Computing and Informatics, Vol. 31, 2012, 639–664

A METHOD OF XML DOCUMENT FRAGMENTATION FOR REDUCING TIME OF XML FRAGMENT STREAM QUERY PROCESSING

Jin Kim, Hyunchul Kang

School of Computer Science and Engineering
Chung-Ang University
Seoul, 156-756, Korea
e-mail: jkim@dblab.cse.cau.ac.kr, hckang@cau.ac.kr

Communicated by Jacek Kitowski

Abstract. As XML has been established as the standard for data exchange not just on the Web but among heterogeneous devices, systems, and applications, effective processing of XML queries is one of core components of ubiquitous computing. Most of the mobile/hand-held devices deployed in ubiquitous computing environment are still limited in memory and processing power. An effective query processing is required when the source XML document is of large volume. The framework of *fragmenting* an XML document and *streaming* the XML fragments for query processing at the mobile devices has received much attention. However, the main focus was on the *memory efficiency* to cope with the memory constraint in the mobile devices. Query processing time might be compromised in those techniques. Since the processing power is also limited in the mobile devices, the *time optimization* deserves attention. We have found out that the query processing time is significantly affected by how the source XML document is fragmented. In this paper, we propose a method of XML document fragmentation whereby query processing gets efficient in time while the size constraint for each resulting fragment is satisfied. Through implementation and a set of detailed experiments, we show that our proposed method considerably outperforms other methods.

Keywords: XML, XML document fragmentation, XML fragment stream, XML fragment labeling, XML fragment stream query processing, ubiquitous computing

1 INTRODUCTION

XML has originally emerged as the standard for data exchange and representation on the Web but now it is widely adopted as a means of data exchange and representation among heterogeneous devices, systems, and applications. This, in turn, has resulted in higher expectation for a technology that can effectively process XML queries in those resource-constrained mobile devices, because effective XML query processing should be a core component in ubiquitous computing environment.

In [6, 9], a framework called *XStreamCast* was proposed whereby XML document gets *fragmented* at the sever and *streamed* to a number of *client* mobile devices through wireless broadcast, then each client receives the *XML fragment stream* and processes queries over the stream. In [4, 5, 10, 11, 12, 13], techniques of XML fragment stream query processing were proposed based on the framework of XstreamCast. In [5], a method called *XFrag* was proposed in which the *hole-filler model* [6, 9] is employed to describe the structural relationship among the XML fragments. With the hole-filler model, each XML fragment could contain *holes*, which are supposed to be *filled* with other XML fragments possibly with other holes. The historical XML fragment management with the hole-filler model in a dynamic environment was investigated in [4]. In [11], *XFPro*, an improved version of XFrag, was proposed. It was also based on the hole-filler model. In [12, 13], *XML fragment labeling(XFL)* which is based on the XML labeling schemes [14, 17] was proposed as a new method of representing XML fragmentation to replace the hole-filler model in the previous work. The hole-filler model is popular because of its simplicity in representing XML fragmentation but has fundamental limitations as investigated in [12, 13] where it was shown that XFL was much more effective for memory-efficient query processing. With XFL, it was shown that large and/or dynamic XML fragment stream can also be dealt with in memory-efficient way and that stream query processing is *scalable* with limited memory as the stream size increases [13].

These techniques, however, have focused on memory efficiency. Query performance in *time* has received little attention. So has the *XML fragmentation method*. Query processing would take longer as the size of the XML fragment stream gets larger. This is natural because query processing is not over until the whole stream is received. Under such an inherent constraint, we note that query processing time is significantly affected by how the source XML document is fragmented. A method of fragmenting an XML document into XML fragments is a core component of XML fragment stream query processing. In fragmenting an XML document, its nodes are clustered to form XML fragments. For a large XML document, there could be virtually an infinite number of different fragmentations possible depending on which part of the XML document is assigned to which fragment.

In this paper, we address the problem of fragmenting an XML document into XML fragments such that query processing gets efficient in time while memory efficiency is sustained. We develop a *cost model* of query processing over XML fragment stream, and propose a method of fragmenting an XML document. The novel and important features of our method are:

- It employs a *cost-based* fragmentation where the cost model of XML fragment stream query processing plays a key role.
- It deals with *multiple queries* that would be in conflict with each other when it comes to the fragmentation of their common source XML document. One particular fragmentation could not be effective for all the queries. It tries to minimize the average query processing time for the entire set of queries running in all the client devices.
- It takes *query frequencies* into account such that the derived fragmentation would be favorable for the *hot* queries rather than for cold queries. However, the ad-hoc queries not in the given set of queries are *not* excluded.
- One of its input parameters is the *upper size limit* of each XML fragment. No matter how efficient a fragmentation method is, it is useless and impractical unless the size of every resulting fragment is less than that of the buffer allocated for receiving the fragments at the client devices.

The rest of this paper is organized as the follows: Section 2 describes the background material for this paper. Section 3 proposes a method of XML document fragmentation given a set of queries and their frequencies. It also describes our cost model of XML fragment stream query processing. Section 4 describes the implementation and reports experimental results with our proposed method and other methods in the literature. Section 5 gives related work. Finally, Section 6 concludes the paper.

2 BACKGROUND

In this section, we give a brief description of XML fragment stream query processing as a background of this paper. Part of this section is a revised version of the excerpt from our earlier paper on memory efficiency of XML fragment stream query processing [13]. In XML fragment stream query processing, each client processes the fragment stream, one fragment at a time, rather than receiving and storing the full stream of XML fragments to reconstruct the original XML document before processing a query. The fragments could be received in different order than they were broadcast at the server. The components of XML fragment stream query processing include *tag structure* [6, 9], *XML fragment labeling* [12, 13], and *query pipeline* [5].

2.1 Tag Structure

As Figures 1 a) and b) show, XML data can be modeled as a node-labeled tree. The tag structure is to summarize the structure of the XML tree and also to specify how it is fragmented. It is specified in XML. Figure 1 c) shows the tag structure of the XML tree of Figure 1 b). The *id* attribute is the *tag structure ID (tsid)* assigned to each tag that occurs on a path in the XML tree. Their values are uniquely given.

In Figure 1 c), there are 4 tags specified, 'a' through 'd', and 10 through 13 are respectively assigned as their unique *tsid*'s. As for the tag 'b', there are 2 instances in the XML tree of Figure 1 b). They are considered the same tag, because both appears on the same path from the root of the tree (i.e., '/a/b'). The same for the tags 'c' and 'd'. Meanwhile, the *name* attribute stores the tag name, and the value of attribute *filler* being "true" indicates that the corresponding tag is the root of an XML fragment. Figure 1 d) shows an example of fragmentation for the XML tree of Figure 1 b). The region marked by a triangle denotes an XML fragment. This fragmentation conforms to the specification of fragmentation in the tag structure of Figure 1 c).

