Computing and Informatics, Vol. 29, 2010, 757–782

MEMORY-EFFICIENT QUERY PROCESSING OVER XML FRAGMENT STREAM WITH FRAGMENT LABELING

Sangwook Lee, Jin Kim, Hyunchul Kang

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

Manuscript received 3 February 2009; revised 16 March 2009
Communicated by Imre J. Rudas

Abstract. The portable/hand-held devices deployed in mobile computing environment are mostly limited in memory. To make it possible for them to locally process queries over a large volume of XML data, the data needs to be streamed in fragments of manageable size and the queries need to be processed over the stream with as little memory as possible. In this paper, we report a considerable improvement of the state-of-the-art techniques of query processing over *XML fragment stream* in *memory efficiency*. We use *XML fragment labeling (XFL)* as a method of representing XML fragmentation, and show that XFL is much more effective than the popular *hole-filler (HF) model* employed in the state-of-the-art in reducing the amount of memory required for query processing. The state-of-the-art with the HF model requires more memory as the stream size increases. With XFL, we overcome this fundamental limitation, proposing the techniques to make query processing *scalable* in the sense that memory requirement is *not* affected by the size of the stream as long as the stream is bounded. The improvement is verified through implementation and a detailed set of experiments.

Keywords: XML, XML fragment stream, XML fragment labeling, hole-filler model, XML stream query processing, mobile computing

Mathematics Subject Classification 2000: 68N01: Software; 68U01: Computing Methodologies and Applications; 68M14: Distributed Systems; 68M20: Performance Evaluation

1 INTRODUCTION

Due to the advance of wireless communication technology, the portable/hand-held devices of various sorts have been widely deployed in mobile and pervasive computing environment. For instance, the global market for the smart phones is explosively growing in recent years. According to strategy analytics, a market research institute, 91 million smart phones were sold worldwide in 2006, 150 million in 2007, 211 million in 2008, and 460 million are expected to be sold in 2010 [21]. In the near future, with a number of mobile devices around with its own processing capabilities, dissemination of XML data to them over wireless broadcast channels will be prevalent, for XML has been established as a standard for data exchange not just on the Internet but among heterogeneous devices and systems. Additional filtering of the broadcast XML data could be locally conducted in the client devices through XML queries. The portable/hand-held client devices of today are still with limited resource, especially with limited memory. As such, the whole of the broadcast data could not be stored as an XML document against which the client queries are to be executed. Rather, the server partitions the source XML document into *XML fragments* of small and manageable size, and broadcasts a stream of them. Thus, query processing at the client is over such an XML fragment *stream*. Arrival of the XML fragments at a client may not be in proper order, and yet the stream query processing over them should return the correct result while using the limited amount of memory available at the client. In this scenario, the most critical requirement is to use as little memory as possible in query processing.

To realize such a capability, three components are needed. First, a method of representing XML fragmentation; second, XML data schema with fragmentation information; third, a stream query processing algorithm. The state-of-the-art techniques in this approach include XFrag [2] and XFPro [11]. They employ the *hole-filler model* (*HF model* for short) [3, 6] for the first component. The HF model is simple in representing XML fragmentation. Each XML fragment could contain holes, which are supposed to be filled with other XML fragments (i.e., *fillers*) that could have other holes. For each hole and its corresponding filler, a value is assigned. It is called a *hole ID* (for the hole) or a *filler ID* (for the filler). The HF model is getting popular and widely adopted for relevant research [2, 11, 1, 10].

This paper is to claim that there exists a much more effective method of representing XML fragmentation than the HF model. We call it *XML fragment labeling* (*XFL* for short). XFL is as simple as the HF model, and is basically an extension of the well-known XML labeling (a.k.a. XML numbering) [23, 15, 22, 17, 14] in the sense that the unit of labeling is an XML fragment whereas in the conventional XML labeling, the unit is an XML node when the XML data is modeled as a tree.

In [2], the memory usage for each of three XPath queries on an *auction.xml* document was measured with the HF model. With only the HF model replaced by XFL, in our earlier work we had conducted exactly the same experiments. The results with XFL were very promising compared with those with the HF model.

When the *minimum* amount of memory required during the entire course of query processing is compared, the improvement was about in the range of 25 % to 50 % for each of the three queries. Encouraged by these preliminary results reported in [13], we have further explored XFL and drawn a conclusion that the state-of-the-art techniques of XML fragment stream query processing should *replace* its HF model with XFL. First, we explain in detail why XFL is much more efficient than the HF model with examples. Then, we propose several techniques to *optimize* XML fragment stream query processing for memory efficiency with XFL and also with the HF model for fairness in comparison. There, we show that XFL is much more lenient to such optimizations than the HF model. The state-of-the-art with the HF model can deal only with a static stream, and requires more memory for query processing as the stream size increases. With XFL, a *dynamic* stream could be handled as well, and a very crucial result is that query processing is *scalable* due to the aforementioned optimizations in the sense that memory requirement is *not* affected by the stream size as long as the stream is bounded. Such scalability could not be achieved with the HF model even with the possible optimizations.

The rest of this paper is organized as follows. Section 2 describes the HF model and introduces XFL. The inefficiency of the HF model compared with XFL is described. Section 3 proposes some possible optimization techniques with XFL and also with the HF model. Section 4 reports the implementation and experimental results. Section 5 overviews the related work. Finally, Section 6 summarizes the contributions of this paper and gives some concluding remarks.

2 HOLE-FILLER MODEL AND XML FRAGMENT LABELING

2.1 Hole-Filler Model [3, 6]

XML data can be modeled as a node-labeled tree. Figure 1 shows a) an example of an XML data, b) its tree representation, c) its fragmentation where the region of the tree marked by a triangle denotes an XML fragment, and d) its representation of the fragmentation with the HF model. In Figure 1 d), there are 5 fragments. The root element of each fragment has two attributes – *fid* (filler ID) and *tsid* (*tag structure* ID). The fid is the ID of a fragment (i.e., a filler), and the explanation of the tsid and the tag structure will be given shortly. The IDs of the 5 fragments are 1 through 5. Filler 1 has 2 holes in it, and their IDs are 2 and 3, respectively. Filler 2 and the filler 3 have 1 hole each whose IDs are is 4 and 5, respectively. In filler 1, the original start tag <a> is annotated with attributes fid and tsid, resulting in <a fid=1 tsid=10>. The original body of the element 'a', <c>10</c><d>20</d> <c>30</c><d>40</d> is replaced by the 2 holes, <hole hid=2 tsid=11/> and <hole hid=3 tsid=11/> where *hid* represents the hole ID. These are called *hole elements*. Meanwhile, in filler 2, the original body of the first instance of element 'b', <c>10</c><d>20</d>, is replaced by <c>10</c> and a hole element <hole hid=4 tsid=13/>. The same pattern of fragmentation occurs in the filler 3, too.

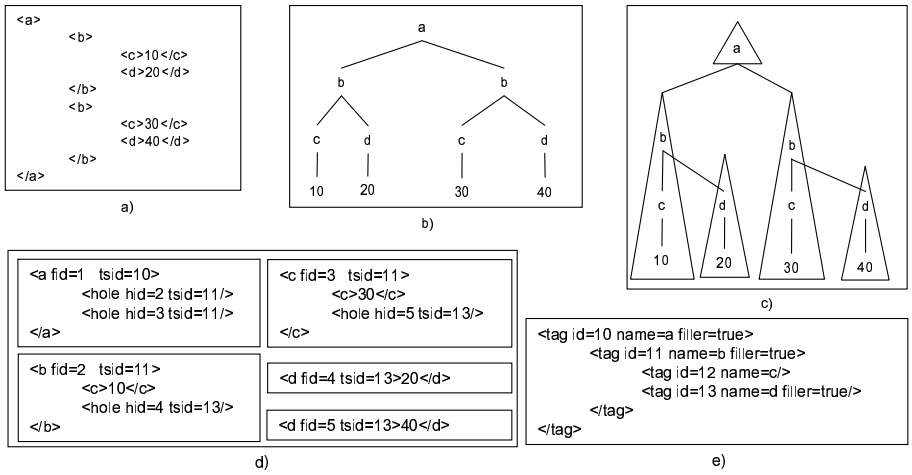


Fig. 1. Hole-filler model; a) XML data, b) tree representation, c) fragmentation, d) XML fragments with HF model, e) tag structure

The tsid is the ID assigned to each tag that occurs on a path in the XML tree. Their values are uniquely given in the *tag structure*, which is to summarize the structure of the XML data and also to specify how it is fragmented. Figure 1 e) shows the tag structure of the XML data of Figure 1 a). There are 4 tags, '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 data of Figure 1 a). They are considered as the same tag because both appears on the same path from the root of the tree (i.e., /a/b). The same is true for the tags 'c' and 'd'. The value of attribute *filler* being "true" specifies that the corresponding tag is the root of an XML fragment (i.e., the root of the subtree that corresponds to an XML fragment). The fid, hid, and tsid are the *metadata* added to XML fragments to correlate the fragments with the HF model.

