Subtree-based XML Storage and XPath Processing

Ki Hoon Shin and Hyunchul Kang
School of Computer Science and Engineering, Chung-Ang University
Seoul, 156-756, Korea
[e-mail: koronya@gmail.com, hckang@cau.ac.kr]
*Corresponding author: Hyunchul Kang

Received August 27, 2010; revised September 5, 2010; accepted September 7, 2010; published October 30, 2010

Abstract

The state-of-the-art techniques of storing XML data, modeled as an XML tree, are *node*-based in the sense that they are centered around *XML node labeling* and the storage unit is an XML node. In this paper, we propose a generalization of such techniques so that the storage unit is an XML *subtree* that consists of one or more nodes. Despite several advantages with such generalization, a major problem would be inefficiency in XPath processing where the stored subtrees are to be parsed on the fly in order for the nodes inside them to be accessed. We solve this problem, proposing a technique whereby *no* parsing of the subtrees involved in XPath processing is needed at all unless they contain the nodes of the final query result. We prove that the correctness of XPath processing is guaranteed with our technique. Through implementation and experiments, we also show that the overhead of our technique is acceptable.

Keywords: XML storage, XPath processing, XML node labeling, XML tree

This research was supported by the Chung-Ang University Research Grants in 2010.

DOI: 10.3837/tiis.2010.10.0010

1. Introduction

Over the last decade, XML has been established as the standard for data representation and exchange on the Internet. One of the fundamental problems in XML data processing is to store XML data and process queries against it. The state-of-the-art technique of storing XML data has been centered around *XML node labeling*, described as follows: An XML document is modeled as a tree. **Fig. 1-(a)** and **(b)** shows a sample XML data and its XML tree. Three types of XML nodes are shown: element, attribute, and text. **Fig. 2** shows the XML tree of **Fig. 1-(b)** where each node is labeled, for example, with the well-known *range numbering scheme* of [1] and the nodes with the same element tag/attribute name are distinguished with a subscript number (e.g., b_1, b_2) just for exposition below. Each label denotes the *(begin:end) range* and the *level* of a node. For example, (2:21,2) labeled to b_1 denotes that the (begin:end) range assigned to b_1 is [2,21] and that the level of b_1 in the tree is 2. Those labels are for *structural join* [1][2][3][4] in XPath processing. For any two nodes n_1 and n_2, their *structural relationship* (ancestor-descendant or parent-child) can be determined with their labels. If the range of n_1 contains that of n_2, n_1 is an ancestor of n_2. If the levels of an ancestor-descendant pair differs by 1, one is the parent of the other. In **Fig. 2**, for example, b_1 is an ancestor of d_2 because the range of b_1 (2:21) contains that of d_2 (16:19) whereas b_1 is not an ancestor of d_4 because the range of b_1 (2:21) does not contain that of d_4 (36:39).

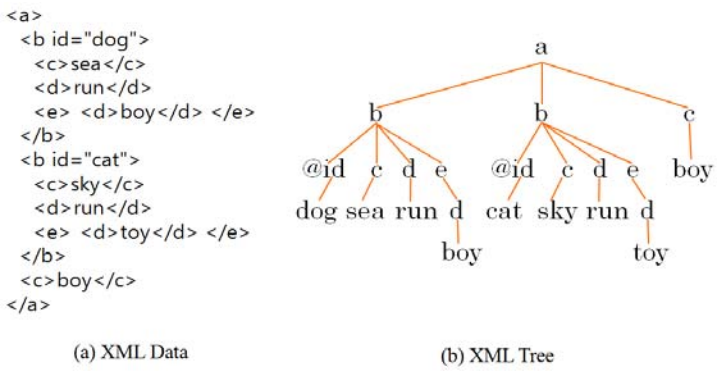

Fig. 1. Sample XML Data and XML Tree

In storing an XML tree, it is shredded into XML nodes each of which is separately stored. The stored items associated with each node include its element tag name or attribute name (prefixed with @) or text/attribute value depending on its node type, its (begin:end) range, and its level if the range numbering scheme of [1] is employed.[1] Often these nodes are indexed on the element tag/attribute name, on the text/attribute value, and on the "begin" value which is the unique identifier of each node. The former two types of indices are called *name index* and *value index*, respectively [4].