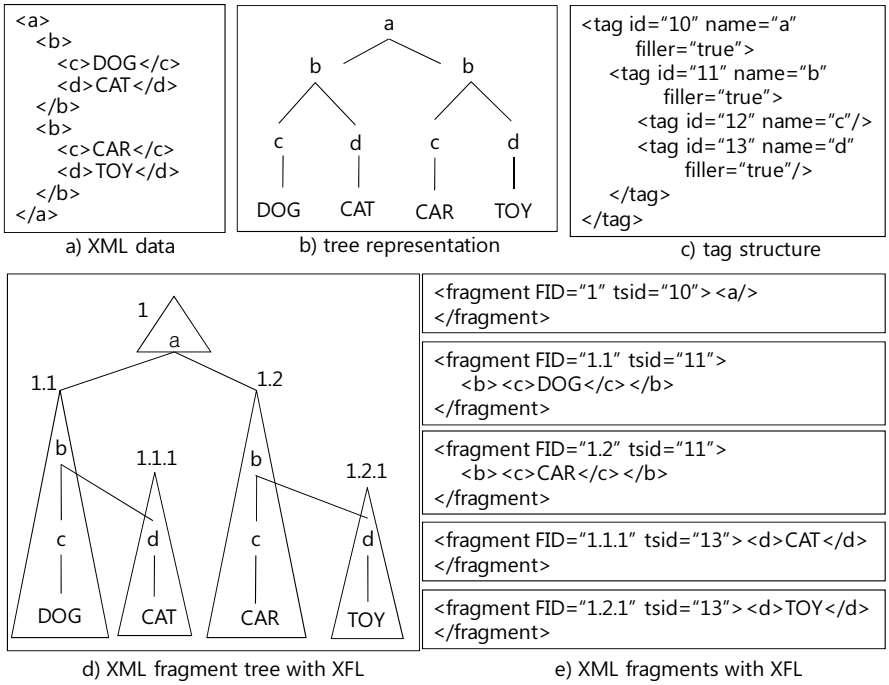


Fig. 1. Example of XML fragmentation

2.2 XML Fragmentation with Fragment Labeling

The conventional XML node labeling was devised to represent structural relationship (e.g., parent-child, ancestor-descendant, etc.) among the nodes of XML data modeled as a tree, and is exploited in the structural joins for XML query processing. In the *vertical* fragmentation of an XML tree, each of the resulting fragments is a subtree of the original XML tree. Thus, the relationship among the fragments

could also be represented as a tree where a node is an XML fragment. We call it an *XML fragment tree*, an example of which is shown in Figure 1 d). Thus, its fragments could be labeled with *any* of the conventional XML labeling schemes in the same way as the nodes of the original XML tree are labeled. Figure 1 d) shows XML fragment labeling (XFL) of the XML fragment tree where *Dewey order encoding* [18] is employed as an XML labeling scheme. There are 5 fragments whose labels are 1, 1.1, 1.2, 1.1.1, and 1.2.1. They are given in XML in Figure 1 e). The top-level element of each fragment, which we call the *header element*, has two attributes *FID* (fragment ID) and *tsid*. The value of FID is the label assigned with XFL. In the fragment 1, the original element 'a' is enclosed by the header element, resulting in *<fragment FID="1" tsid="10"><a/></fragment>*. Note that the original body of element 'a' is now null, because the whole of it (<c>DOG</c><d>CAT</d><c>CAR</c><d>TOY</d>) is taken out into its child fragments whose FIDs are 1.1, and 1.2. Meanwhile, in the fragment 1.1, the original body of the first instance of element 'b', <c> DOG </c> <d> CAT </d>, is replaced by <c> DOG </c> alone without <d> CAT </d>, which is taken out into the fragment 1.1.1. The same pattern of fragmentation occurs in the fragments 1.2. The structural relationship among the fragments can be identified due to XML labeling used for the FIDs. For complete description of XML fragment labeling, refer to [13].

2.3 Query Pipeline

In this paper, we consider XPath expressions for XML queries. Consider an XML fragment tree in Figure 1 d), the tag structure of which is given in Figure 1 c). For an XPath expression, it is transformed into a *query pipeline* where each *location step* [8] of the XPath expression is an operator. For example, XPath expression '/a/b[c='CAR']/d' against the XML tree of Figure 1 b) is transformed into a query pipleline shown in Figure 2 where each box is an operator and the value in parentheses beside each operator is the *tsid* of the tag that corresponds to the operator as defined in the tag structure.

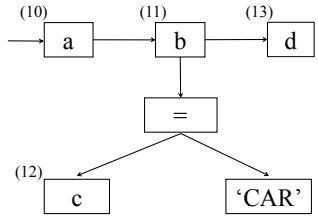


Fig. 2. Query pipeline for '/a/b[c='CAR']/d'

For query processing, the tag structure is delivered to the client first. When each fragment arrives at the client, at least some information on it (if not the fragment

itself) may need to be kept in the memory of the client for query processing. We call it *fragment information*. In principle, to process a query over an XML fragment stream, some form of bookkeeping of the information on the streamed fragments is essential because the whole of the source XML document would not be reconstructed and the fragment itself just streamed in would be discarded soon. For further details in processing the query '/a/b[c='CAR']/d' (whose result is <d>TOY</d>), refer to Section 2.3 of [13].

3 XML DOCUMENT FRAGMENTATION

3.1 Overview

In this section, we propose a method of XML document fragmentation whereby query processing time is reduced. First, we note that there are too many ways of fragmenting a given XML document. It can grow *exponentially* to the number of different *path types* existing in the document. For example, in the XML tree of Figure 1 b), there are 4 different path types, '/a', '/a/b', '/a/b/c', '/a/b/d', and we can come up with 8 different fragmentations when considering only *vertical* fragmentation. For each path type except the one with the top-level element (i.e., '/a'), its terminal element (e.g., 'c' for '/a/b/c') can *either* be the root of a fragment that includes itself and all/some of its descendent nodes *or* belong to the fragment where its parent node belongs to. Therefore, if there are n different path types in a document, 2^{n-1} vertical fragmentations are possible. For the XML tree of Figure 1 b), $n = 4$ and thus, there are 8 different fragmentations. In reality, n would be very large. For example, there exists 502 different path types in the *auction.xml* document of 10.8MB created using *xmlgen* in XMark benchmark [19], and thus, 2^{501} different fragmentations are possible. It is infeasible to check the efficiency of every possible fragmentation in order to find the optimal one for a given set of queries and their frequencies. Thus, we use a *heuristic* approach to narrow down the search space. Our heuristic is based on the *XPath step reduction* [13] whereby the query pipeline operates efficiently both in time and space.

Secondly, there should be the *upper limit* on the size of every XML fragment. It is given as an input parameter of the method of XML fragmentation, and should be *smaller* than the size of the buffer allocated for receiving a fragment at a client. It can be determined considering the memory capacity of the deployed client devices. XML fragmentation based solely on query patterns and frequencies would not meet this requirement.

Given an XML document, its tag structure, a set of XPath queries, their frequencies, and the upper limit of the fragment size, our proposed XML fragmentation is conducted in the following steps:

1. *document analysis*,
2. *size-constrained fragmentation*,

3. *cost and frequency-based fragmentation,*
4. *physical fragmentation.*

In document analysis, several statistics about the XML document are gathered while it is parsed. In size-constrained fragmentation, a preliminary fragmentation is derived such that the size of every fragment is less than the given upper limit of the fragment size. In this and the next steps, the XML fragments themselves are not yet produced. Rather, just the tag structure of the XML document is modified so that the fragmentation derived thus far is specified with the relevant filler attributes set to "true". The physical fragmentation of the XML document is done after the final tag structure with the full specification of fragmentation is produced. In cost and frequency-based fragmentation, which is the main step for our fragmentation, the tag structure out of the size-constrained fragmentation undergoes a sequence of modifications to take into account the processing cost and frequency of all the client queries. Memory efficiency, that is, the space cost of XML fragment query processing was thoroughly dealt with in the previous work [5, 11, 12, 13]. In this paper, we focus on its *time* cost, the time it takes for query processing. In this step, both vertical and horizontal fragmentations are considered. The tag structure is modified such that a set of XML nodes are to be taken out together, forming a separate fragment. To perform this step, we developed a *cost model* of XML fragment stream query processing.