2.2 XML Fragment Labeling

In this section, we introduce XML fragment labeling (XFL). 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 data modeled as a tree, each of the obtained 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 c). 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 2 a) shows

XFL of the XML fragment tree of Figure 1 c) 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 2 b). The root element of each fragment has two attributes *FID* (fragment ID)[1] and tsid. The value of FID is the label assigned with XFL. With XFL, there is *no* need of holes or fillers altogether. Thus, there are *no* hole/filler IDs specified. Only the fragment IDs (FIDs) will do. In fragment 1, the original start tag <a> is annotated with attributes FID and tsid, resulting in <a FID=1 tsid=10/>, and the original body of element 'a', <c>10</c><d>20</d><c>30</c><d>40</d>, is now *null* because the whole of it is taken out into its child fragments whose FIDs are 1.1, and 1.2. Meanwhile, in fragment 1.1, the original body of the first instance of element 'b', <c>10</c><d>20</d>, is replaced by <c>10</c> alone without the hole for <d>20</d>, which is taken out into fragment 1.1.1. The same pattern of fragmentation occurs in fragment 1.2, too. Though there is no hole element, the structural relationship among the fragments can be identified due to XML labeling used for the FIDs. The metadata added to XML fragments is just FID and tsid.

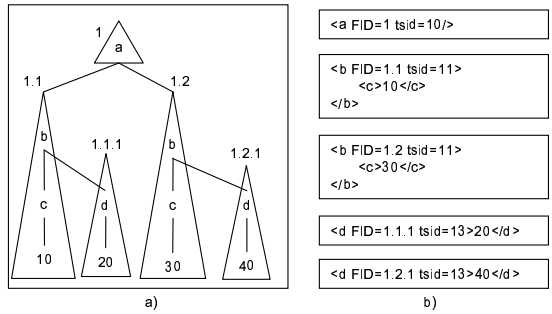


Fig. 2. XML fragment labeling; a) XML fragment tree with XFL, b) XML fragments with XFL

The basic XFL presented so far is not complete in representing an XML fragmentation. There could exist *ambiguity*, example of which is shown in Figure 3. In Figure 3 a), we have an XML fragment tree consisting of 2 fragments whose FIDs are 1 and 1.1. Since there are 2 instances of 'b' elements in fragment 1, we cannot identify which 'b' is the parent of the 'c' element in fragment 1.1. Such ambiguity should be avoided in XFL. Otherwise, it is not always guaranteed that the original XML source data could be reconstructed exactly as it was, given its fragments and their FIDs. Though such reconstruction is not needed in XML fragment stream query processing, correct reconstruction is a requirement for any method of representing XML fragmentation for correct query processing. Figures 3 b) and Figure 3 c) show

[1] To distinguish the fragment ID in XFL from the filler ID of the HF model, the acronym in uppercase letters, FID, is used for the fragment ID whereas the one in lowercase letters, fid, is used for the filler ID.

examples of unambiguous fragmentation with XFL. With a simple rule, the ambiguity can be avoided. The rule is that the repeating elements, say n elements, with the same tag name at the same path from the root of an XML document (e.g., those that are marked with * or + in a DTD) are partitioned into n different fragments. We wrap up this subsection with two definitions and a lemma. The notations here will be used again in Section 3.

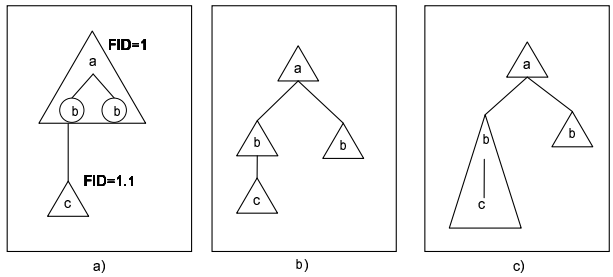


Fig. 3. Ambiguous and unambiguous fragmentation with XFL

Definition 1 (XML Fragmentation Function)**.** Let D, τ, M_f be an XML document, a tag structure for D, and a method of representing XML fragmentation, respectively. Let F_x be a function which maps D into a collection of XML fragments out of D such that $F_x(D, \tau, M_f) = < d_1, \ldots, d_k > (k \geq 1)$ where d_i is an XML fragment which comprises ${d_i}^s$ and ${d_i}^m$ where ${d_i}^s$ is a subtree of D obtained as specified in τ when D is modeled as a node-labeled tree, and ${d_i}^m$ is the metadata added to ${d_i}^s$ to correlate ${d_i}^s$ with other ${d_j}^s$'s (i≠j) according to M_f.

Definition 2 (Unambiguous XML Fragmentation)**.** Let D, τ, M_f be an XML document, a tag structure for D, and a method of representing XML fragmentation, respectively. An XML fragmentation of D with τ and M_f is *unambiguous* if $R_x(F_x(D, \tau, M_f), \tau, M_f) = D$ where R_x is a reverse function of F_x which reconstructs D from a collection of XML fragments out of D produced by F_x.

Lemma 1. Given an XML document D, there exists an unambiguous XML fragmentation of D with XFL.

Proof. Suppose D is vertically fragmented under the aforementioned rule such that $F_x(D, \tau,$ 'XFL') $= < d_1, \ldots, d_k > (k \geq 1)$ where τ is the tag structure and $d_i (i = 1, \ldots, k)$ is an XML fragment as defined in Definition 1. Let r_D be the root node of D. For each fragment d_i except the one with r_D, the following two statements hold:

1. The parent fragment d_j of $d_i (i \neq j)$ is uniquely identified.
2. The parent node of the root element of d_i in D is uniquely identified in d_j.

The first statement holds because of the FIDs of d_i and d_j. To show that the second one also holds, let r_i and r_j be the root elements of d_i and d_j, respectively, and given two node instances x and y in D such that x is an ancestor of y, let $P(x, y)$ denote the path from x to y. Note that r_i has the tsid attribute. Consulting the tag structure τ with the tsid value of r_i, $P(r_D, r_i)$ can be known. Let $P(r_D, r_i) = P(r_D, r_j) \cdot P(r_j, n_p) \cdot P(n_p, r_i)$ where n_p is the parent node of r_i in D, and the $\cdot$ denotes the path concatenation where the connecting node appears just once (e.g., '/a/b/c' $\cdot$ 'c/d/e' = '/a/b/c/d/e'). The node n_p must be in r_j at the path $P(r_j, n_p)$, and there is only one instance of such n_p in r_j because of the rule applied in fragmentation.

Since both d_i and d_j are trees, the merge of d_i and d_j by connecting r_i as a child of n_p in d_j results in a tree. In such a merge, there is no ambiguity because (1) and (2) above hold. Thus, the function R_x exists such that $R_x(F_x(D, \tau, \text{'XFL'}), \tau, \text{'XFL'}) = D$. □

2.3 Query Processing over XML Fragment Stream with XFL

In this section, a brief sketch of how an XPath query is processed over XML fragment stream with XFL is given with an example. Consider an XML fragment tree with XFL in Figure 2 a). Its corresponding tag structure is given in Figure 1 e). For an XPath expression, it is transformed into a query processor where each *location step* of the XPath expression [4] is an operator. For example, XPath expression /a/b[c=30]/d against the XML tree in Figure 2 a) is transformed into a query processor shown in Figure 4 a) where each box is an operator and the value in parentheses beside each operator is the tag ID of the operator as defined in the tag structure. Now the question is how <d>40</d>, the answer to the query /a/b[c=30]/d, would be produced.

