *n* XML trees built from small *n* XML documents $D_1,...,D_n$, which are generated from D_o by the partitioning and padding steps. In this retrieving step, a given $GET(T_o,e,P,S)$ operation is then executed sequentially on each of T_i, $i=1,...n$, as T_o replaced with T_i, using a general-purpose DOM parser. These replacements are valid, since any ancestor of each element of D_o is preserved in any of the *n* small XML trees containing that element.

On completing the GET operation on each of T_i, $i=1,...n$, a list of elements that satisfy the given condition, if it exists, is found. Next, to combine the results from $T_1,...,T_n$, in the form of another XML tree, a tree T_r having only one root node is first generated and then updated sequentially after each $GET(T_i,e,P,S)$ is executed.

Let $N_1,...,N_n$ be the *n* results of the GET operations from $T_1,...,T_n$, where each N_i is either a list of nodes or an empty list. For each nonempty N_i, each node of N_i is checked to see if it has either a dummy attribute or a cut attribute. If it has either of these, we first check that there is a node of T_r which has the same value of the identification attribute of N_i. If it exists, the contents of N_i, including its descendant in T_i, are copied to the contents of the node having same value of the identification attribute. Otherwise, the copy of N_i, also including its descendant in T_i, is added to be a new child node of the root of T_r. This process repeats until the last search result N_n is reflected on the tree T_r. Based on the tree T_r which is formulated this way, the final result of the $GET(T_o,e,P,S)$ operation is returned after applying *S* on the child nodes of the root of T_r; if *S* is *ALL*, this GET operation returns all child nodes as a list of nodes. However, if the value of *S* is either *FIRST* or *LAST,* this GET operation returns either the first or last child node of the root of T_r correspondingly.

For example, suppose that the *GET*(T_o*, A, the child node of A, FIRST)* operation is executed on the structure of T_o shown in Figure 9. If a general DOM parser is directly applied on T_o for executing this operation, it finds *B* from this subtree as the one which satisfies the given condition, and returns <*B* of T_o> as the final answer. If it is executed on the three small trees T_1, T_2 and T_3 in Figure 9,we get three search results from T_1, T_2 and T_3, which are <*B of* T_1>, <*B of* T_2>, and <*E of* T_3>. These results are combined into <*B of* T_r*, E of* T_r>, because the nodes *B of* T_1 and *B of* T_2 have the same value of the identification attribute. The *S* of this operation is *FIRST*, so <*B of* T_r> is returned as the final result of this operation, as shown in Figure 10.

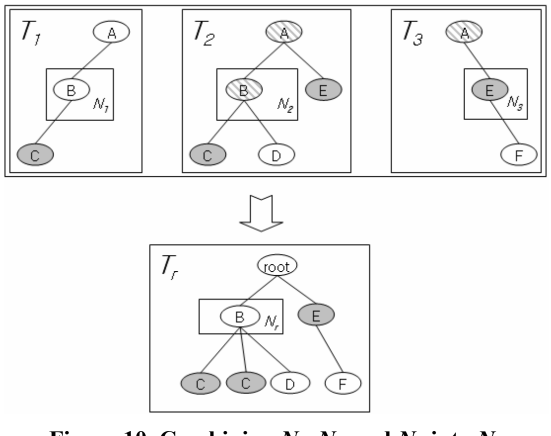


Figure 10. Combining N_1, N_2 and N_3 into N_r

As explained above, in combining *n* results of GET operations, if a node contained in some of *n* results has either a dummy or cut node as a child node, the copy of this dummy or cut node is also contained in the combined result, as shown from two *C* nodes of <*B of* T_r> in Figure 10. This is because the original *C* element in the original XML Document D_o was split by the *PARTITION* and *PAD* algorithm intentionally. Therefore, the final step for retrieving data is to unify the split nodes by removing the duplicate information generated due to such dummy or cut nodes. We apply a lazy approach to unify the split child nodes: at the first time a node *e* of a node list is referred by an XML application, we combine child nodes of *e* having a dummy or cut attribute and the same identifier. This can avoid time consumption caused by complete navigation of subtrees for unifying unused nodes of T_r.