Statistic	**Description**	**Notation in Cost Model (subsection 3.4.2)**
The Number of Path Instances	The number of instances of P in the document	$I(P)$
The Number of Elements	The number of elements in the subtree of the tag structure whose root is the terminal node of P	$E(P)$
Total Subtree Size	The total sum of the byte size of the subtrees whose root is the terminal node of P	–
Average Subtree Size	The average byte size of the subtrees whose root is the terminal node of P	–

Table 1. Statistics for each path type P in the XML document

3.2 XML Document Analysis

While parsing an XML document D, we gather the statistical information for each path type P appearing in D as listed in Table 1. The set of these path types is equal

to the set of path types appearing in the tag structure of D. The terminal node of each path type P could be the *root* of an XML fragment when vertical fragmentation is considered. For example, let us consider the XML tree in Figure 1 b). There are 4 different path types: '/a', '/a/b', '/a/b/c', and '/a/b/d'. In the fragmentation of Figure 1 d), the terminal nodes of three path types ('/a', '/a/b', and '/a/b/d') are the root of a fragment. For these path types, there are respectively 1 ('/a'), 2 ('/a/b'), and 2 ('/a/b/d') instances in Figure 1 b). It means that there are 1, 2, and 2 fragment instances whose root is at '/a', '/a/b', and '/a/b/d', respectively (Figure 1 d)). The number of path instances for each path type P, denoted as $I(P)$, is necessary in the fragmentation process. Additional information for each path type P includes the number of elements in the subtree whose root is the terminal node of P, denoted as $E(P)$ and the byte size of the subtree whose root is the terminal node of P.

3.3 Size-Constrained Fragmentation

Size-constrained fragmentation is to ensure that each resulting fragment is smaller than the given upper limit of the fragment size. Figure 3 shows *Algorithm 1* which conducts this fragmentation. To minimize the *relevance checking time* incurred during query processing, which will be described in the next subsection, it tries to come up with as large a fragment as possible under the fragment size constraint. Since the tag structure is a tree, invoking Algorithm 1 with its root as input would traverse it to derive the size-constrained fragmentation. For a node N of the tag structure, let $T(N)$ denote the subtree whose root is N. Given a node N of the tag structure, Algorithm 1 terminates if the size of $T(N)$ satisfies the size constraint (line 2). Otherwise, it recursively calls itself for each child of N (lines 3–6). If the size constraint is not yet satisfied for $T(N)$ after the recursive calls, it takes out the subtrees of N as separate fragments one by one from the largest to the smallest until $T(N)$ meets the size constraint (line 7-13). $T(N)$ shall eventually satisfy the size constraint, because its size gets reduced as the subtrees of N are taken out as separate fragments. In a recursive manner, eventually all the fragments shall satisfy the size constraint.

When $T(N)$ is taken out as a separate fragment, the filler attribute of N in the tag structure is set to "true" to mark such a fragmentation. Figure 4 describes *Algorithm 2* that performs that (line 17). It is invoked at line 10 of Algorithm 1. It also updates the statistics maintained for all the affected subtrees in the tag structure (lines 4–16). When $T(N)$ is taken out from the subtree T containing $T(N)$, the size of T is reduced by that of $T(N)$. The number of descendants of T is also reduced by that of $T(N)$. Such deductions are done for every subtree T that contains $T(N)$. Every node in the path type from the root of the tag structure to the parent of N is the root of such an affected subtree T.

Algorithm 1: Size-constrained Fragmentation

```
1: procedure fragmentBySize(node)
2:   if (node.getSubtreeSize() <= fragmentSizeLimit) return;
3:   children := node.getChildrenSortedBySize();
4:   for each child in children do
5:     fragmentBySize(child);
6:   endfor
7:   if (node.getSubtreeSize() > fragmentSizeLimit) then
8:     children := node.getChildrenSortedBySize();
9:     for each child in children do
10:        markFragUpdateStat(child);
11:        if(node.getSubtreeSize() <= fragmentSizeLimit) break;
12:    endfor
13:  endif
14: endprocedure
```

Fig. 3. Algorithm for size-constrained fragmentation

Algorithm 2: Mark of Fragmentation and Update of Statistics

```
1: procedure markFragUpdateStat(node)
2:   root := node.getFragmentRoot();
3:   if (node == root) root := NULL;
4:   if (root != NULL) then
5:      numElements := node.getNumElements();
6:      byteSize := node.getByteSize();
7:      parentOfRoot := root.getParent();
8:      p := node.getParent();
9:      while(p != parentOfRoot) do
10:        pNumElements := p.getNumElements();
11:        pByteSize := p.getByteSize();
12:        p.setNumElements(pNumElements - numElements);
13:        p.setByteSize(pByteSize - byteSize);
14:        p := p.getParent();
15:     endwhile
16:  endif
17:  node.setFillerAttribute("true");
18: endprocedure
```

Fig. 4. Algorithm for marking fragmentation and updating statistics

3.4 Cost and Frequency-Based Fragmentation

3.4.1 Performance Trade-Off

In fragmenting an XML document, if the size of each fragment increases, the total number of fragments decreases. If the size gets smaller, there would be more fragments. As an extreme example, if the whole document forms a single fragment, the size of the fragment would be the document size plus the size of the header element. In the other extreme, if each element of an XML document forms a fragment on its own, the size of a fragment would be equal to the size of an XML element plus the size of the header element, but the number of fragments would be equal to the number of elements in the document. The number and the size of fragments *matter* in query performance as described below.

When a fragment f is received at a client, it is first checked whether f is relevant to any of the queries running at that client. *Relevance checking* is done by referring to the *tsid* attribute of the header element of f and by consulting the tag structure. If irrelevant, f is discarded right away. If relevant, f is passed through the relevant query pipeline operators, which *navigate* f to extract and store the information for query processing. The time for fragment stream query processing consists of the *relevance checking time* for *all* the fragments and the *navigation time* for the *relevant* fragments.

Increasing the number of fragments would proportionally increase the relevance checking time. However, reducing it would increase the average size of relevant fragments, increasing the navigation time. There exists a performance trade-off between the relevance checking time and the navigation time. The key variable in this trade-off is the size of the fragment. To reduce query processing time, (1) the size of the irrelevant fragments needs to be as large as possible, and (2) the size of the relevant ones needs to be as small as possible. However, such ideal fragment size neglects memory efficiency in query processing. A large fragment would occupy more memory while it is received and processed. We already mentioned that there should be the upper limit of the fragment size. Thus, even for the irrelevant fragments, their size is constrained to some extent. As for the size of a relevant fragment, the minimum size might not be the best, either. Rather, it is better to put the XML nodes that are inter-related in the query to the same fragment for memory efficiency [13]. However, such a practice might result in large fragments and/or they might be too large to meet the upper limit of the fragment size. In all, we need more objective criteria on the fragment size. That led us to develop a cost model of XML fragment stream query processing.

3.4.2 Cost Model

We develop a cost model of XML fragment stream query processing based on the performance trade-off between the relevance checking time and the navigation time. Our aim is to derive a formula that computes the query processing cost, $c(q, F(D))$,

given an XPath query q and a fragmentation $F(D)$ of its source XML document D. Let n denote the total number of fragments in $F(D)$, and m be the number of fragments relevant to q as determined by relevance checking. (That is, $n - m$ fragments are discarded right away after their relevance checking.) Let $f_1, \ldots, f_n$ denote all those n fragments, and $f_{r_1}, \ldots, f_{r_m}$ denote those m fragments relevant to q where $1 \leq r_j \leq n$ and $1 \leq j \leq m$. $c(q, F(D))$ can be given in Equation (1).

$$c(q, F(D)) = \sum_{i=1}^{n} T_R(f_i) + \sum_{j=1}^{m} (T_N(f_{r_j}) + T_P(f_{r_j})) \tag{1}$$

where $T_R(f_i)$ is the relevance checking time for fragment f_i, $T_N(f_{r_j})$ is the navigation time for fragment f_{r_j}, and $T_P(f_{r_j})$ is the time for conducting the remaining tasks of query processing with the fragment information extracted out of fragment f_{r_j} (e.g., storing the fragment information, evaluating the query predicates if any, checking parent-child/ancestor-descendant relationship among XML nodes, producing the query results, etc.).