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 the completion of 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 fragment is essential because the whole of the source XML data would not be reconstructed and the fragment itself just streamed in would be discarded soon. This is especially so when the memory to be taken up for query processing is limited.

The tag structure in Figure 1 e) is delivered to the client first. Now suppose the order of fragments of Figure 2 b) arriving at the client is 1, 1.1, 1.2, 1.1.1, 1.2.1. Then, the query /a/b[c=30]/d is processed as follows:

- Fragment 1, <a FID=1 tsid=10 />, arrives. The tsid of the root element in this fragment is 10. Consulting the tag structure, it is known that the *set* of tsid values of the elements contained in this fragment is {10}. (Actually, there is only one element 'a'). Thus, in the query processor in Figure 4 a), the operator which needs to process this fragment is only the 'a' operator. Fragment

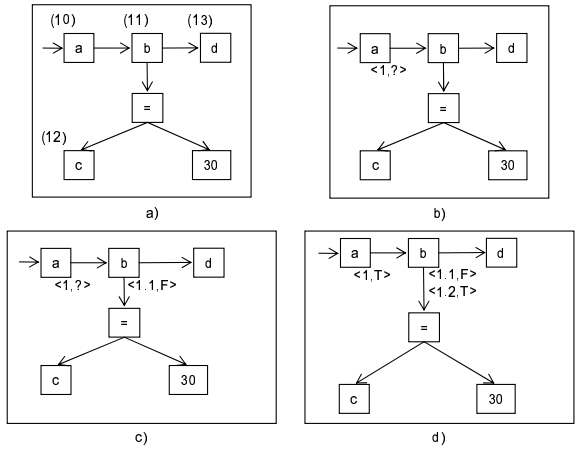


Fig. 4. XML fragment stream query processing with XFL; a) fragment stream query processor, b) after fragment 1 arrives, c) after fragment 1.1 arrives, d) after fragment 1.2 arrives

information <1, ?> is recorded in association with the operator 'a' where '?' denotes 'undecided' (Figure 4 b)). Then, the fragment itself is discarded.

- Fragment 1.1, <c>10</c>, arrives. The tsid of the root element in this fragment is 11. Consulting the tag structure, it is known that the set of tsid values of the elements contained in this fragment is {11,12}, where '11' corresponds to the 'b' node at the path /a/b, and '12' corresponds to the 'c' node at the path /a/b/c. The 'd' node is excluded because its filler attribute value is "true" in the tag structure. Thus, the operators which need to process this fragment are 'b' and 'c'. Fragment information <1.1, F> is recorded in association with the operator 'b' where 'F' denotes that the predicate condition c=30 is false with this fragment. Then, the fragment itself is discarded (Figure 4 c)).
- Fragment 1.2, <c>30</c>, arrives. The same processing as for fragment 1.1 is done except that fragment information <1.2, T> is recorded in association with the operator 'b' where 'T' denotes true. Also the fragment information <1, ?> associated with the operator 'a' is now changed to <1, T>. This is because <1.2, T> is recorded in association with the operator 'b', and fragment 1.2 is a child of fragment 1 (Figure 4 d)).
- Fragment 1.1.1, <d FID=1.1.1 tsid=13>20</d>, arrives. The tsid of the root element in this fragment is 13. Consulting the tag structure, it is known that the set of tsid values of the elements contained in this fragment is {13}. Thus, the operator which needs to process this fragment is 'd'. Since the 'b' operator, the parent of 'd', has the fragment information <1.1, F>, fragment 1.1.1, a child of fragment 1.1, is discarded.

- Fragment 1.2.1, <d FID=1.2.1 tsid=13>40</d>, arrives. The same processing as for fragment 1.1.1 is done except that the data <d>40</d> out of the fragment is produced as a query result. This is because <1,T> and <1.2, T> are recorded in association with the operator 'a' and 'b', respectively, and fragment 1.2.1 is a descendant of fragment 1 and of fragment 1.2.

2.4 Comparison between HF Model and XFL

The main inefficiencies inherent in the HF model are two-fold: *space overhead* incurred because of the holes and inability to directly support the *descendant axis* of XPath. In this section, we explain them and describe why XFL overcomes them.

2.4.1 Hole Overhead

The overhead for the hole IDs and the hole elements could be very huge considering the typical structure of XML documents on the Web. According to [16], most of the XML data on the Web is not deep in height, and the heights of the 99 % of XML data on the Web are not greater than 8. The large volume of XML data is due to its great width (i.e., because of a number of repeating subtrees.) Consider the DTD in Figure 5 a). In an XML document that conforms to it would have as many 'book' subtrees as there are books on sale in the bookstore. For an XML fragmentation shown in Figure 5 b) where each book subtree is an XML fragment, there would be as many holes in the 'books' fragment as there are 'book' fragments.

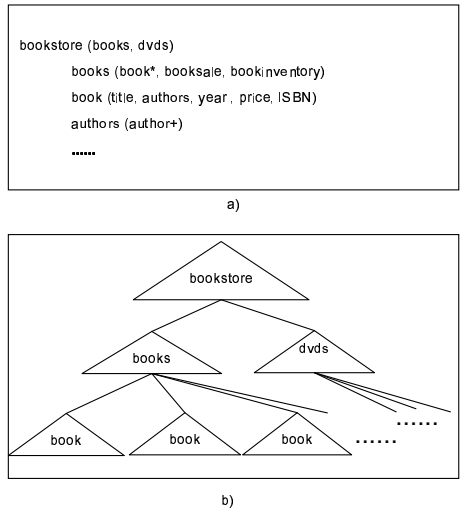


Fig. 5. Example of DTD and XML fragmentation; a) DTD, b) XML fragment tree

With XFL, in contrast, there is neither a hole nor a hole element in the fragments. As such, we could considerably reduce the size of the XML fragments (e.g.,

the 'books' fragment). In the first fragment of Figure 1 d), for example, there are 2 hole elements (i.e, <hole hid = 2 tsid = 11/ > and <hole hid = 3 tsid = 11/ >). Other hole elements, <hole hid = 4 tsid = 13/ > and <hole hid = 5 tsid = 13/ >, exist in the second and the third fragments. Let x be the size of a hole element, and n be the total number of holes created in XML fragmentation. (For example, $x = 21$ bytes and $n = 4$ in Figure 1 d). x could be reduced depending on the implementation.) Then, because of the hole elements, the size of the XML fragment stream with the HF model is greater than that with XFL by $x * n$. Let m be the average number of holes in a fragment. (For example, there are on the average 0.8 holes per fragment in Figure 1 d).) The average size of a fragment with the HF model is greater than that with XFL by $x * m$. If a fragment needs to be buffered in memory in its entirety on its arrival at a client, the size of the buffer with the HF model should be larger than that with XFL by $x * m$ on the average. Thus, the values of x and m would affect the feasible size of an XML fragment.

A more important improvement with XFL is that memory requirement at the client for query processing could be reduced considerably. In XFrag and XFPro with the HF model, the fragment information *accumulated* at the client memory during query processing consists mostly of hole IDs, their filler IDs, results of predicate evaluation, and potential query results that are temporarily buffered until their fate is determined. The hole IDs and their filler IDs are stored to figure out the parent-child relationship among fragments. With removing holes altogether in XFL, the volume of fragment information that needs to be accumulated could be considerably reduced. The *extra* space overhead incurred for bookkeeping of fragment information with the HF model during the entire course of query processing compared with that with XFL is $O(n)$ where n is the total number of holes created in XML fragmentation. Considering the aforementioned typical structure of XML documents on the Web [16], n could be very large, and thus, with XFL, we could save a good amount of memory in query processing.