The final result from Figure 10 is then as follows.

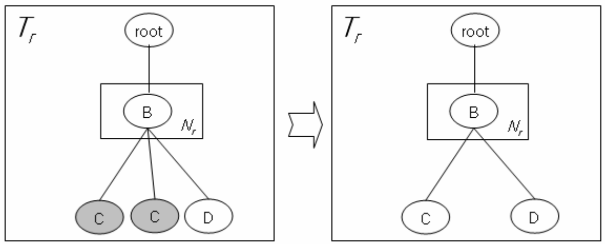


Figure 11. Unifying the split child nodes

The above overall retrieval procedure of *GET(T,e,P,S)* can be summarized by the following algorithm *RETRIEVE(D,e,P,S)* where *T* is the XML tree of the given XML document *D*. The following algorithm *UNIFY(e)* is called to unify the split child nodes of node *e*.

Algorithm *RETRIEVE(D, e, P, S)*
<Input>
1. *D* : An XML document from which data is to be retrieved
2. *e* : An element of *D*
3. *P*: A condition that can be satisfied by an element of *D*
4. *S*: one of three values, *ALL*, *FIRST* and *LAST*

<Output>
1. N_r: A node list that is the same result of the *GET(T, e, P, S).*

```
Declare $T_r$ as an XML Tree having only a root node;
Declare $N_r$ as an empty node list;
Declare DP as a general- purpose DOM parser;
Generate a sequence of multi-small XML documents <$D_1D_2...D_n$> by using
algorithms PARTITION and PAD on D;
For each $D_i$ from i=1 to n
  $T_i$ := Build an XML tree from $D_i$ by using DP;
  $N_i$ := Execute GET($T_i$, e, P, S) by using DP;
  For each node e of $N_i$
    If ((e is a cut element) or (e is a dummy element))
      $e_t$ := Find the original element of e from $N_r$ by using the identification
          attribute of e;
      If ($e_t$ exists)
        Copy the contents of e into $e_t$ by using DP;
      Else
        Copy e as a child of $T_r$ by using DP;
        Add e into $N_r$ using P;
      End if
    Else
      Copy e as a child of $T_r$ by using DP;
      Add e into $N_r$ by using DP;
    End if
  End for
End for
Remove cut attributes from every node e of $N_r$;
Return $N_r$;
```

Algorithm *UNIFY(e)*
<Input>
1. *e: An element*
<Output>
1. *Unified e: An element having no dummy and cut child elements*

```
For each child element $e_c$ of e
  If (($e_c$ is a cut element) or ($e_c$ is a dummy element))
    $e_t$ := Find the original element of $e_c$ from all child elements of e
        by using the identification attribute of $e_c$;
    If ($e_t$ exists)
      Copy the contents of $e_c$ into $e_t$;
      Remove the link from e to $e_c$;
    End if
  End if
End for
Return e;
```

4. EXPERIMENTS

In this section, the performance of the proposed DOM method to retrieve data from a very large XML document is evaluated in terms of comparing the processing time required to execute a number of GET operations on some sample XML documents [2, 20]. We chose XML documents from two different categories as sample XML documents for performance evaluation: (1) various-sized XML documents generated artificially for performance evaluation purpose only, (2) very large XML documents which are used actively and also downloadable from the Internet without any permission required.

The hardware platform for experiments using sample XML documents was exactly the same computer with the same configuration, which was used for checking DOM parsing speed in section 3. We implemented a prototype of the proposed method using JDK 5, and chose Xerces-J (version 2.9.0) as a general-purpose DOM parser. We used Xerces-J for building XML trees of the generated XML documents and for executing GET operations on the XML trees.

Case 1: Retrieving data from various-sized XML documents

The purpose of using various-sized XML documents as sample XML documents was for checking the performance of our algorithms as the number of fragments was changed. For generating these various-sized XML documents, we used xmlgen [20], because this software is used in many XML related researches as a tool for generating XML files which can be used as sample XML files for evaluating performances. We generated a set of XML documents from 100 MB to 1 GB in increments of 100 MB using xmlgen. Figure 12 shows the common structure of every XML document generated by xmlgen.