For each fragment f, its relevance to a query q is checked with the *tsid* attribute of the header element of f. With that attribute value, the tag structure, and with the query pipeline of q, whether or not f is relevant to q can be easily determined. To expedite this process, the tag structure and the query pipeline could be preprocessed beforehand to derive the data structures that can efficiently support relevance checking. In that way, $T_R(f)$ can be approximated as taking some constant time for each fragment f.

For each fragment f, its navigation time is dependent on the size of f, which is, in turn, correlated to the number of XML elements in f. While the relevance checking of f deals with only 1 element (i.e., the header element of f), the navigation of f goes through all the elements in f. If we *normalize* the time cost such that $T_R(f) = 1$ for each fragment f, $T_N(f)$ can be estimated to be equal to the number of elements in f. Meanwhile, $T_P(f)$ would vary from fragment to fragment. We approximate it as taking some constant time K for each fragment relevant to q. In all, the normalized approximation of the query processing cost, $\hat{c}(q, F(D))$, can now be given as in Equation (2) from Equation (1)

$$\hat{c}(q, F(D)) = n + \sum_{j=1}^{m} e(f_{r_j}) + m \times K \tag{2}$$

where $e(f_{r_j})$ denotes the number of elements in fragment f_{r_j}. For a fragment f, $e(f)$ is obtained in the document analysis, because for every path type P in the tag structure, the number of elements in the subtree whose root is the terminal node of P is obtained. In Table 1, we denoted it as $E(P)$ while we denoted the number of instances of P in the document as $I(P)$. (Note that $I(P)$ is also obtained in the document analysis.) Given a vertical fragmentation $F(D)$ of an XML document D, let $\alpha(F(D))$ denote the set of all the path types in the tag structure of D such that

for each $P \in \alpha(F(D))$, the terminal node of P is the root of some fragments in $F(D)$. Then, n is given as follows:

$$n = \sum_{P \in \alpha(F(D))} I(P). \tag{3}$$

Given an XPath query q and a vertical fragmentation $F(D)$ of its source XML document D, let $\beta(q, F(D))$ denote the set of path types in the tag structure of D such that for each $P \in \beta(q, F(D))$, the terminal node of P is the root of some fragments in $F(D)$ which are relevant to q. (That is, $\beta(q, F(D)) \subseteq \alpha(F(D))$ for every query q.) Then, m is given as follows:

$$m = \sum_{P \in \beta(q, F(D))} I(P). \tag{4}$$

The second term of Equation (2) is computed as follows:

$$\sum_{j=1}^{m} e(f_{r_j}) = \sum_{P \in \beta(q, F(D))} (E(P) \times I(P)). \tag{5}$$

Thus, given an XPath query q and a fragmentation $F(D)$ of its source XML document D, the approximate normalized cost of processing q over the XML fragment stream out of $F(D)$ is finally given in Equation (6).

$$\hat{c}(q, F(D)) = \sum_{P \in \alpha(F(D))} I(P) + \sum_{P \in \beta(q, F(D))} (E(P) \times I(P)) + K \times \sum_{P \in \beta(q, F(D))} I(P) \tag{6}$$

So far, we have developed a cost model for a *single* XPath query with respect to a fragmentation. Given a set of XPath queries running in all the client devices, $Q = \{q_1, \ldots, q_u\}$ with $H = \{h_1, \ldots, h_u\}$ such that h_i is the frequency of q_i, $i = 1, \ldots, u$, the *average* cost of processing each query in Q with respect to a fragmentation $F(D)$ of their source XML document D is given in Equation (7).

$$\hat{C}(Q, H, F(D)) = \sum_{i=1}^{u} (h_i \times \hat{c}(q_i, F(D))) \tag{7}$$

3.4.3 Fragmentation by XPath Step Reduction

This and next subsections describe the main features of our fragmentation. To explain this fragmentation, we first explain as background some optimization techniques in fragment stream query processing which capitalize on the tag structure. An XPath expression is a sequence of the *location steps* consisting of axis, node test, and predicate [8]. As described in detail in [13], in XML fragment stream query processing, only those steps including predicate (*filter step*) and the last step (*result step*) need to be evaluated (rather than evaluating every step in order). Such an optimization is possible due to the tag structure and is called *XPath step reduction* [13],

because not every step needs to be evaluated. A filter step is to filter the child and descendant nodes of the node currently processed that do not satisfy the predicate at hand. A result step is to deliver the query result. Let us consider an example XPath query: '/a/b[c='DOG']/d/e'. For this query, only the steps '/b[c='DOG']' and '/e' need to be evaluated. Such step reduction does not compromise the correctness of query processing due to the tag structure and the *tsid* attribute of the header element of each fragment. Note that each path type is assigned a unique *tsid*. Thus, even if two different path types, '/a/**b**' and '/a/d/**b**' for example, terminate with the same tag name (i.e., '/**b**'), their *tsid*'s are different. Therefore, in evaluating a location step of XPath against a fragment f which contains element 'b' right under the header element, it is possible to figure out whether f is at path '/a/b' or at path '/a/d/b' with the *tsid* value of f.

The fragment that contains the XML nodes that are matched by the filter or result steps of an XPath query is relevant to the query. Those nodes are to be accessed through navigation of the fragment. To reduce the navigation time, it is desired for such a node to be right under the header element of a fragment. If such a node N is not right under the header but possibly deep inside the fragment f that is generated by the size-constrained fragmentation, the subtree with N as its root ($T(N)$) is taken out of f to form a separate fragment. Figure 5 shows an example of such a fragmentation for an XPath query '/a/b[c='DOG']/d/e' where the area marked by the dotted line represents a fragment. The path types involved in processing the filter steps are '/a/b' and '/a/b/c'. Thus, the 'b' and 'c' elements are put into the same fragment with 'b' right under the header. If the 'b' and 'c' elements belong to different fragments, the query pipeline operator for '/a/b' is supposed to keep the intermediate processing results for the 'b' fragments in memory until the related 'c' fragments arrive and are processed by the query pipeline operator for '/a/b/c'. To prevent such a scenario, the elements involved in the filter steps are put to the same fragment. As for the result step, the similar fragmentation is done such that the element matching the result step (i.e., 'e' in the example) is right under the header of its own fragment.

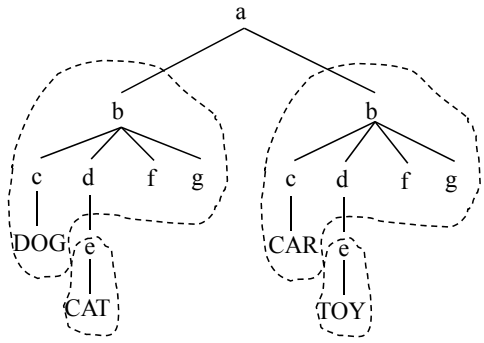


Fig. 5. Fragmentation by XPath step reduction

The fragmentation by XPath step reduction is basically aiming to reduce the navigation time. It might result in the increase of relevance checking time, because it increases the number of fragments compared with the original size-constrained fragmentation. Besides, given a set of different XPath queries, one particular fragmentation by XPath step reduction suitable for some subset of queries may harm the rest of queries.