2.4.2 Descendant Axis

A hole and its filler in the HF model represents a link from a parent fragment to its child one in the XML fragment tree. Thus, a sequence of the holes and their fillers form a parent-child *chain* of fragments. As such, given two fragments that are an ancestor and a descendant with each other, such a structural relationship could not be identified with the HF model until all the intermediate fragments connecting the two are fully streamed in. Because of this, the processing of the widely used descendant axis of XPath, usually abbreviated as //, would be inefficient with the HF model.[2] For a descendant axis, it needs to be resolved into all the possible paths that are expressed only with a sequence of child axes between the ancestor and the descendant elements involved. The final query result is a union of the results of all

[2] To be exact, // is the abbreviated notation for an XPath location step /descendant-or-self::node()/ [15].

XPath expressions with the resolved paths as stated in [11]. Consider an XPath query '/a//b' as an example. Suppose there are two different paths 'c/d/e' and 'd/f' between 'a' and 'b'. Then, the original query '/a//b' is resolved into two queries: '/a/c/d/e/b' and '/a/d/f/b'. All of these two rewritten expressions need to be evaluated, and their results are merged to produce the result of '/a//b'. Due to such a limitation with the HF model, the amount of memory required for query processing over a fragment stream at a client would increase.

With XFL, in contrast, given any pair of XML fragments, their structural relationship can be easily identified only with their fragment IDs without accessing other fragments connecting the two because the fragment IDs are assigned with XML labeling. Thus, with XFL, the descendant axis of XPath can be directly supported.

3 SCALABLE QUERY PROCESSING OVER XML FRAGMENT STREAM

According to the experimental results with the state-of-the-art techniques of query processing over XML fragment stream that employ the HF model [2, 11], the minimum amount of memory required to complete query processing at a client increase as the size of the stream increases.

An XML fragment stream may be very long and *dynamic*. It may be even *unbounded*. Some fragments are static (e.g., the fragment containing the root or non-repeating elements) while most fragments could be dynamically generated and appended to the XML document to be instantly disseminated. The fragments for up-to-second stock prices, hourly weather forecast, sensor readings, new books put on the shelf for sale in a bookstore, etc. are such examples. Figure 6 a) shows an example of an XML fragment tree where 'a' and 'b' fragments are static while 'c' and 'd' fragments are the ones dynamically generated. Figure 6 b) shows an example of an XPath query to be executed against the dynamic XML fragment stream out of the XML fragment tree of Figure 6 a). With a long or a dynamic stream which is possibly unbounded, its size will ever-increase as time goes by, and the amount of memory required for query processing at a client will also keep increasing. It might end up using up to the entire memory available so that the completion of query processing may not be guaranteed.

As such, for the state-of-the-art techniques of query processing over XML fragment stream to be of practical use, they need to be further *optimized* so that they are *scalable* with respect to the increase of the stream size. Ideally, they should be insensitive to the stream size, and the amount of memory required for query processing should be affected only by the complexity of the query not by the stream size. In this section, we propose a few optimization techniques for memory efficiency with XFL as well as with the HF model for fairness in comparison. As you can see shortly, XFL is much more amenable to such optimizations than the HF model.

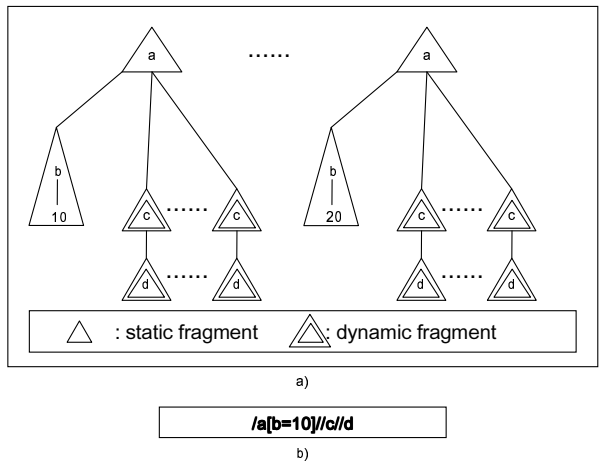


Fig. 6. Dynamic XML Fragment Stream; a) XML fragment tree, b) XPath Query

3.1 XPath Step Reduction

An XPath expression is a sequence of location steps [4]. Each step consists of *axis*, *node test*, and optional *predicates*. The delimiter between steps is a slash (/). For example, an XPath expression '/a/b[@c=1]/d' consists of 3 steps, 'a', 'b[@c=1]', and 'd'. Only the second step has a predicate given in brackets. The axes of all the three steps are 'child', which is omitted because it is the default of the 13 axes of XPath [4]. The axis in the predicate '[@c=1]' is 'attribute'. The abbreviated expression above can be fully given as '/child::a/child::b[attribute::c=1]/child::d'. The tag names 'a', 'b', and 'd' in the three steps and also 'c' in the predicate are for the *name test*, the most common form of node test in XPath.

Bookkeeping of fragment information during query processing over XML fragment stream needs to be done for each location step of the XPath query. As such, in general, the more the location steps in the query, the more memory would be required. With XFL, a given XPath expression could be rewritten into the one with fewer steps such that the two expressions produce the same result.

Given an XML document D and an XPath expression q, let $q(D)$ denote the result of processing q against D. Given a collection of XML fragments $\langle d_1, \ldots, d_k \rangle$ ($k \geq 1$) obtained out of D by the XML fragmentation function F_x with the tag structure τ and the method of representing XML fragmentation M_f as defined in Definition 1 (i.e., $F_x(D, \tau, M_f) = \langle d_1, \ldots, d_k \rangle$), let $q(\langle d_1, \ldots, d_k \rangle, \tau, M_f)$ denote the result of processing q against the XML fragment stream of $\langle d_1, \ldots, d_k \rangle$ regardless of the order of the fragments in the stream. With these notations, we have the following lemma:

Lemma 2 (Removal of Predicate-free Prefix Steps)**.** Let D, τ, F_x be an XML document, a tag structure for D, and the XML fragmentation function, respectively.

An XPath expression q of the form '$/a_1/a_2/\ldots/a_n/b[P]/\ldots/c$' where the step $a_i\,(i = 1,\ldots,n)$ does not have a predicate and P is a predicate, can be rewritten into a reduced one q_r of the form '$//b[P]/\ldots/c$' such that $q(D) = q_r(F_x(D,\tau,$ 'XFL'), τ, 'XFL').

Proof. Given two node instances x and y in D such that x is an ancestor of y, let $P(x,y)$ denote the path from x to y. Let r_D be the root node of D. Out of all the 'b' nodes in D, only those at the path $/a_1/a_2/\ldots/a_n/b$ are relevant to q. For any 'b' node contained in an XML fragment f, $P(r_D,b)$ is known because of the tag structure τ. More specifically, $P(r_D,b) = P(r_D,f_r)\cdot P(f_r,b)$ where f_r is the root element of f, and $\cdot$ denotes the path concatenation. Note that f_r has a tsid attribute. Consulting the tag structure τ with it, we can know $P(r_D,f_r)$. (For example, consider the last fragment in Figure 2 b) (i.e., <d FID = 1.2.1 tsid = 13 >40</d>). Looking up the tag structure in Figure 1 e) for the tag element whose id = 13, we can see that the full path leading to the 'd' node in the root element of the fragment at hand is /a/b/d.) $P(f_r,b)$ is known because it is within f at hand. As such, with the prefix path $/a_1/a_2/\ldots/a_n/b$ in q being reduced to $//b$ in q_r, the answer to q could be obtained with q_r. □

Lemma 2 means that bookkeeping of fragment information out of the path that forms any prefix steps of an XPath expression is not necessary if there is no predicate involved in it. For the step with a predicate, the bookkeeping is required to relate its result nodes that satisfy the predicate with the result nodes of its subsequent steps. We call this optimization *removal of predicate-free prefix steps*.