We evaluated the processing time required to execute GET operations, which are expressed with DOM API, on each of sample XML documents. For choosing meaningful GET operations, we referred the twenty queries from XMark [21] because these queries are used widely in many research papers for benchmark performance on the files generated by xmlgen.

These queries are expressed with XQuery [22]. Any XQuery processor using DOM API to access XML documents executes a number of GET operations for finding nodes, which are specified in any given query. Therefore, in the viewpoint of checking the performance of GET operations, there are no critical differences in these queries. So, we chose two queries that need relatively short execution time among the queries of Q1~Q20 for evaluation of our method: Q1 (*return the name of the person with ID 'person0'*) and Q6 (*return the number of items, which are listed on all continents*). For experiments, we implemented two XML applications which call a number of GET operations for finding nodes specified in the Q1 and Q6.

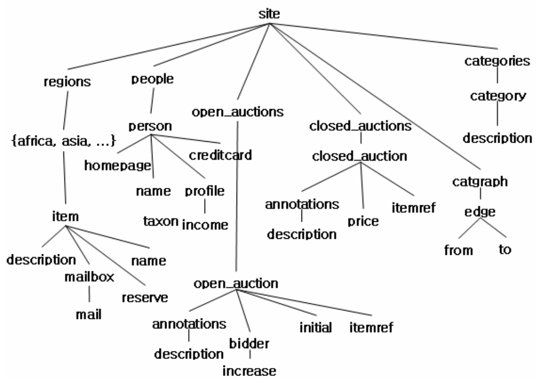


Figure 12. Structure of an XML Document generated by xmlgen

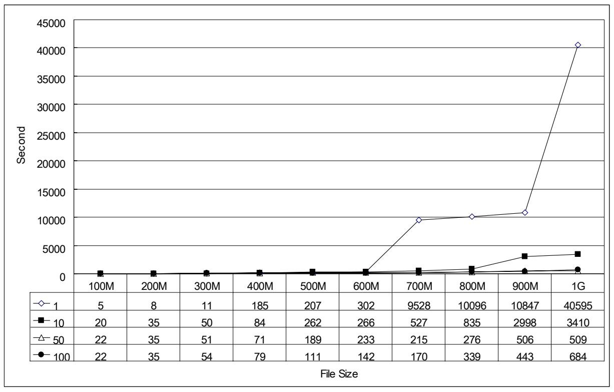
Figure 13 and 14 show the execution time of Q1 and Q6 on 10 sample XML documents correspondingly. In checking the executing time of both queries, we first computed the execution time on non-partitioned XML documents, and we computed the execution time on 3 partitions with 10, 50, and 100 fragments for every XML document. We performed every experiment on each XML document 5 times repeatedly to get the average time as an execution time. In these figures, the time required for the partitioning and padding steps is included in the time on each partitioned XML document.

In Figure 13, when an XML document is smaller than 400 MB, the execution time of Q1 on non-partitioned XML documents is shorter than on any-partitioned XML documents. This is due to the overheads caused by the partitioning, padding and combining their results. But, when the size of an XML document increases to the 1 GB, there are steep increases of execution time on non-partitioned XML documents. This is the same pattern as the results described in section 3. In partitioned XML documents, however, there are relatively gradual increases of execution time. Figure 14 shows almost the same pattern of execution time increasing shown in Figure 13.

The results of these experiments show that the method suggested in this paper can reduce the time required for retrieving data from very large XML documents, which were very ineffective to be processed with general-purpose DOM parsers. For example, the execution time of Q6 on the non-partitioned 1 GB XML document is 253,381 seconds, i.e. almost 3 days. It is actually meaninglessness. However, the time required for executing the same query on the partition of 100 fragments is reduced to 205 seconds, a manageable level by one single general-purpose personal computer.