Algorithm 3: Fragmentation by XPath Step Reduction

```
1: procedure fragmentByXSR(node)
2:    if (node.isFilterOrResultStep()) then
3:       beforeCost := estimateQueryProcessingTime();
4:       markFragUpdateStat(node);
5:       afterCost := estimateQueryProcessingTime();
6:       if(afterCost > beforeCost) undoMarkFragUpdateStat(node);
7:    endif
8:    children := node.getChildrenSortedBySize();
9:    for each child in children do
10:      fragmentByXSR(child);
11:   endfor
12 : endprocedure
```

Fig. 6. Algorithm for fragmentation by XPath step reduction

The cost model is employed to make feasible fragmentation decision. Taking out an XML element as a separate fragment from the fragment where its parents belong to is done only when such a fragmentation reduces the query processing cost estimated by Equation (6) for a single query or by Equation (7) for a set of queries. *Algorithm 3* in Figure 6 describes such a fragmentation by XPath step reduction. Algorithm 3 takes a node N of the tag structure. Lines 2–7 decide whether N corresponds to the filter or result step of any of the queries and if so, conduct fragmentation such that $T(N)$ is taken out as a separate fragment. Line 3 estimates the cost before fragmentation whereas line 5 estimates the cost after fragmentation. Line 6 *undoes* the fragmentation if after-cost is higher than the before-cost. Lines 8–11 invoke Algorithm 3 recursively for each of the children nodes of N. Since the tag structure is a tree, invoking Algorithm 3 with its root as input would traverse it, conducting fragmentation by XPath step reduction.

3.4.4 Horizontal Fragmentation by Sibling Subtree Merge

Thus far, all the fragmentations considered are *vertical* ones. Employing vertical fragmentation only, however, has inherent limitation in enhancing query performance. Let us consider, for example, the fragmentation of Figure 5 and an XPath

query q: '/a/b[c='DOG']/d/e'. The fragment with 'b' right under the header contains the child elements of 'b', that is, 'c', 'd', 'f', and 'g'. Among these children, the only one which needs to be accessed for the processing of q is 'c' due to XPath step reduction. It is part of the filter steps with a predicate condition. The rest of the children merely increases the navigation time. To reduce the navigation time, each of them could be taken out to form its own fragment. Figure 7 a) shows such a fragmentation. However, such a purely vertical fragmentation would result in the increase of relevance checking time, because now there exist more fragments.

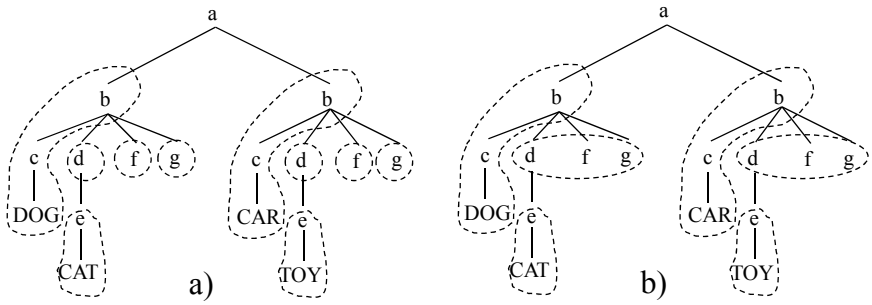


Fig. 7. Fragmentation by sibling tree merge

To solve such a problem, the merging of sibling subtrees can be considered. Such a merge results in *horizontal* fragmentation. Thus, along with the vertical fragmentation employed so far, our fragmentation becomes a hybrid one. Figure 7 b) shows the result of incorporating the sibling subtree merge into the fragmentation of Figure 7 a). With sibling subtree merge, all the sibling elements *not* needed for query processing can be gathered and put to one fragment as long as the fragment size constraint is not violated. Such a merge would improve query performance if the benefit of navigation time reduction exceeds the cost of relevance checking time increase. Given a set of queries and their frequencies, the sibling subtree merge may not always improve query performance for all the queries, because such cost and benefit out of the merge considered would not be the same for all the queries. Therefore, we need to rely on the cost model. Sibling subtree merge is applied only if the query processing cost estimated by Equation (6) for a single query or by Equation (7) for a set of queries is reduced with it. In addition, this merge can be applied only when the fragment size constraint is satisfied.

Figure 8 describes *Algorithm 4*, which conducts sibling subtree merge. It takes a node N of the tag structure. Lines 3–9 select the candidate nodes for merge. The conditions that a candidate should satisfy are checked (line 5), and also the fragment size constraint is checked every time a node becomes a candidate, enlarging the candidate pool (line 7). Lines 10–15 conduct the merge, taking out the selected nodes together as a separate fragment and updating the statistics of the resulting fragmentation. Line 2 estimates the cost before fragmentation whereas line 16 estimates the cost after fragmentation. Line 17 undoes the fragmentation if after-cost is

Algorithm 4: Fragmentation by Sibling Subtree Merge

```
1: procedure mergeSiblingSubtrees(node)
2:   beforeCost := estimateQueryProcessingTime();
3:   children := node.getChildren();
4:   for each child in children do
5:     if (!child.isFilterOrResultStep()
               && !child.hasChildFragment()
               && !child.isFragmentRoot()) then
6:       child.setMergeCandidate("true");
7:       if (node.getMergeRoot().getSize() > fragmentSizeLimit)
           child.setMergeCandidate("false");
8:     endif
9:   endfor
10:  mergeRoot := node.getMergeRoot();
11:  if (mergeRoot == NULL) return; // there is not a
                                    // candidate node for merge
12:  mNumElements := mergeRoot.getNumElements();
13:  mByteSize := mergeRoot.getByteSize();
14:  node.setNumElements(node.getNumElements() - mNumElements);
15:  node.setByteSize(node.getByteSize() - mByteSize);
16:  afterCost = estimateProcessingTime();
17:  if (beforeCost < afterCost) node.undoMerge();
18: end procedure
```

Fig. 8. Algorithm for fragmentation by sibling subtree merge

higher than the before-cost. Since the tag structure is a tree, invoking Algorithm 4 with its root as input would traverse it, conducting fragmentation by sibling subtree merge.

4 IMPLEMENTATION AND PERFORMANCE EVALUATION

4.1 Overview

This section describes implementation of the proposed method and compares its performance with previous methods in the literature through experiments. Since our proposed method considers the constraint on the fragment size while conducting the fragmentation based on query processing cost and query frequency, we call it XML Fragmentation by Size and Query (SQ). It is compared with *DTD-based fragmentation* method of [13] (denoted as D, hereafter) and with *Path Frequency Tree* method (PFT) proposed in [10].

These three schemes were implemented in Java using J2SE Development Kit 5.0 Update 11. Performance evaluation was conducted in a system of Windows XP platform with Intel Pentium D 2.66 GHz CPU and 1.5 GB RAM. The *auction* XML documents used for performance evaluation were generated by *xmlgen* program in the XMark benchmark [19]. The size of the document used in the experiments is 10.8 MB (the scaling factor set in running *xmlgen* was 0.1, and all the white spaces such as spaces for indentation, tabs, and newlines were removed).

As for the algorithm of processing an XPath query over an XML fragment stream, the one described in [13] was employed, because it is the most memory-efficient among those proposed in the literature. Since the order in which each client receives the XML fragments of the stream could be different from that in which the server broadcasts them, we considered both the best and the worst case scenarios. Given an XML fragmentation, it can be represented as an XML fragment tree where a node is an XML fragment. In the best case scenario, the XML fragments are received at clients in the order of *preorder* traversal of the fragment tree. In the worst case one, they are received level by level of the fragment tree in the *bottom-up* order. In Figure 9, both the preorder and the bottom-up order of fragments are shown for an XML fragment tree.