Put in a nutshell, the current state-of-the-art technique of storing XML data is *node-based*. The unit of storage is an XML node. In this paper, we investigate a *generalization* of such a

[1] Throughout this paper, we assume that the range numbering scheme of [1] is employed as the XML node labeling scheme. That choice is just for exposition of the concept. In fact, the technique proposed in this paper is *independent* of the XML node labeling scheme employed. Any scheme in the literature including the immutable ones for the dynamic XML documents can be employed for the technique proposed in this paper.

technique. A *subtree* of the XML tree is considered as the unit of XML storage. For example, **Fig. 3** shows a fragmentation of the XML tree of **Fig. 1-(b)** into subtrees each of which is marked by a triangle. Each subtree as a whole is stored as a unit of storage. At one extreme of the subtree-based XML storage, the entire XML tree could be stored as a whole without being shredded. At the other extreme, every XML node could be separately stored as in the conventional node-based technique. In this extreme, each node *n* constitutes a subtree which consists of only *n*. In between, a subtree can consist of one or more nodes. Such a generalization is desirable for the following reasons:

1. A semantic unit of XML data matches the storage unit. For example, an employee element which includes employee ID, first name, last name, department, e-mail, and so on as subelements or attributes can be stored as a whole without being shredded. The whole XML subtree can be retrieved, transmitted and so on without re-constructuring the constituent XML nodes. Handling and exchanging XML data in subtrees is particularly essential in the applications like web service [5].
2. More efficient mapping of XML data into other data models such as relational or object-relational is possible. This problem received much attention [6][7] because XML data, in its semistructured nature, usually contains irregular portions, which cannot be efficiently mapped to relational or object-relational schema. Such overflow data can be stored as a subtree.
3. In the conventional node-based storage, there are quite many stored nodes produced for a large volume of XML data. The space required to store every node separately with its label could be large. More compact storage is possible with the subtree-based storage. The shredding of XML data into nodes is for efficient query processing. As for cold data against which queries are infrequent, it could be stored as a subtree, saving space.

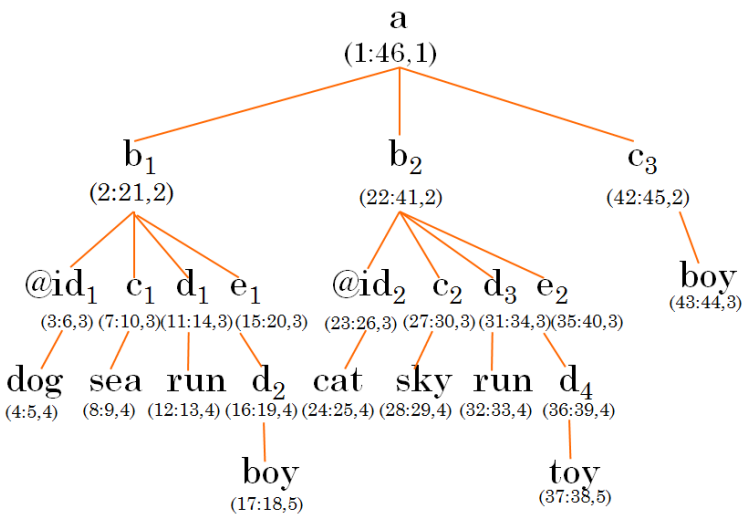

Fig. 2. XML Tree with Range Numbering

Despite these advantages, a major problem with subtree-based XML storage is *inefficiency in query processing*. Inefficiency arises when the XML nodes within a subtree need to be accessed. The subtree needs to be parsed on the fly. For example, in processing an XPath expression *e*, */a/b[@id='dog']//d*, against the XML document of **Fig. 1-(a)** which is stored in subtrees as in **Fig. 3**, it is necessary to parse the subtree rooted at b_1 to access the attribute $@id_1$

which resides inside the subtree. More subtrees need to be parsed in processing *e*. If the stored subtrees are eventually to be parsed for query processing, the afore-mentioned advantages are irrelevant or compromised.

In this paper, we propose a solution whereby *no* parsing of the stored subtree is needed at all in XPath processing unless the nodes of the final query result (i.e., node set) are to be retrieved from inside the subtrees. Such a solution is worth reporting because with it the subtree-based XML storage which has not been considered before in the literature would become a viable generalization of the conventional node-based XML storage.