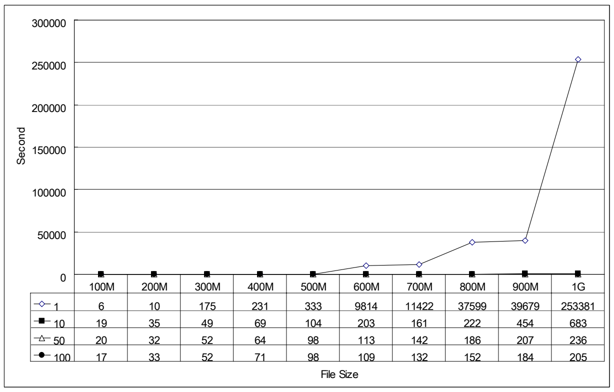
Case 2: Retrieving data from XML documents containing Protein Sequence Database

For checking the efficiency of our method in the fields where a number of very large XML documents exist, we used two XML documents from UniProt [2] as sample XML documents. UniProt is the consortium for supporting biological research. They maintain high quality protein sequence databases. We downloaded two XML documents from the web site of UniProt: uniref100.xml and uniprot_trembl.xml. The properties of these two files are summarized in Table 2, and the structure of uniprot_trembl.xml is described in Figure 15 (the structure of uniref100.xml is a simplified version of the structure of uniprot_trembl.xml). These two files are the aggregations of protein sequences, a subtree under an 'entry' node represent a protein sequence.



	100M	200M	300M	400M	500M	600M	700M	800M	900M	1G
◇ 1	5	8	11	185	207	302	9528	10096	10847	40595
■ 10	20	35	50	84	262	266	527	835	2998	3410
△ 50	22	35	51	71	189	233	215	276	506	509
● 100	22	35	54	79	111	142	170	339	443	684

Figure 13. Execution of Q1 on various-sized XML Documents from 100 MB to 1 GB in increments of 100 MB



	100M	200M	300M	400M	500M	600M	700M	800M	900M	1G
◇ 1	6	10	175	231	333	9814	11422	37599	39679	253381
■ 10	19	35	49	69	104	203	161	222	454	683
△ 50	20	32	52	64	98	113	142	186	207	236
● 100	17	33	52	71	98	109	132	152	184	205

Figure 14. Execution of Q6 on various-sized XML Documents from 100 MB to 1 GB in increments of 100 MB

Table 2. Sample XML Documents from the Biological Information Area

File Name	File Size (MB)	# of Elements
uniref100.xml	4,518	53,523,902
uniprot_trembl.xml	12,999	242,506,824

Figure 16 and 17 show the time required for retrieving data with GET operations of DOM API from uniref100.xml, uniprot_trembl.xml correspondingly. First, we tried to retrieve data on these two files without applying our method. Next, we tried to retrieve data on partitioned XML documents while increasing the number of fragments. Like the experiments of case 1, we performed every experiment with same condition 5 times repeatedly to get the average time as the time required for retrieving data. We forced a process of retrieving to be stopped when that process did not finish in a day.

In Figure 16, the time required for retrieving the names of every protein from uniref100.xml is shown. DOM parsing failed on non-partitioned uniref100.xml because of the shortage of memory. When partitioning was applied, it did not finish in a day when the number of fragments is less than 40. As the number of fragments increase from 40 to 100, we can see the time required for retrieving data is stabilized less than 1 hour (3,600 seconds).