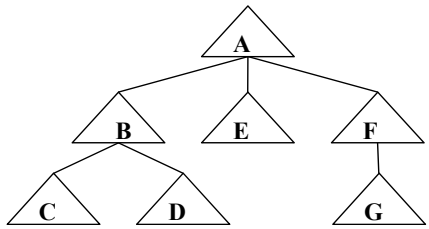


Fig. 9. Orders of fragment streaming

As for the constant K used in Equation (6) in our cost model, it was set to 5 in the experiments. Such a value was obtained empirically through a number of experiments with the query processing algorithm of [13]. As we mentioned in Section 3, the size constraint on every fragment can be specified with SQ. The upper limit on the fragment size with SQ was set to 20 KB. This value was chosen because the largest size of fragments out of D or PFT was about 20KB. The performance metric used was the query processing time. Every experiment was repeated 10 times under the same condition, and the results were averaged.

4.2 Evaluation of XML Fragmentation for a Single Query

This section presents experimental results for query performance with the three methods of XML fragmentation for a single query. Table 2 shows a list of 20 XPath

ID	XPath Expression
Q_1	/site/open_auctions/open_auction[initial>“10”]//increase
Q_2	/site/open_auctions/open_auction/bidder[increase>“200”]
Q_3	/site/open_auctions/open_auction[initial>“0”]/bidder
Q_4	/site/open_auctions/open_auction[initial>“10”]//start
Q_5	/site/open_auctions/open_auction[initial>“200”]/bidder/time
Q_6	/site/open_auctions/open_auction/bidder[increase>“200”]/time
Q_7	/site/open_auctions/open_auction[initial>“500”]/bidder[increase>“200”]/time
Q_8	/site//increase
Q_9	/site/people/person[name=“Claudine Nunn”]/watches/watch
Q_{10}	/site/people/person[name=“Torkel Prodrodmidis”]/profile
Q_{11}	/site/people/person/homepage
Q_{12}	/site/closed_auctions/closed_auction[price>“100”]/type
Q_{13}	/site/closed_auctions/closed_auction[price>“200”]/annotation/author
Q_{14}	/site/closed_auctions/closed_auction[price>“40”]/itemref
Q_{15}	/site/closed_auctions/closed_auction/buyer
Q_{16}	/site/regions/africa/item[location=“United States”]/mailbox/mail/from
Q_{17}	/site/regions/namerica/item[payment=“Creditcard”]
Q_{18}	/site/regions/australia/item/description
Q_{19}	/site/regions//item
Q_{20}	/site//emailaddress

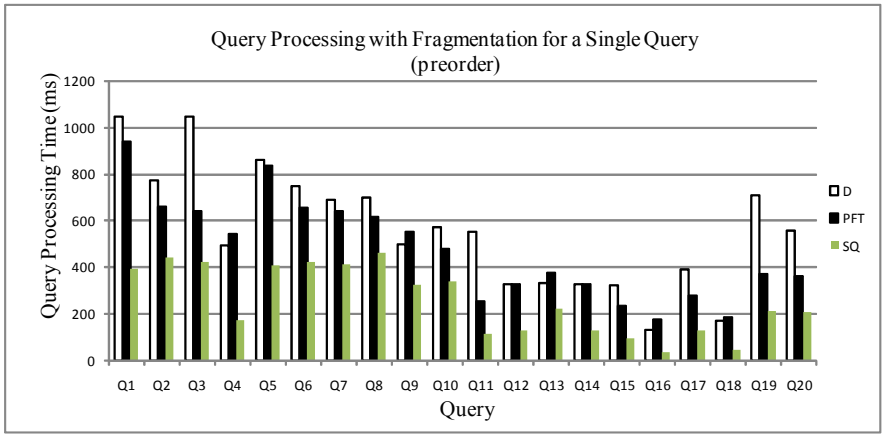
Table 2. XPath queries in experiments

queries used for this experiment. For each query of Q_1 through Q_{20} in Table 2, the *auction* XML document was fragmented by SQ, D, and PFT, respectively.

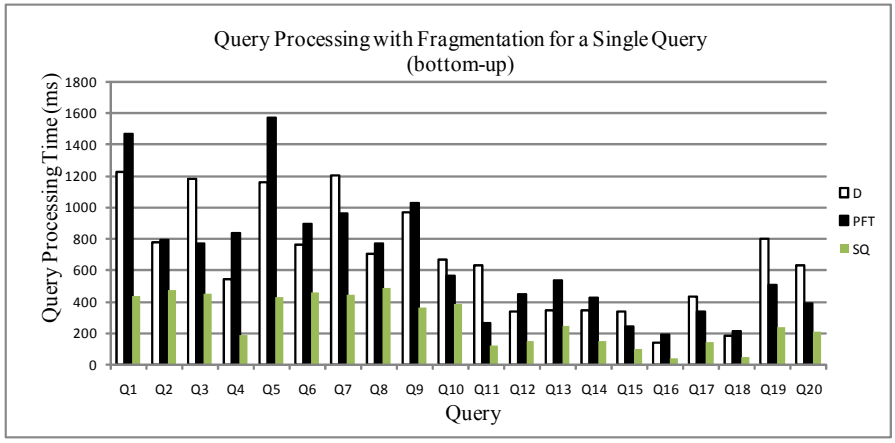
Figure 10 a) compares the query processing time of each query with respect to three different fragmentations when the order of fragment arrival is the preorder. We can see that SQ outperforms D and PFT for all the 20 different queries. Figure 10 b) compares them for the bottom-up order. Even in this worst case scenario, SQ absolutely outperforms the other two for all the queries. We can also note that SQ actually performs similarly for both the best and the worst case scenarios while the performance with D or PFT in the worst case has considerably degraded compared with their best case counterpart.

ID	XPath Expression
Q_1	/site/open_auctions/open_auction[initial>“10”]//increase
Q_2	/site/open_auctions/open_auction[initial>“10”]//start
Q_3	/site/open_auctions/open_auction[initial>“500”]/bidder[increase¿“200”]/time
Q_4	/site/people/person[name=“Claudine Nunn”]/watches/watch
Q_5	/site/closed_auctions/closed_auction[price>“100”]/type
Q_6	/site/closed_auctions/closed_auction/buyer
Q_7	/site/regions/australia/item/description

Table 3. Query set 1



a) Preorder



b) Bottom-up

Fig. 10. Query processing time with fragmentation for a single query

4.3 Evaluation of XML Fragmentation for Multiple Queries

This section presents experimental results for multiple queries. The 20 XPath queries in Table 2 were partitioned into 3 sets of queries: query set 1, 2, and 3 as shown in Tables 3 through 5. Performance evaluation for the case of multiple queries has been conducted for each query set. For each query set, the *auction* XML document was fragmented with SQ, D and PFT, respectively for the multiple queries in the set assuming that they have the equal frequencies.

The three graphs in the left-hand side of Figure 11 show the query processing time for each query in the query set 1, 2, and 3, respectively when the fragments were broadcast in preorder. SQ outperforms the other two methods for all

ID	XPath Expression
Q_1	/site/open_auctions/open_auction/bidder[increase>"200"]
Q_2	/site/open_auctions/open_auction[initial>"200"]/bidder/time
Q_3	/site//increase
Q_4	/site/people/person[name="Torkel Prodrodmidis"]/profile
Q_5	/site/closed_auctions/closed_auction[price>"200"]/annotation/author
Q_6	/site/regions/africa/item[location="United States"]/mailbox/mail/from
Q_7	/site/regions//item

Table 4. Query set 2

ID	XPath Expression
Q_1	/site/open_auctions/open_auction[initial>"0"]/bidder
Q_2	/site/open_auctions/open_auction/bidder[increase>"200"]/time
Q_3	/site/people/person/homepage
Q_4	/site/closed_auctions/closed_auction[price>"40"]/itemref
Q_5	/site/regions/namerica/item[payment="Creditcard"]
Q_6	/site//emailaddress

Table 5. Query set 3

the three different query sets. As for PFT, it performs better than D in most queries but there exist queries (e.g., Q_2, Q_4, Q_5, and Q_7 in the query set 1) for which that is not the case. The reason for that deserves analysis. For the query set 1, PFT created 51 857 fragments, that is more than 32 289 fragments created by D. The reason for the worse performance of PFT compared with D for some queries is because the relevance checking takes longer for the entire stream with more number of fragments produced by PFT. In comparison, SQ created 13 144 fragments. We note that SQ has created the least number of fragments among the three methods. SQ takes out the part of the XML document irrelevant to the queries or infrequently referenced by the queries as large a fragment as possible under the fragment size constraint. It helps reduce query processing time by reducing the total relevance checking time for the entire stream of fragments. SQ also takes out the part of the XML document relevant to the queries or frequently referenced by the queries as small a fragment as possible, reducing the total navigation time. The superiority of SQ is mostly due to the fact that its fragmentation decision is based on the cost model. Only when the query processing cost estimated with the Equation (7) is reduced with a certain fragmentation, it is adopted.