The rest of this paper is organized as follows: In Section 2, we describe the subtree-based XML storage with the XPFS tree, which is the XML path and fragmentation schema needed in our proposed solution. In Section 3, we describe the conventional node-based XPath processing. In section 4, we present our technique of XPath processing where not any of the involved subtrees are parsed unless they contain the nodes of the final query result. We also prove that our technique guarantees the correctness of query processing. In Section 5, we report the implementation and experimental results, showing that the overhead of our solution is acceptable. In Section 6, we present related work. Conclusions and future work will be given in Section 7.

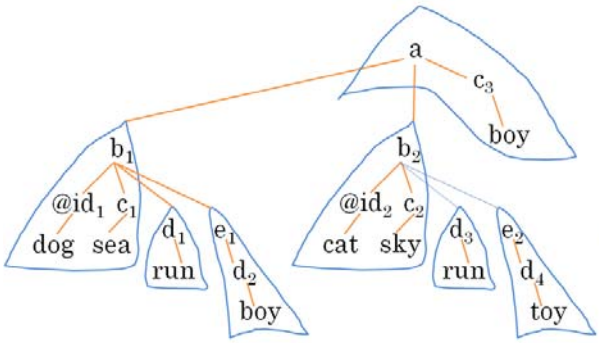

Fig. 3. Fragmentation of XML Tree into Subtrees

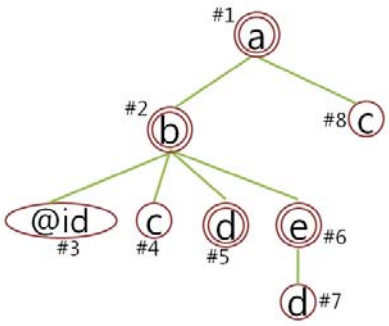

Fig. 4. XPFS Tree

2. Subtree-based XML Storage with XPFS Tree

To store XML data in subtrees and process XPath queries without parsing the subtrees involved, we need an *XML path schema* and an *XML fragmentation schema*. The former stores every *path type*, a sequence of element tag/attribute names, appearing in an XML document *D* with their identifiers (i.e., *path ID*). They are identified while *D* is parsed for its nodes to be labeled. The path schema can be represented as a tree as shown in **Fig. 4**, which is derived for the XML tree of **Fig. 1-(b)**. The number (prefixed with #) beside each node *N* is the path ID of *N*, which is the path ID assigned to the path type from the root to *N*. For example, in **Fig. 4**, the path type */a/b/c* is assigned #4 as its path ID.

As for the fragmentation schema, it specifies how the original XML tree is fragmented into subtrees. The fragmentation schema can be incorporated into the path schema, resulting in an *XML path and fragmentation schema(XPFS) tree* as shown in **Fig. 4**. There are two types of nodes in an XPFS tree. One is marked with a circle, and the other with a *double* circle. A double circle node denotes the root of a subtree to be stored while a single circle one denotes a node that belongs to a subtree *t* as a descendant of the root of *t*. In other words, given a double circle node *N*, *N* and all its single circle descendants constitute a subtree to be stored. For example, the fragmentation shown in **Fig. 3** conforms to the XPFS tree of **Fig. 4**. The nodes '*a*',

'*b*', '*d*', and '*e*' are in double circle in **Fig. 4**, and each of their instances is the root of the subtree marked by a triangle in **Fig. 3**.

For an XML document *D* and its XPFS tree, *D* is shredded into XML subtrees. Each subtree *t* with *r* as its *root node* is stored separately as a storage unit. The stored items associated with *t* include the element tag/attribute name of *r*, the (begin:end) range and the level of *r*, path ID of *r*, and *t* itself as a plain text not in any parsed format. Though *t* is not parsed and thus not accessed during XPath processing in our technique, we leave a special symbol wherever a subtree is fragmented out of *t*. For example, **Fig. 5** shows the subtree-based storage of the XML tree of **Fig. 1-(b)**, which is fragmented as in **Fig. 3**. Seven subtrees are stored (subtree *no.* (1) through (7) in **Fig. 5**. The *no.*column is not stored. It is just for reference.) The subtrees (1), (2), and (5) contain the symbol "$" as a marker for a subtree.