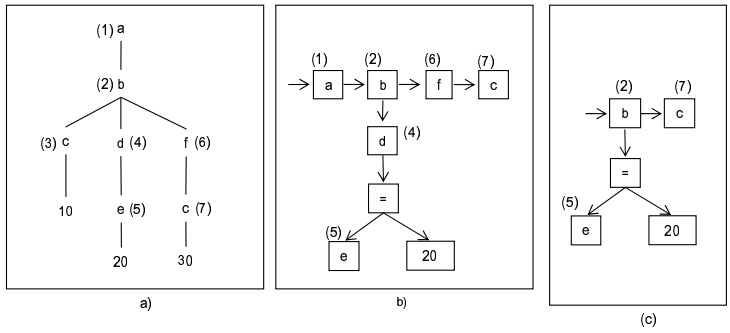


Fig. 7. Example of XPath step reduction; a) XML data with tag ID, b) fragment stream query processor, c) reduced fragment stream query processor

Example 1. Consider an XML data in Figure 7 a) where the value in parentheses beside each node is the tag ID as defined in the tag structure. XPath expressions /a/b/d/e, /a/b[c=10]/d/e, /a/b[d/e=20]/f/c against it can be reduced to //e, //b[c=10]/d/e, //b[d/e=20]/f/c, respectively.

The removal of predicate-free prefix steps can be further generalized as given in the following lemma:

Lemma 3 (Removal of Predicate-free Steps)**.** Let D, τ, F_x be an XML document, a tag structure for D, and the XML fragmentation function, respectively. An XPath expression q of the form '$/S_1/b_1[P_1]/S_2/b_2[P_2]/\ldots/S_n/b_n[P_n]/S_{n+1}/c$' where $S_i\,(i = 1,\ldots,n+1)$ is a sequence of location steps without any predicate, $P_i\,(i = 1,\ldots,n)$ is a predicate, and each of $b_i\,(i = 1,\ldots,n)$ and c is a location step, can be rewritten into a reduced one q_r of the form '$//b_1[P_1']//b_2[P_2']//\ldots//b_n[P_n']//c$' where $P_i'\,(i = 1,\ldots,n)$ is a predicate obtained by removing all the predicate-free steps in P_i, such that $q(D) = q_r(F_x(D,\tau,$ 'XFL'$),\tau,$ 'XFL'$)$.

Proof. Let $q_1 = //b_1[P_1]/S_2/b_2[P_2]/\ldots/S_n/b_n[P_n]/S_{n+1}/c$. We know that $q(D) = q_1(F_x(D,\tau,$ 'XFL'$),\tau,$ 'XFL'$)$ because of Lemma 2.
Let $q_i = //b_1[P_1]//b_2[P_2]//\ldots//b_i[P_i]/S_{i+1}/b_{i+1}[P_{i+1}]/\ldots/S_n/b_n[P_n]/S_{n+1}/c\,(i = 1,\ldots,n)$. That is q_i is obtained from q by removing the predicate-free steps S_1, S_2, ..., S_i. Suppose $q(D) = q_i(F_x(D,\tau,$ 'XFL'$),\tau,$ 'XFL'$)$. Then we can show that $q_i(F_x(D,\tau,$ 'XFL'$),\tau,$ 'XFL'$)$=$q_{i+1}(F_x(D,\tau,$ 'XFL'$),\tau,$ 'XFL'$)$.

Out of all the 'b_{i+1}' nodes in D that satisfy the predicate condition P_{i+1}, only those satisfying the following two conditions are relevant to q:

1. It is at the path $/S_1/b_1/S_2/b_2/\ldots/S_i/b_i/S_{i+1}/b_{i+1}$.
2. It is a descendant of a 'b_i' node which satisfies the predicate P_i.

For any 'b_{i+1}' node contained in an XML fragment f, the condition (1) can be checked because $P(r_D, b_{i+1})$ is known through the tag structure τ as stated in the proof of Lemma 2. As for the condition (2), even if the ancestor 'b_i' node that satisfies the predicate P_i is contained in a fragment f_a other than f, the ancestor-descendant relationship between 'b_i' and 'b_{i+1}' can be identified because of the FIDs of f_a and f with XFL. Thus, $q_i(F_x(D,\tau,$ 'XFL'$),\tau,$ 'XFL'$)$=$q_{i+1}(F_x(D,\tau,$ 'XFL'$),\tau,$ 'XFL'$)$, and by induction, $q(D) = q_{n+1}(D,\tau,$ 'XFL'$),\tau,$ 'XFL'$)$.

The proof for the reduction of P_i to $P_i'\,(i = 1,\ldots,n)$ is omitted because the same logic as above is applied. Hence, $q(D) = q_r(F_x(D,\tau,$ 'XFL'$),\tau,$ 'XFL'$)$. □

Example 2. Consider the XML data in Figure 7 a).

XPath expression /a/b[d/e=20]/f/c can be reduced to //b[.//e=20]//c. Figure 7 b) shows its fragment stream query processor in the current state-of-the-art whereas Figure 7 c) shows its reduced counterpart.

Lemma 3 means that bookkeeping of fragment information out of the path that forms any predicate-free subsequence of steps of an XPath expression is not necessary. For the steps with a predicate, the bookkeeping is required to relate among their result nodes that satisfy their respective predicates. As an example, for an XPath expression '$//b_1[P_1]//b_2[P_2]//\ldots//b_n[P_n]//$c', let us consider the two instances of 'b_i' to 'b_{i+1}' paths in Figure 8 where only the left one is qualified for '$b_i[P_i]//b_{i+1}[P_{i+1}]$'. What is essential here for correct query processing is to figure

out if the left 'b_{i+1}' is a descendant of the left 'b_i' but not of the right one, and the other way round for the right 'b_{i+1}'. As such, the bookkeeping of the IDs of the fragments that contain the instances of 'b_i' and 'b_{i+1}' is a necessary condition for query processing, and also sufficient because with XFL, the ancestor-descendant relationship between two fragments can be identified only with their FIDs. This general form of XPath step reduction is called *removal of predicate-free steps*.

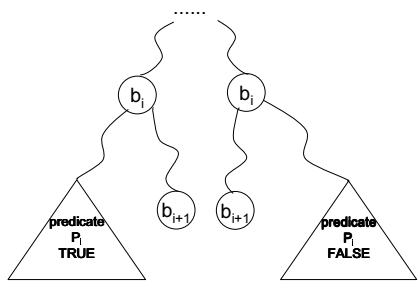


Fig. 8. Example for removal of predicate-free steps

Meanwhile, are these optimizations applicable with the HF model as well? The removal of predicate-free prefix steps is applicable for the same reason as with XFL but not the removal of predicate-free steps. The reason for the latter can be explained with the aforementioned example in Figure 8. Without the fragment information on the full parent-child chain through the hole IDs and the filler IDs along the 'b_i' to 'b_{i+1}' paths, it is impossible to figure out if the left 'b_{i+1}' is a descendant of the left 'b_i' but not of the right one, and the other way round for the right 'b_{i+1}' unless all the 'b_i's and 'b_{i+1}'s belong to the same fragment.

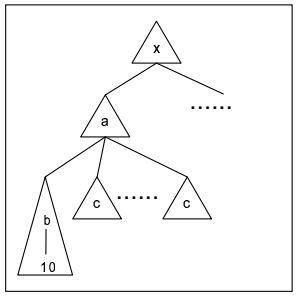
3.2 Deletion and No Bookkeeping of Useless Fragment Information

With the XPath step reduction, memory efficiency in query processing would be enhanced because no bookkeeping is done for the removed steps; but the XPath step reduction alone does not yet guarantee the scalability of query processing.