The three graphs in the right-hand side of Figure 11 show the query processing time for each query in the query set 1, 2, and 3, respectively when fragments were broadcast bottom-up. SQ outperforms the other two even in this worst case scenario. While query processing times with D and PFT for the bottom-up order significantly increased compared with those for the preorder, little performance degradation was observed for the case with SQ. Table 6 summarizes the average performance im-

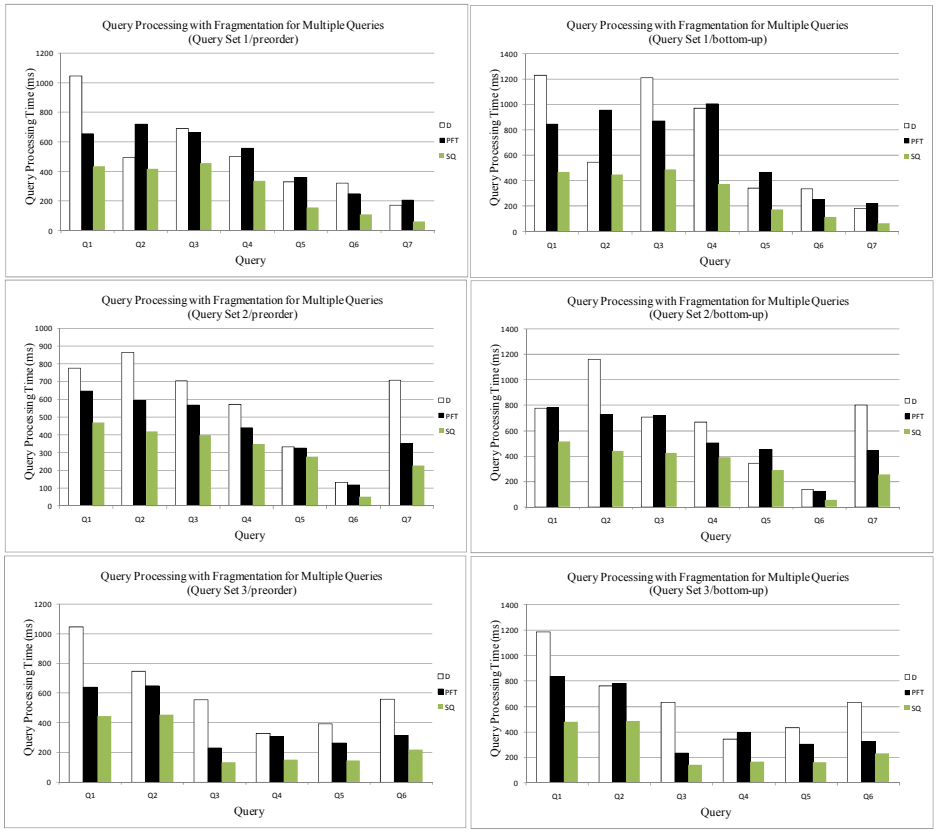


Fig. 11. Query processing time with fragmentation for multiple queries

provement of SQ compared with D and PFT for the three query sets and for the two orders of fragment streaming.

5 RELATED WORK

The previous work on XML fragment stream query processing [4, 5, 10, 11, 12, 13] as outlined in Section 1 has paid little attention to the method of efficient XML fragmentation with rare exceptions [10, 13], because the main problem dealt with in those work was memory efficiency in query processing. In [13], a *DTD-based fragmentation* was devised. It is assumed that the source XML document to be fragmented conforms to a DTD (Document Type Definition), and the characteristic that when an XML document is represented as a tree, typically the width is wide while depth is shallow [16], was considered. This method fragments an XML document such that each of the repeating element (marked with '*' or '+' in the DTD)

Unit [%]

Query Set	Method	D	PFT
Query Set 1	(preorder)	44	42
Query Set 2	(preorder)	46	28
Query Set 3	(preorder)	53	31
Query Set 1	(bottom-up)	56	54
Query Set 2	(bottom-up)	48	37
Query Set 3	(bottom-up)	59	41

Table 6. Performance improvement with SQ over D and PFT

instances separately forms a fragment. Such a straightforward vertical fragmentation does not consider the effect of fragmentation on the time it takes for XML fragment stream query processing. In [10], such an effect is considered when the fragmentation is designed with the schema information based on the query frequencies. Two methods of XML fragmentation, called *Path Frequency Tree (PFT)* and *Markov tables (MP)*, were proposed. Both methods work under the same heuristic, taking into account the query frequencies on the elements of the XML document. They merge the XML elements with low frequency in the same fragment to enhance fragment utilization, and merge those with high frequency in the same fragment to enhance fragment cohesion. They require the threshold values for the high and low frequencies be given for fragmentation. Since PFT performs better than MP according to the experiments in [10], we have employed PFT as well as the DTD-based method of [13] for comparison with our proposed method as described in the previous section. The fundamental limitations of PFT and DTD-based method are that first, they did not consider the performance trade-off between relevance checking time and navigation time in XML fragment stream query processing, and second, they *cannot* impose the upper limit on the fragment size.

In [20], a method of fragmenting an XML document for efficient transmission over the network was investigated. In this method, an XML document modeled as a tree is partitioned into fragments called XDU (Xstream Data Unit), each of which is a transmission unit smaller than the maximum amount of data that can be transferred over the network. At the receiving end, the received XDUs are merged to reconstruct the original XML document. This fragmentation method is different from our XML fragmentation, because the emphasis is put to the efficiency of reconstruction at the receiving end. In our work, the XML fragments are not merged for reconstruction of the source XML document.

The methods of XML document fragmentation were investigated in distributed and/or parallel XML query processing environments [7, 15, 2, 3]. In [7], design and query models for distributed XML repository were presented. Considering that XML data is usually distributed on the Web, techniques of fragmenting virtual XML repository and allocating them in appropriate locations for the expected tasks were

proposed to build an effective XML distributed processing environment. In [15], adaptation of the well-known methods of relational database fragmentation, horizontal and vertical, to XML data was formally dealt with. In [2], a framework called *PartiX* was proposed where the vertical, horizontal, and hybrid fragmentation of XML data is employed for distribution of XML data and parallel processing of an XML query against the distributed data. In PartiX, the fragmentation of not just a single XML document but a set of XML documents was considered. In [3], the problem of producing large intermediate results during XML query processing in a distributed computing environment was tackled by devising a method of XML document fragmentation that takes into account the structural characteristics of the document (e.g., document size, the depth and width of the subtrees when the document is represented as a tree). It contrasts to the conventional semantic-based fragmentation (e.g., predicate-based horizontal fragmentation). There are also other methods of XML fragmentation proposed for *active XML* suitable for Web services and P2P applications [1]. However, all these works cannot be applied to query processing over XML fragment stream. Basically, they are for different framework of query processing and their goal is to improve efficiency through data distribution and through distributed and parallel processing of XML data in a distributed environment.