no.	name	pathid	begin	end	value
(1)	a	#1	1	46	<a>$$<c>boy</c>
(2)	b	#2	2	21	<c>sea</c>$$
(3)	d	#5	11	14	<d>run</d>
(4)	e	#6	15	20	<e><d>boy</d></e>
(5)	b	#2	22	41	<c>sky</c>$$
(6)	d	#5	31	34	<d>run</d>
(7)	e	#6	35	40	<e><d>toy</d></e>

Fig. 5. Subtree-based XML Storage

For comparison, **Fig. 6** shows the conventional node-based storage of the same XML tree of **Fig. 1-(b)**, where a total of 23 nodes are stored. (The *no.*column is not stored. It is for reference. As for the *pathid* column, let us ignore it for the moment. It will be mentioned later.) Comparing **Fig. 5** with **Fig. 6**, we note that storing a subtree *t* can be regarded as equal to storing only its root node *r* except that path ID of *r* and *t* itself are stored as well. In XPath processing, *r* represents the whole subtree *t* without *t* being parsed on the fly.[2]

In this paper, we assume that the XML documents conform to some DTD (Document Type Definition). The only rule we enforce in fragmenting an XML tree into subtrees is that the *repeating or optional* elements, which are specified with +, *, or ? in the DTD, are to be the roots of different subtrees. We call this rule *basic XML fragmentation rule.* In **Fig. 3**, for example, b_1 and b_2 are the repeating elements because they have the same parent '*a*', and thus, the same path type */a/b*. Under the basic XML fragmentation rule, each of them is the root of a subtree as shown. As for the optional elements, if the XML nodes whose tag name is '*d*' at the path type */a/b/d* are optional, each of their instances, d_1 and d_3, is the root of a subtree as shown. The basic XML fragmentation rule plays a key role in subtree-based XPath processing as shall be shown in Section 4.

There are several related issues that are beyond the scope of this paper. For example, which nodes of the XPFS tree are marked with a double circle in coming up with an XPFS tree is important. So is updatability for dynamic XML documents. These and other issues are discussed in Section 7. In the remainder of this paper, we focus on subtree-based XPath processing where no subtree is parsed on the fly.

[2] As shall be shown in Section 4, as far as our subtree-based XPath processing is concerned, the element tag/attribute name and the level need not be stored because they are not refered to in XPath processing. The stored subtree *t* itself is not accessed either unless *t* contains the nodes of the *final* query result. Here, we include them all just to compare with the node-based storage.

no.	name	pathid	begin	end	level
(1)	a	#1	1	46	1
(2)	b	#2	2	21	2
(3)	@id	#3	3	6	3
(4)	dog	-	4	5	4
(5)	c	#4	7	10	3
(6)	sea	-	8	9	4
(7)	d	#5	11	14	3
(8)	run	-	12	13	4
(9)	e	#6	15	20	3
(10)	d	#7	16	19	4
(11)	boy	-	17	18	5
(12)	b	#2	22	41	2
(13)	@id	#3	23	26	3
(14)	cat	-	24	25	4
(15)	c	#4	27	30	3
(16)	sky	-	28	29	4
(17)	d	#5	31	34	3
(18)	run	-	32	33	4
(19)	e	#6	35	40	3
(20)	d	#7	36	39	4
(21)	toy	-	37	38	5
(22)	c	#8	42	45	2
(23)	boy	-	43	44	3

Fig. 6. Node-based XML Storage

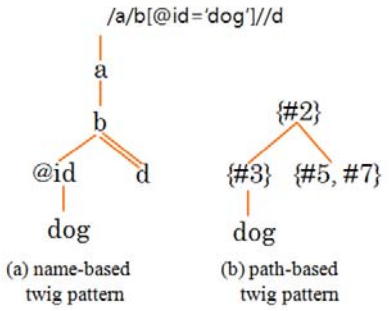

Fig. 7. Two Types of Twig Patterns

3. Node-based XPath Processing

In the literature, an XPath expression *e* is represented as a tree *T(e)* called *twig pattern* [1][2] [3]. **Fig. 7-(a)** shows an example for an XPath expression: */a/b[@id='dog']//d*. A *twig node* of *T(e)* is labeled with an element tag/attribute name or a text/attribute value. An edge between a pair of parent-child twig nodes is denoted either by a line or by a double line. The former represents the child axis (/) and the latter the self-or-descendant axis (//) in *e*. There is a special edge leading to the root of *T (e)*: a line or a double line. If *e* starts with a /, it is a line. If *e* starts with a //, it is a double line.