3.2.1 Child Counting

The major reason why more and more memory is used up as query processing continues is that the fragment information is accumulated but not deleted even after it has become of no use for query processing any more. As an example, consider (a) an XML fragment tree and (b) an XPath query Q_1 in Figure 9. All the information on the fragments 'a', 'b', and 'c' in Figure 9 a) could be deleted because all the 'c' fragments are *not* qualified for Q_1 (the text node of 'b' element is 10 not 20 to satisfy the query predicate). However, none can be deleted because a new 'b' fragment,

‘20’ at the path /x/a/b, may be streamed in later in order to make all the ‘c’ fragments qualified for Q_1. But what if no such element, ‘20’, will ever come?



a)

Q1 : /x/a[b=20]/c

b)

Fig. 9. Example for child counting; a) XML fragment tree, b) XPath Query

For scalability, the fragment information useless for query processing should be deleted or not kept in the first place. As for the aforementioned case in the example of Figure 9, if there were a way to know that *all* the child fragments of ‘a’ fragment had been streamed in, we could have deleted all the accumulated information on the ‘a’, ‘b’, and ‘c’ fragments all at once. One simple solution is to let every fragment record the number of its child fragments though it cannot be applied when the stream of child fragments is unbounded. This *child fragment counting* alone could considerably help reduce the memory requirement at a client. It can be used both with XFL and with the HF model. With the latter, the child fragment counting amounts to the *counting of the holes* in a fragment.

3.2.2 Deletion and No Bookkeeping of Information on Disqualified Fragments

More delicate case where the useless fragment information can be deleted is shown in Figure 6. Suppose that the ‘a’ and ‘b’ fragments in the XML fragment tree of Figure 6 a) are static whereas the ‘c’ and ‘d’ ones are dynamically generated. There are two ‘a’ fragments and two ‘b’ fragments shown, and all the ‘d’ fragments that are the descendants of the right ‘a’ are *not* qualified for the XPath query of Figure 6 b). Thus, on arrival of the right ‘b’ fragment (i.e., 20), all the accumulated information on the ‘c’ and ‘d’ fragments that has already arrived as the descendants of the right ‘a’ can be deleted. Besides, from then on, no bookkeeping of any information is necessary for the ‘c’ and ‘d’ fragments that are yet to come as the descendants of the right ‘a’.

This technique of *deletion and no bookkeeping of information on the disqualified fragments* can be very easily applied with XFL because the ancestor-descendant relationship between two fragments can be identified only with their fragment IDs. In the above example, the fragment ID of the left 'a' is kept with the mark of its predicate having been evaluated true whereas the fragment ID of the right 'a' is kept with the mark of its predicate having been evaluated false. Then, all the 'c' and the 'd' fragments that are yet to come can be properly dealt with depending on which 'a' is their ancestor. With the HF model, however, it could only be applied in a very inefficient way, resulting in virtually no benefit. The information on a 'c' fragment, in the above example, cannot be deleted until the information on its child 'd' fragment is deleted first. Otherwise, when a 'd' fragment arrives, it becomes a *dangling* fragment. There is no way to figure out if its ancestor is the left 'a' or the right 'a' without the information on its parent 'c' fragment. As such, the information on the full parent-child chain through the involved hole IDs and the filler ID needs to be kept first before any part of it can be deleted; and, in deletion, the order matters. It is always strictly from descendant to ancestor. Thus, even when a fragment which is known to be disqualified arrives, its filler ID and hole IDs need to be still kept in order to connect its ancestor to its descendants. We cannot expect that a child fragment instance arrives right after its parent fragment instance not just because the order of arrival of the fragments at a client could be arbitrary but because the order of dynamic generation and dissemination of XML fragments may be that all the parent fragments with holes in them (e.g., all the 'c' fragments in Figure 6 a)) precede that of all their fillers (e.g., all the 'd' fragments in Figure 6 a)). As such, the precious memory could be temporarily used up to store the information of disqualified fragments such as the 'c' fragments below the right 'a'. Even if it could be eventually deleted, such *too late* deletion does not help reduce the amount of the *minimum* memory required for query processing. With XFL, in contrast, there could be no dangling fragment, the order does not matter at all in deleting the fragment information, and no bookkeeping is necessary at all for the fragments which are known to be disqualified for the query.

3.3 Hole Sharing

The original HF model cannot represent a dynamic XML fragment stream while XFL easily can. Suppose a fragment f contains the repeating elements that are to be dynamically generated and streamed as filler fragments. Then, f needs to arrange holes to accommodate them before f itself is streamed. But how many? Simply reserving a predetermined number of holes may turn out to be a waste, just incurring the hole overhead; or a shortage when the number of dynamic fragments exceeds the number of reserved holes (see Figure 10 a)). To cope with this situation along with memory efficiency in mind, the HF model could be extended by employing an XML labeling scheme as in XFL. Only *one* hole is reserved to be shared by all the dynamically generated filler fragments. With the ID of the shared hole being 'p', the filler IDs become 'p.1', 'p.2', ... with a prefix labeling scheme (see Figure 10 b)).

This *hole sharing* would considerably alleviate the hole overhead of the HF model, and thus reduce the amount of the memory required for query processing.

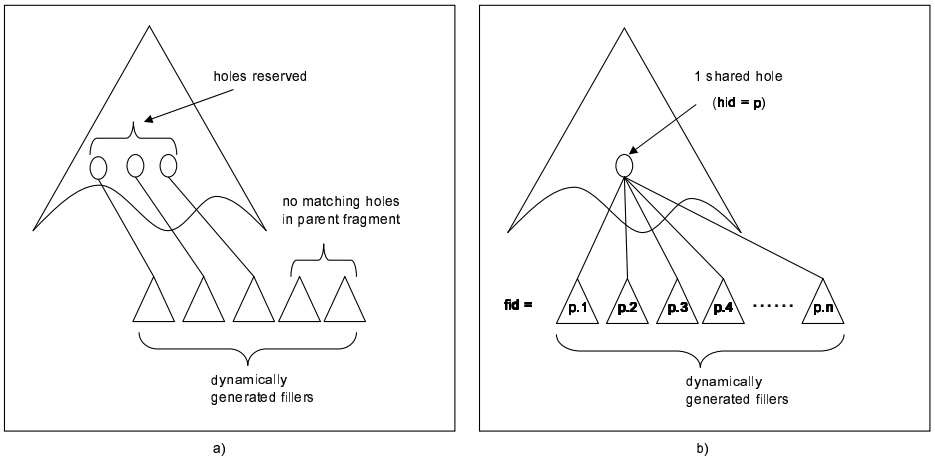


Fig. 10. Hole sharing

The hole sharing is applicable only to the HF model. With XFL, no special arrangement such as the hole sharing is needed to represent a dynamic XML fragment stream. Suppose a fragment f contains the repeating elements that are to be dynamically generated and streamed as filler fragments. With f labeled as L_p with an XML labeling scheme, each of the dynamically generated fragments, f_c, is given a label L_c in a way that f_c can be identified as a child of f given two labels L_p and L_c.

Although the hole sharing with the HF model employs an XML labeling scheme as in XFL, it is not exactly the same as XFL. First, the shared hole in a fragment is still a *hole*, which needs to be represented as a hole element with its own hole ID and tsid as defined in Section 2.1 and as depicted in the first three fragments of Figure 1 d) for example. With XFL, there is no hole in any fragment, and thus, no hole ID. Secondly, in the hole sharing with the HF model, the labels are assigned as the ID of the shared hole and as the IDs of its fillers such that their parent-child relationship can be identified. For a pair of a conventional non-shared hole and its singleton filler, their parent-child relationship is identified when the hole ID and the filler ID are equal to each other. For a shared hole with L_h as its hole ID and one of its fillers with L_f as its filler ID, their parent-child relationship is identified when L_h is a prefix of L_f. With XFL, the labels are assigned to the fragments so that the structural relationship among the fragments can be identified.

4 IMPLEMENTATION AND PERFORMANCE EVALUATION

We have implemented *four* versions of the techniques of query processing over XML fragment stream in Java using J2SE Development Kit 5.0 Update 11. They are called *HF-b*, *HF-o*, *XFL-b*, and *XFL-o*, where -b stands for *basic* whereas -o for *optimized*. HF-b is the implementation of XFrag as described in [2] plus its improvement proposed in XFPro as described in [11]. Both XFrag and XFPro are based on the HF model, so is HF-b; but it is not with the optimization techniques of Section 3. HF-o is the same as HF-b except that it is optimized with the applicable techniques of Section 3. XFL-b (XFL-o), which is based on XFL instead of the HF model, is the counterpart of HF-b (HF-o).