6 CONCLUSIONS

One of distinct aspects in modern computing is that resource-limited devices or systems often play a key role. In such an environment, data stream processing with efficient resource management is getting more important and becomes a core technique. Such a challenge is widely encountered in ubiquitous computing as dealt with in this paper and even in grid computing [21]. In this paper, we addressed the problem of *XML document fragmentation* for query processing over XML fragment stream in mobile and hand-held devices widely deployed in ubiquitous computing environment. Since the memory capacity of those mobile devices is yet limited, a large volume of XML document is fragmented and the fragments are streamed for query processing. The *time* it takes for such stream query processing significantly depends on how the source XML document is fragmented. We proposed a method of XML document fragmentation whereby the stream query processing time gets reduced while the constraint of the upper limit of the fragment size is satisfied. Effectiveness of our proposed method was verified through implementation and a detailed set of experiments. The mean reduction in query processing time with our method compared with the previous ones measured in single query experiments was about 56 %, and that in multiple query experiments was about 45 %.

The *contribution* of this paper is that a foundation of *time optimization* of XML fragment stream query processing has been suggested. Compared with the issue of memory efficiency, time efficiency in XML fragment stream query processing has received little attention. We have observed the performance trade-off between

the *relevance checking time* and the *navigation time* for each of the fragments, and developed a *cost model* of XML fragment stream query processing, which plays a key role in making decisions on the fragmentation based on the *XPath step reduction* and *subtree merge*.

In this paper, we have focused on the time cost of XML fragment stream query processing, developing a cost model. In [13], its space cost (i.e., memory requirement) was investigated, but without a cost formula. In general, time optimization often sacrifices space efficiency. However, we firmly believe that such a trade-off would not exist with our proposed method, because the upper limit of the fragment size is enforced and memory efficiency is also taken care of in the fragmentation based on the XPath step reduction. The latter technique is devised for time efficiency in this paper but it is also expected to benefit space efficiency. As a future work, we plan to investigate if *memory-conscious* XML fragmentation other than the fragmentation based on the XPath step reduction is possible, and how it could be integrated with the time-conscious fragmentation proposed in this paper.

REFERENCES

[1] Abiteboul, S.—Benjelloun, O.—Cautis, B.—Manolescu, I.—Milo, T.—Preda, N.: Lazy Query Evaluation for Active XML. Proc. ACM SIGMOD Int'l Conf. on Management of Data, 2004, pp. 227–238.

[2] Andrade, A.—Ruberg, G.—Baiao, F.—Braganholo, V.—Mattoso, M.: Efficiently Processing XML Queries over Fragmented Repositories with PartiX. Proc. Int'l Conf. on Extending Database Technology, 2006, pp. 150–163.

[3] Bonifati, A.—Cuzzocrea, A.: Efficient Fragmentation of Large XML Documents. Proc. Int'l Conf. on Database and Expert Systems Applications, 2007, pp. 539–550.

[4] Bose, S.—Fegaras, L.: Data Stream Management for Historical XML Data. Proc. ACM SIGMOD Int'l Conf. on Management of Data, 2004, pp. 239–250.

[5] Bose, S.—Fegaras, L.: XFrag: A Query Processing Framework for Fragmented XML Data. Proc. Int'l Workshop on the Web and Databases, 2005, pp. 97–102.

[6] Bose, S.—Fegaras, L.—Levine, D.—Chaluvadi, V.: A Query Algebra for Fragmented XML Stream Data. Proc. Int'l Conf. on Data Base Programming Languages, 2003, pp. 195–215.

[7] Bremer, J.—Gertz, M.: On Distributing XML Repositories. Proc. Int'l Workshop on the Web and Databases, 2003, pp. 73–38.

[8] Clark, J.—DeRose, S. (Eds.): XML Path Language (XPath) version 1.0. W3C Recommendation, Nov. 1999, `http://www.w3.org/TR/xpath`.

[9] Fegaras, L.—Levine, D.—Bose, S.—Chaluvadi, V.: Query Processing of Streamed XML Data. Proc. Int'l Conf. on Information and Knowledge Management, 2002, pp. 126–133.

[10] Huo, H.—Wang, G.—Hui, X.—Xiao, C.—Zhou, R.: Document Fragmentation for XML Streams Based on Query Statistics. Proc. Int'l Conf. on WISE, 2006, pp. 350–356.

[11] Huo, H.—Wang, G.—Hui, X.—Zhou, R.—Ning, B.—Xiao, C.: Effiecient Query Processing for Streamed XML Fragments. Proc. Int'l Conf. on DASFAA, 2006, pp. 468–482.

[12] Lee, S.—Kim, J.—Kang, H.: XFLab: A Technique of Query Processing over XML Fragment Stream. Proc. British Nat'l Conf. on Databases, 2007, pp. 182–186.

[13] Lee, S.—Kim, J.—Kang, H.: Memory-Efficient Query Processing over XML Fragment Stream with Fragment Labeling. Computing and Informatics, Vol. 29, 2010, No. 5, pp. 757–782.

[14] Li, C.—Ling, T.: QED: A Novel Quaternary Encoding to Completely Avoid Relabeling in XML Updates. Proc. Int'l Conf. on Information and Knowledge Management, 2005, pp. 501–508.

[15] Ma, H.—Schewe, K.: Fragmentation of XML Documents. Proc. Brazilian Symposium on Databases, 2003, pp. 200–214.

[16] Mignet, L.—Barbosa, D.—Veltri, P.: The XML Web: A First Study. Proc. Int'l WWW Conf., 2003, pp. 500–510.

[17] O'Neil, P.—O'Neil, E.—Pal, S.—Cseri, I.—Schaller, G.—Westbury, N.: ORDPATHs: Insert-Friendly XML Node Labels. Proc. ACM SIGMOD Int'l Conf. on Management of Data, 2004, pp. 903–908.

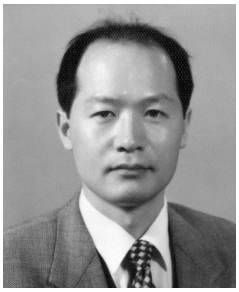
[18] Online Computer Library Center: Introduction to the Dewey Decimal Classification. `http://www.oclc.org/oclc/fp/about/about_the_ddc.htm`.

[19] Schmidt, A.—Wass, F.—Kersten, M.—Carey, M.—Manolescu, I.—Busse, R.: XMark: A Benchmark for XML Data Management. Proc. Int'l Conf. on VLDB, 2002, pp. 974–985.

[20] Wong, E.—Chan, A.—Leong, H.: Efficient Management of XML Contents over Wireless Environment by Xstream. Proc. ACM Symposium on Applied Computing, 2004, pp. 1122–1127.

[21] Zhang, W.—Cao, J.—Zhong, Y.—Liu, L.—Wu, C.: Grid Resource Management and Scheduling for Data Streaming Applications. Computing and Informatics, Vol. 29, 2010, No. 6+, pp. 1193–1220.

Jin Kim received the B. Sc. and M. Sc. degrees in computer science and engineering from Chung-Ang University, Seoul, Korea in 2006 and 2008, respectively. His research interests include XML database and web database.



Hyunchul Kang received the B. Sc. degree in computer engineering from Seoul National University, Seoul, Korea in 1983, and the M. Sc. and Ph. D. degrees in computer science from University of Maryland, College Park in 1985 and 1987, respectively. In 1988, he joined the School of Computer Science and Engineering, Chung-Ang University, Seoul, Korea where he is currently a Professor. His current research interests include XML and web data management, stream data management, and mobile data management.