With the node-based XML storage, the well-known node-based XPath processing scheme proceeds as follows [2]:

1. For each twig node, a list of corresponding XML nodes is retrieved as an input to structural join.
2. These XML node lists are scanned while *node-based structural join* is conducted to produce a node set as the final query result.

We note that the above popular formulation of an XPath expression *e* as a twig pattern tree *T(e)* and the processing of *e* with *T(e)* is *name-based* in the following sense:

1. Each twig node is labeled with an element tag/attribute name or a text/attribute value. (The text/attribute value can also be regarded as a name.)
2. For each twig node *N*, the XML node list for structural join is retrieved through *name matching* by which we mean that the element/attribute/text nodes whose name/value is equal to the name/value of *N* are retrieved no matter which path they exist at.

Since we have the XPFS tree, for an XPath expression *e*, its *name-based twig pattern* $T_{name}(e)$ can be transformed into a *path-based twig pattern* $T_{path}(e)$ where each twig node is associated

either with *a set of path IDs* or with a text/attribute value. For example, the name-based twig pattern in **Fig. 7-(a)** is transformed into a path-based one as shown in **Fig. 7-(b)**. For an XPath expression *e*, $T_{path}(e)$ is obtained from $T_{name}(e)$ by mapping each *liner path* from the root to node *N* in $T_{name}(e)$ into the set of corresponding path IDs where *N* is either a branching node, a non-text leaf node, or the parent of a text/attribute value. For example, in $T_{name}(e)$ of **Fig. 7-(a)**, there are three afore-mentioned linear paths: */a/b*, */a/b/@id*, and */a/b//d* ('*b*' is a branching node, '*d*' is a non-text leaf node, and '*@id*' is the parent of an attribute value.) Looking up the XPFS tree of **Fig. 4**, the path */a/b* is mapped into {#2}, */a/b/@id* into {#3}, and */a/b//d* into {#5,#7}. (*/a/b//d* is resolved into */a/b/d* or */a/b/e/d*.) If the linear path does not contain a //, it corresponds to a unique path type and is mapped to a set of just one path ID. Otherwise, it may be resolved into multiple path types and is mapped to a set of one or more path IDs. As for a text/attribute value in $T_{name}(e)$, it remains intact.

To conduct XPath processing for a path-based twig pattern, a slight modification is needed to the node-based XML storage. For each element/attribute node *n* in the XML tree, the path ID of *n* is stored and indexed. Instead, the element tag/attribute name of *n* need not be stored. (In **Fig. 5** and in **Fig. 6**, the *pathid* column is stored instead of the name column.) With such a modified XML storage, the node-based XPath processing scheme for a path-based twig pattern proceeds in a similar way to that for a name-based one:

1. For each non-text twig node labeled with a set of path IDs, *S*, a list of corresponding XML nodes is retrieved as follows: For each path ID $ID_i \in S$, the set of all the XML nodes whose path ID is ID_i is retrieved. Union of these node sets forms the list. For this, an index on path ID is desired. For a text twig node, its list is retrieved through the value index.
2. These XML node lists are scanned for structural join to produce a node set as the final query result.

Such name-based to path-based transformation was materialized in XRel [8]. Since XRel is an XML storing and XPath processing system built on top of an RDBMS, all the path types in XML documents were enumerated and stored in a table called PATH with their path IDs. In translating an XPath expression into an SQL statement, such a transformation occurs. In the remainder of this paper, we deal only with the XPath processing for the path-based twig pattern. The equivalence between XPath processing for a path-based twig pattern and that for its original name-based counterpart is referred to [8].