The performance experiments were conducted in a system with Intel Dual Core CPU 6600 2.40 GHz and 2 GB memory on Windows XP Professional. A total of 10 XPath queries, Q1 through Q10 as listed in Table 1, were employed. The XML stream sources against which the client queries were processed were *auction.xml*'s of 5 different sizes generated by xmlgen of XMark benchmark [20]. Their sizes are 11.3 MB, 22.8 MB, 34 MB, 45.3 MB, and 56.2 MB, respectively. For queries Q1 through Q6, we assumed that these streams were bounded and thus we turned the child counting *on* during query processing with XFL-o and with HF-o. For the remaining queries, Q7 through Q10, we assumed that these streams were unbounded and thus we turned it *off* as mentioned in Section 3.2.1. Though each of these streams was of fixed size, whether it was bounded or not could be simulated in such a way.

We first compared XFL-b with HF-b in their memory requirement. The size of the auction.xml used was 56.2 MB. Figure 11 shows the results where the y-axis denotes the memory requirement for query processing. XFL-b required much less memory than HF-b in processing of all the queries. Figure 12 compares the memory requirements of XFL-b and HF-b for the query Q1 as the stream size increases. Although XFL-b required much less memory than HF-b in all XML stream sizes, both basic versions turned out *not* to be scalable with respect to the increase of the stream size. Their memory requirements increased as the stream size increased. As such, both would not be of practical use when the stream size is large and the client has only limited amount of memory.

Would the desired scalability be achieved with the optimization techniques of Section 3? Figure 13 shows a) the comparison between HF-b and HF-o and b) that between XFL-b and XFL-o for Q1. The optimization techniques of Section 3 were revealed as effective for both XFL and the HF model, reducing memory requirement considerably; but their effectiveness was greater with XFL than with the HF model, and more importantly, the desired scalability was observed only with XFL-o. Figure 14 compares the memory requirement of XFL-o and that of HF-o for all the queries in Table 1 as the stream size increases. Note that for queries Q1 through Q6 with the bounded streams, XFL-o is *not* sensitive to the increase of the stream size whereas HF-o still is. For queries Q7 through Q10 with the unbounded streams, the

scalability was not observed even with XFL-o. This is because the child counting was not used.[3]

These results mean that the technique of query processing over XML fragment stream with XFL is scalable with respect to the stream size increase as long as the stream is bounded, and thus it is of practical use whereas that with the HF model is not.

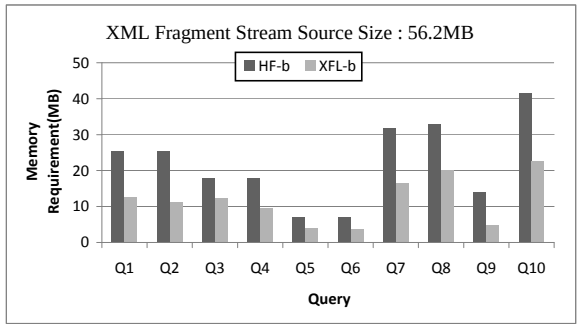


Fig. 11. Memory requirment in query processing (HF-b vs. XFL-b)

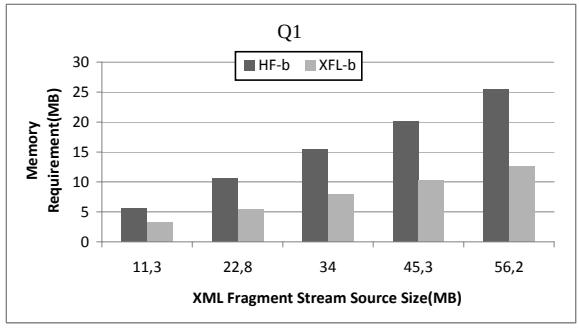


Fig. 12. Memory requirement in processing Q1 (HF-b vs. XFL-b)

[3] The meaningful queries against an unbounded stream are usually the *continuous* ones which often include some timing semantics or constraints (e.g., sliding windows). We did not consider such a thing in the experiments. For continuous queries against an unbounded stream, the child counting needs to be extended to incorporate the timing semantics or constraints so that it may be used in query processing for memory efficiency. Such an extension is beyond the scope of this paper.

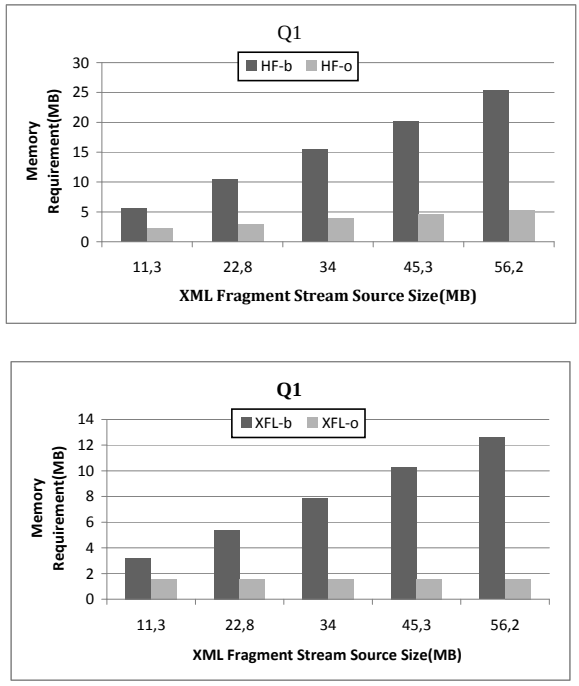


Fig. 13. Memory requirement in processing Q1 (Basic vs. optimized versions)

5 RELATED WORK

The HF model and the tag structure were proposed in [3, 6] as the components of *XStreamCast* whereby the source XML data is fragmented and streamed to a number of clients where queries are processed over the stream. The tag structure plays the same role as the well-known *DataGuide* [8] which summarizes the path structure of

Q1	/site/people/person[name="Claudine Nunn"]/watches/watch
Q2	/site/people/person[name="Claudine Nunn"]//watch
Q3	/site/people/person[name="Torkel Prodrodmidis"]/profile/interest
Q4	/site/people/person[name="Torkel Prodrodmidis"]//interest
Q5	/site/closed_auctions/closed_auction[price>"100"]/type
Q6	/site/closed_auctions/closed_auction[price>"200"]//author
Q7	/site/open_auctions/open_auction[initial>"200"]/bidder/time
Q8	/site/open_auctions/open_auction/bidder[increase>"200"]/time
Q9	/site/open_auctions/open_auction[initial>"200"]//start
Q10	/site/open_auctions/open_auction[initial≤"500"]/bidder[increase>"200"]/time