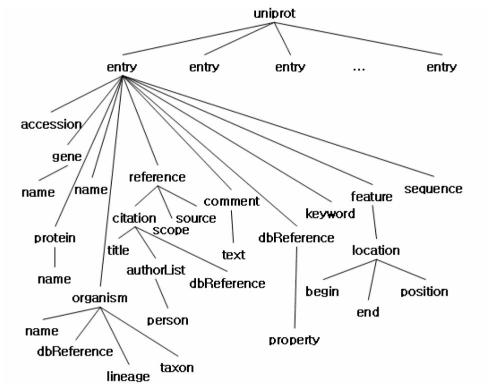
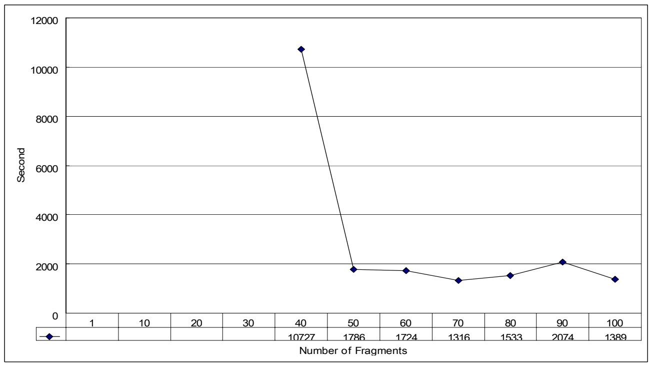


Figure 15. Structure of uniprot_trembl.xml



	1	10	20	30	40	50	60	70	80	90	100
◆					10727	1786	1724	1316	1533	2074	1389

Figure 16. Retrieving Gene Names from uniref100.xml

In Figure 17, the time required for retrieving the names of every protein from unprot_trembl.xml is shown. The size of unprot_trembl.xml is almost four times bigger than the size of uniref100.xml. Therefore it was clear that DOM parsing failed on non-partitioned unprot_trembl.xml due to the shortage of memory. When partitioning was applied on unprot_trembl.xml, it did not finish in a day while the number of fragments is less than 150. As the number of fragments increase to 500, we can see the time required for retrieving data is going to down near 1 hour.

We also checked the ratio of the time required for generating small XML documents to the time required for retrieving data using these XML documents. In Figure 18, the result shown in Figure 16 is reorganized using two categories: P&P and R. P&P (abbreviated from the partitioning and padding) represents the time required for generating small XML documents, and R (abbreviated from the retrieving) represents the time required for retrieving data using the generated XML documents.

Figure 18 shows that the time required for generating small XML documents was not changed widely during the number of fragments increased from 40 to 100. However, the time required for retrieving data using the generated XML documents dropped sharply when the number of fragments increased from 40 to 50. This demonstrates that, in spite of the costs due to the generation of small XML documents, there is a possibility to enhance the performance of DOM parsing by the using of these small XML documents.

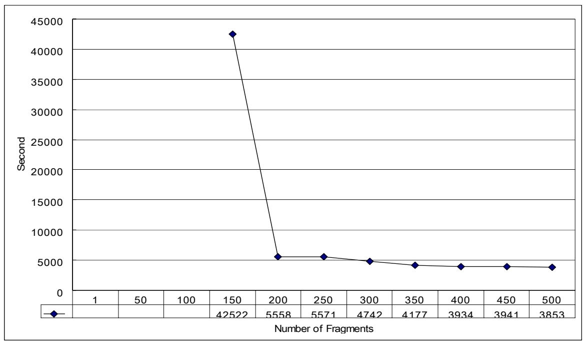


Figure 17. Retrieving Gene Names from uniprot_tembl.xml

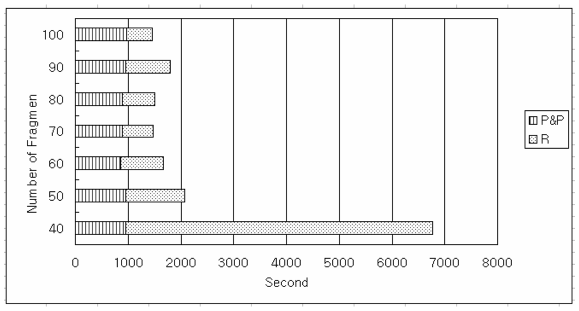


Figure 18. Ratio of the time required for generating fragments to the time required for retrieving data using fragments

5. CONCLUSIONS

In this paper, we presented a method to retrieve data from a very large XML document with DOM. When an XML document is very large and various operations for finding elements are to be executed on this XML document frequently, the method presented in this paper can reduce the execution time of these operations on the XML document to the level of usefulness. In the future, we plan to conduct a study on the automatic and dynamic selections of (1) the number of fragments to be generated and (2) cut points without the help of the external variables, currently given by users, and to extend this method to support INSERT, DELETE, and UPDATE operations on very large XML documents as well.