4. Subtree-based XPath Processing

4.1 Overview

With a subtree-based XML storage, the conventional node-based XPath processing seems impossible without parsing the stored subtrees involved in the query. To come up with the query result, for each twig node, a list of XML nodes needs to be retrieved for structural join. Since the relevant XML nodes may reside *inside* the stored subtrees, it seems inevitable to retrieve the subtrees and parse them on the fly. For a node *n* inside a subtree *t*, actually there is no way to isolate and retrieve *n* without parsing *t*. Note that *n* was not stored or indexed separately on its own. Rather, what is stored is *t* (more specifically, the root node of *t*) and what is possibly indexed is just the root node of *t*. The values of the text/attribute nodes inside *t* cannot be indexed, either. (They simply cannot be accessed without parsing *t*.) Only the root node of *t* with its (begin:end) range and its path ID can be accessed. Then, how could an XPath

query be processed without the subtrees involved being parsed? In this section, we propose a solution.

In the node-based XPath processing, both the input and the output of structural join are XML *node sets*. Note that this is not the case for the subtree-based processing simply because the XML nodes may reside inside subtrees and could not be accessed. As such, the input and the output of structural join are *subtrees*, the unit of storage. More specifically, the differences between node-based and subtree-based XPath processing for a path-based twig pattern are as follows:

1. In the node-based processing, for each twig node, an XML node list (n_1, …, n_j) is retrieved where each n_i is represented by the (begin:end) range of n_i. In the subtree-based processing, for each twig node, a list of <path ID, subtree> pairs (<ID_1,t_1>, …, <ID_j,t_j>) is retrieved. A pair <ID_i,t_i> corresponds to n_i such that ID_i is the path ID of n_i and t_i is the subtree to which n_i belongs. Since for a subtree t, its root node is to represent t in XPath processing, what is really retrieved is a list of <path ID, the root of a subtree> pairs (<ID_1,r_1>, …, <ID_j,r_j>) where r_i is the root node of t_i and represented by its (begin:end) range. As mentioned in Section 2, the (begin:end) range of r_i is stored outside t_i. The subtrees themselves (t_i's) need not be retrieved.[3]
2. The final query result in the node-based processing is an XML node set {n_1, …, n_k} as defined in the XPath specification [9]. That should be the same in the subtree-based processing but here subtrees, not individual nodes, are dealt with. Thus, a set of <path ID, subtree> pairs, {<ID_1,t_1>, …, <ID_k,t_k>} are returned first as a *pre-result*. To be exact, a set of <path ID, the root of a subtree> pairs (<ID_1,r_1>, …, <ID_k,r_k>) is returned where r_i is the root of t_i. For the final result, each t_i is retrieved from the "begin" value of r_i, and each n_i whose path ID is ID_i is retrieved from t_i. Only in this stage, parsing of t_i is needed.

Our proposed subtree-based XPath processing scheme proceeds as follows: Given an XPath expression e,

STEP 1: its name-based twig pattern $T_{name}(e)$ is transformed into a path-based one $T_{path}(e)$.
STEP 2: For each twig node of $T_{path}(e)$, the input subtree list is retrieved from the subtree-based XML storage.
STEP 3: These subtree lists are scanned while *subtree-based structural join* is conducted to produce a set of subtrees as a *pre-result*.
STEP 4: The final result node set is obtained from the pre-result subtrees.

The above steps 1 and 4 are already described. The steps 2 and 3 are described below.

4.2 Input of Subtree-based XPath Processing

In STEP 2, to retrieve the input subtree list from the subtree-based XML storage, for each non-text twig node labeled with a set of path IDs S = { ID_1, …, ID_j }, S is first converted to a new set S_r = { <ID_1, $R(ID_1)$>, …., <ID_j, $R(ID_j)$> } where the function R called *root function* is defined as follows:

Definition 4.1 [Root Function]
For a node N in an XPFS tree T, let x be the path ID of the path type from the root of T to N. Let N_r be the root node (i.e., one in double circle) of the subtree to which N belongs in T. Then,

[3] In the remainder of this paper, we use the followings interchangeably: a list of subtrees, a list of <path ID, subtree> pairs, and a list of <path ID, the root of a subtree> pairs.

$R(x) = y$ where y is the path ID of the path type from the root of T to N_r. □

Example 4.1
In **Fig. 4**, R(#3)=#2 because the node '*@id*' (path ID=#3) is a child of '*b*' (path ID=#2) in double circle. R(#7)=#6 because the node '*d*' at */a/b/e/d* (path ID=#7) is a child of '*e*' (path ID=#6) in double circle. Meanwhle, R(#5)=#5 and R(#2)=#2 because for the double circle node, the root function returns the path ID of itself. □

The conversion of S to S_r is to retrieve the input subtree list for structural join instead of the XML node list, which cannot be retrieved without parsing the subtrees. For each $<ID_i, R(ID_i)>$ in S_r, the set of all the subtrees the path ID of whose root is $R(ID_i)$ is retrieved. These subtrees are those that contain the *target* XML nodes whose path ID is ID_i. Union of these subtree sets forms the input list for the twig node labeled with S_r.