Table 1. XPath queries in experiments

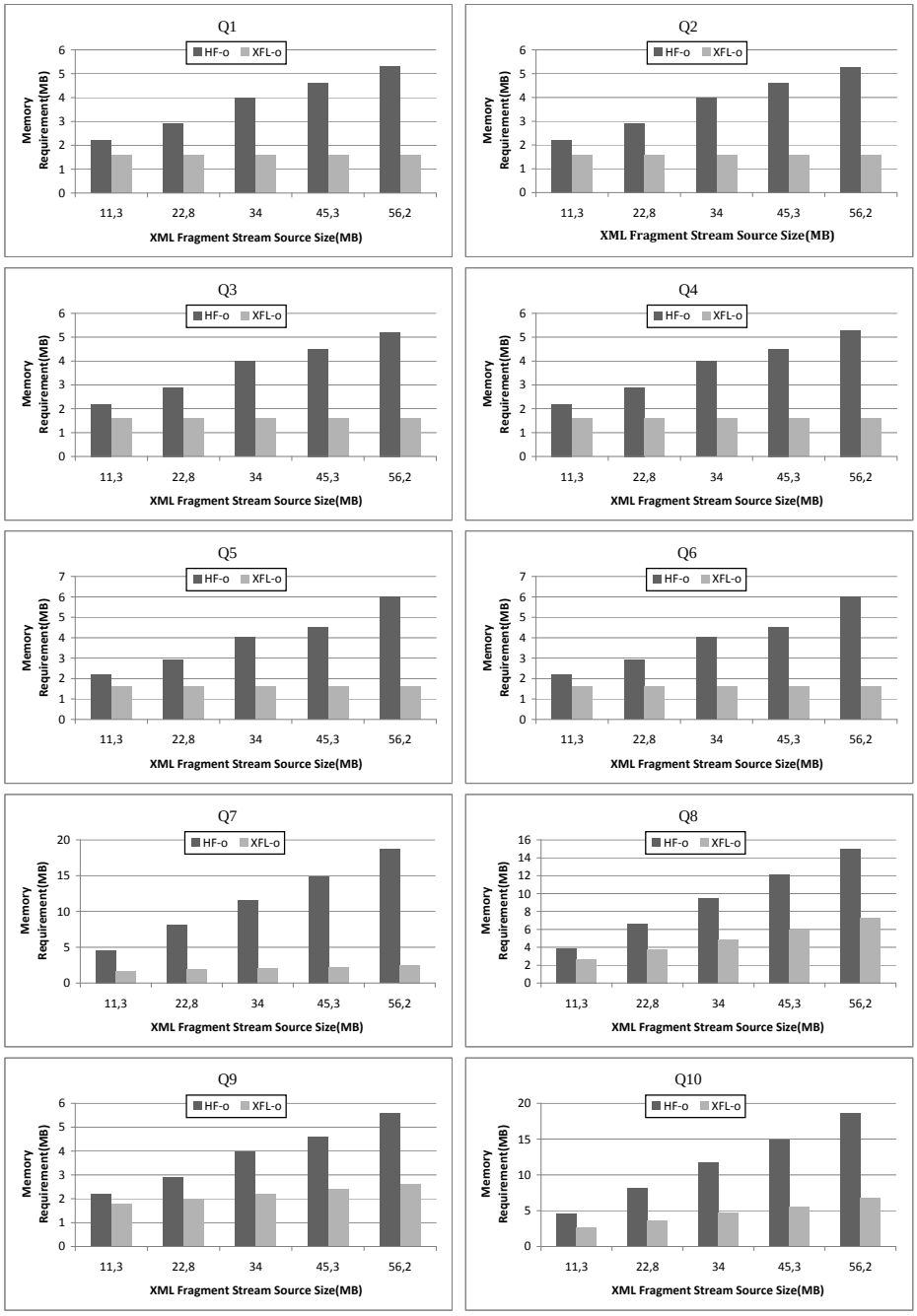


Fig. 14. Memory Requirement in Query Processing (HF-o vs. XFL-o)

a semistructured data except that the former also specifies the XML fragmentation schema. It also corresponds to a part of the *fragment context specification* defined as a core component of the *XML Fragment Interchange* specified by the W3C XML Fragment Working Group [9].

XFrag [2] is the first proposed framework of query processing over XML fragment stream based on the HF model. XFPro [11] which is based on the XFrag framework proposed techniques and data structures to expedite query processing compared with XFrag. However, its main focus was on query processing time, not on memory efficiency. Memory efficiency with XFPro and that with XFrag were comparable. The historical XML data management with the HF model was investigated in [1]. Methods of XML fragmentation with the HF model for XML fragment stream query processing based on query frequencies were proposed in [10].

Much work has been done on XML stream query processing. They are either for XML document stream filters (e.g., YFilter [5], FiST [12]) or conventional XML query processing against the XML stream not against the stored XML (e.g., XSQ [19], BEA/XQRL [7]). These works are different from the work on query processing over XML fragment stream. First, in the XML stream filters, the goal is basically not for the conventional query processing. The role is reversed in the sense that queries are selected when they are matched against the streaming XML documents rather than the streaming XML data is retrieved. Secondly, XML fragmentation and the relationship among the fragments are not considered. Finally and most importantly, the amount of memory available for query processing is not assumed to be so limited as in our work though memory efficiency in stream query processing is basically a requirement.

6 CONCLUDING REMARKS

In this paper, we addressed query processing over XML fragment stream in portable/hand-held devices widely deployed in the mobile and pervasive computing environment. Since those devices are usually with limited memory, the most important technical goal is to devise a method to complete query processing with as little memory as possible. In this paper, we presented and verified a considerable improvement of the state-of-the-art techniques so that they could now be practically viable.

The *contributions* of this paper are two-fold. First, we employed *XML fragment labeling(XFL)* as a method of representing XML fragmentation and correlating among the XML fragments, and showed that XFL is much more effective than the popular hole-filler model employed by the state-of-the-art techniques in reducing memory requirement in query processing. Secondly, we proposed several techniques like *XPath step reduction* and others to optimize query processing with XFL, showing that query processing with XFL is *scalable* with the increase of the stream size in the sense that the memory requirement of query processing is not affected by the stream size as long as the stream is bounded.

As a future work, we are currently investigating the methods of fragmenting an XML document with XFL. This is to study the effect of XML fragmentation on resource requirements in the client devices in fragment stream query processing.

Acknowledgement

This work was supported by the Basic Research Program of the Korea Science and Engineering Foundation (grant No. R01-2006-000-10609-0). The authors also thank Prof. C. Lee at Chung-Ang University for his valuable comments and help.

REFERENCES

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

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

[3] 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.

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

[5] Diao, Y.—Altinel, M.—Franklin, M.—Zhang, H.—Fischer, P.: Path Sharing and Predicate Evaluation for High-Performance XML Filtering. ACM Trans. on Database Systems, Vol. 28, Dec. 2003, No. 4, pp. 467–516.

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

[7] Florescu, D.—Hillery, C.—Kossmannn, D.—Lucas, P.—Riccardi, F.—Westmann, T.—Carey, M.—Sundararajan, A.—Agrawal, G.: The BEA/XQRL Streaming XQuery Processor. Proc. Int'l Conf. on VLDB, 2003.

[8] Goldman, R.—Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. Proc. Int'l Conf. on VLDB, 1997.

[9] Grosso, P.—Veillard, D. (Eds.): XML Fragment Interchange (XFI). W3C Candidate Recommendation, Feb. 2001, `http://www.w3.org/TR/xml-fragment`.

[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.: Efficient Query Processing for Streamed XML Fragments. Proc. Int'l Conf. on DASFAA, 2006.

[12] Kwon, J.—Rao, P.—Moon, B.—Lee, S.: FiST: Scalable XML Document Filtering by Sequencing Twig Patterns. Proc. of Int'l Conf. on VLDB, 2005, pp. 217–228.

[13] Lee, S.—Kim, J.—Kang, H.: XFLab: A Technique of Query Processing over XML Fragment Stream. Proc. the 24th British Int'l Conf. on Databases (BNCOD2007), Glasgow, U.K., July 2007, pp. 182–186.

[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.

[15] Li, Q.—Moon, B.: Indexing and Querying XML Data for Regular Path Expressions. Proc. Int'l Conf. on VLDB, 2001.

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

[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] Peng, F.—Chawathe, S.: XPath Queries on Streaming Data. Proc. ACM SIGMOD Int'l Conf. on Management of Data, 2003, pp. 431–442.

[20] 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.

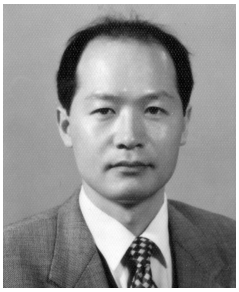
[21] Strategy Analytics: `http://www.strategyanalytics.com`.

[22] Wu, X.—Lee, M.—Hsu, Y.: A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. Proc. Int'l Conf. on Data Engineering, 2004, pp. 66–77.

[23] Zhang, C.—Naughton, J.—DeWitt, D.—Luo, Q.—Lohman, G.: On Supporting Containment Queries in Relational Database Management Systems. Proc. ACM SIGMOD Int'l Conf. on Management of Data, 2001, pp. 425–436.

Sangwook Lee 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 stream data management.

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 received 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.