6. REFERENCES

[1] Extensible Markup Language (XML) 1.0 (Third Edition), http://www.w3.org/TR/2004/REC-XML-20040204/

[2] UniProt, http://www.uniprot.org/database/download.shtml

[3] Apache Xerces, http://xerces.apache.org/

[4] Document Object Model (DOM) Level 3 Core Specification, http://www.w3.org/TR/DOM-Level-3-Core/

[5] SAX: A Simple API for XML, http://www.saxproject.org

[6] Markus L. Noga, Steffen Schott, Welf Löwe, XML manipulations: Lazy XML processing, In *Proceedings of the 2002 ACM symposium on Document engineering*, 2002, 88-94.

[7] Chia-Hsin Huang, Tyng-Ruey Chuang, Hahn-Ming Lee, Prefiltering Techniques for Efficient XML Document Processing, In *Proceedings of the 2005 ACM Symposium on Document Engineering*, 2005, 149-158.

[8] XML Path Language (XPath), http://www.w3.org/TR/xpath

[9] A. R. Schmidt, Florian Waas, Martin L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, R. Busse, The XML benchmark project, Technical Report, CWI, 2001.

[10] Hongjun Lu, Jeffrey Xu Yu, Guoren Wang, Shihui Zheng, Haifeng Jiang, Ge Yu, Aoying Zhou, What makes the differences: benchmarking XML database implementations, *ACM Transactions on Internet Technology*, 5, 1 (February 2005), 154-194.

[11] Timo Böhme, Erhard Rahm, Multi-user Evaluation of XML Data Management Systems with XMach-1, In *Proceedings of the VLDB 2002 Workshop EEXTT and CAiSE 2002 Workshop DTWeb on Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web-Revised Papers*, 2003, 148-158.

[12] H. Jiang, H. Lu, W. Wang, J.X. Yu, XParent: An Efficient RDBMS-Based XML Database System, In *Proceedings of the 18th International Conference on Data Engineering*, 2002, 335-336.

[13] Albrecht Schmidt, Martin L. Kersten, Menzo Windhouwer, Florian Waas, Efficient Relational Storage and Retrieval of XML Documents, *Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*, 2000, 137-150.

[14] M. Yoshikawa, T.Amagasa, T.Shimula and S.Uemura, XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases, *ACM Transactions on Internet Technology*, 1, 1 (August 2001), 110-141.

[15] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, Jeffrey F. Naughton, Relational Databases for Querying XML Documents: Limitations and Opportunities, In *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999, 302-314.

[16] Aoying Zhou, Hongjun Lu, Shihui Zheng, Yuqi Liang, Long Zhang, Wenyun Ji, Zengping Tian, VXMLR: A Visual XML-Relational Database System, In *Proceedings of the 27th International Conference on Very Large Data Bases*, 2001, 719-720.

[17] Tian, F., DeWitt, D. J., Chen, J., and Zhang, C. The design and performance evaluation of alternative XML storage strategies. *Tech. rep., Computer Science Department*, University of Wisconsin, Madison, WI, 2000.

[18] Kentarou Kido, Toshiyuki Amagasa and Hiroyuki Kitagawa, Processing XPath Queries in PC-Clusters Using XML Data Partitioning, In *Proceedings of the 22nd International Conference on Data Engineering Workshops*, 2006, 11-16.

[19] Wei Lu, Kenneth Chiu and Yinfei Pan, A Parallel Approach to XML Parsing, In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, 2006, 223-230.

[20] XMark, http://monetdb.cwi.nl/xml/generator.html

[21] XMark Benchmark Queries, http://www.ins.cwi.nl/projects/xmark/Assets/xmlquery.txt

[22] XML Query (XQuery), http://www.w3.org/XML/Query

[23] XML Namespace, http://www.w3.org/TR/REC-xml-names/