<pathid, value>	node pointers (no. column in Fig. 6)
<#3, dog>	{ (4) }
<#4, sea>	{ (6) }
<#5, run>	{ (8), (18) }
<#7, boy>	{ (11) }
<#3, cat>	{ (14) }
<#4, sky>	{ (16) }
<#7, toy>	{ (21) }
<#8, boy>	{ (23) }

(a) index entries for node-based storage

<pathid, value>	subtree pointers (no. column in Fig. 5)
<#3, dog>	{ (2) }
<#4, sea>	{ (2) }
<#5, run>	{ (3), (6) }
<#7, boy>	{ (4) }
<#3, cat>	{ (5) }
<#4, sky>	{ (5) }
<#7, toy>	{ (7) }
<#8, boy>	{ (1) }

(b) index entries for subtree-based storage

Fig. 8. Composite Value Index

As for the text twig node with a value v, all the text nodes whose value is v need to be retrieved for structural join. They may reside inside the stored subtrees. We handle this problem through a *modified value index* as follows: An index entry of the conventional value index for a node-based XML storage is of the form (v,P) where v is a value and P is a set of pointers to the text nodes whose value is v. Such typical indexing is for the name-based twig pattern, and the pointers in P are supposed to point to all the text nodes whose value is v no matter which path they exist at. We first modify it so that it well fits the path-based twig pattern. A *composite* value index whose entry is of the form ($<I, v>$, P) is used where

1. I is the path ID of the path type from the root of the XML document to the parent node of the text node N_t whose value is v.
2. P is a set of pointers to the instances of such N_t.

With a subtree-based storage, however, a text node inside a subtree cannot be pointed to directly. Thus, P needs further modification: P is a set of pointers { p_1, …, p_q } where each p_k points to the root of the subtree to which the instances of N_t belongs.

Example 4.2
Fig. 8-(a) shows the index entries for the XML tree of **Fig. 1-(b)** with the node-based XML storage of **Fig. 6**. The pointers are represented by the reference number in *no*.column in **Fig. 6**. **Fig. 8-(b)** shows the index entries for the XML tree of **Fig. 1-(b)** with the subtree-based XML storage of **Fig. 5**. The pointers are represented by the reference number in *no*.column in **Fig. 5**.

For a text twig node N_t labeled with a value v whose parent twig node is N_p labeled with a set of path IDs S = { ID_1, …, ID_j } (or with its converted set S_r = { <ID_1, $R(ID_1)$>, …., <ID_j, $R(ID_j)$> }), the input subtree list for N_t is the union of <ID_i, $ST(p_k)$>, i = 1, …, j and k= 1, …, q where $p_k \in P_i$ such that P_i is the set of pointers of the value index entry (<ID_i, v>, P_i) and $ST(p_k)$ denotes the subtree pointed to by p_k.

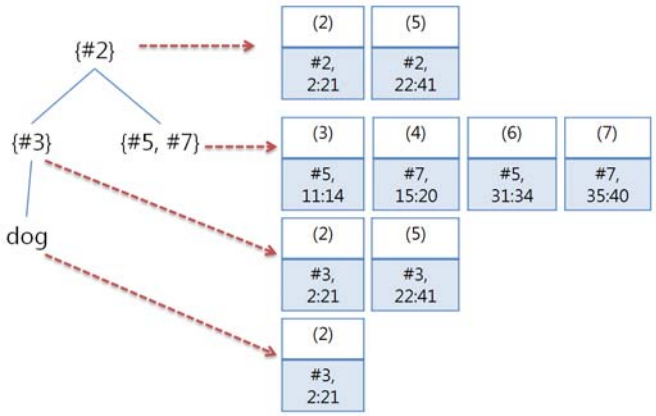

Fig. 9. Input Subtree Lists

Example 4.3
Fig. 9 shows the input subtree lists for the path-based twig pattern of **Fig. 7-(b)** retrieved from the subtree-based XML storage of **Fig. 5**. For the twig nodes labeled with {#2}, {#3}, and {#5,#7}, two, two, and four subtrees are retrieved, respectively. Each box corresponds to a subtree t that contains the target XML node n, denoted by <(i) j, k:l> where (i) is the reference number of t (in *no*.column of **Fig. 5**), j is the path ID of n, and k:l is the (begin:end) range of the root of t. The first subtree denoted by <(2) #3, 2:21>, for the twig node labeled with {#3}, for example, corresponds to the subtree rooted at 'b_1' in **Fig. 3**, and its target XML node is '$@id_1$'.

Now STEP 2 is completed, and every twig node is provided with its input subtree list. We are ready for STEP 3 described in the next subsection.

4.3 Subtree-based Structural Join

A structural join between two lists of XML nodes is well-known [1][2][4]. The basis of node-based structural join is that given two nodes n_1 and n_2 with their (begin:end) ranges, their structural relationship (or *containment relationship*) can be determined by comparing their ranges. In the case of subtree-based XML storage, n_1 and n_2 may reside inside some subtrees and their ranges are not given. It could also be that n_1 and n_2 may reside in the same subtree. How could the structural relationship between n_1 and n_2 be determined without parsing the subtrees? In this subsection, we show that such subtree-based structural join can be correctly conducted.

A path type in an XML document *D* can be defined as a sequence of element tag/attribute names. It corresponds to a path in the XPFS tree of *D*. For two path types, one can be a *prefix* of the other. For example, in the XPFS tree of **Fig. 4**, the path type */a/b* is a prefix of the path types */a/b/c*, */a/b/e/d*, and so on. A path type is also a prefix of itself. For an XML node *n* in an XML document *D*, let *P(n)* denote the path type of the path from the root of *D* to *n*. For example, in the XML tree of **Fig. 2**, $P(b_1)$ = */a/b*, $P(c_1)$ = */a/b/c*, and $P(c_3)$ = */a/c*. (We note that $P(b_1)$ is a prefix of $P(c_1)$ but $P(c_3)$ is not a prefix of $P(c_1)$.) For an XML node *n*, let *range(n)* denote the (begin:end) range of *n*, and for a stored subtree *t* whose root node is *r*, let *range(t)* denote the *range(r)*. Then, with a subtree-based XML storage of an XML document *D* derived by the XPFS tree with the basic XML fragmentation rule as described in Section 2, we have the following Theorem.

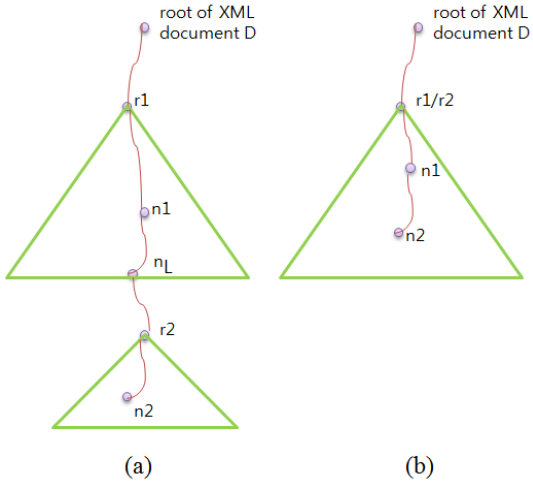

Fig. 10. Two Cases in Theorem 4.1

Theorem 4.1

For two XML nodes n_1 and n_2 which respectively belong to subtrees t_1 and t_2 in an XML document *D* stored in a subtree-based storage derived by the XPFS tree of *D*, n_1 is an ancestor of n_2 *iff* the following two conditions hold:

1. *range*(t_1) properly contains *or* is equal to *range*(t_2).
2. $P(n_1)$ is a prefix of $P(n_2)$ □

Proof

(if) Suppose the above two conditions hold. Then, we show that n_1 is an ancestor of n_2. Let r_1 and r_2 be the root node of t_1 and t_2, respectively (cf. **Fig. 10**). In case that range(t_1) properly contains range(t_2), there is a leaf node n_L in t_1 which is the parent of r_2. In case that $range(t_1)$ is equal to $range(t_2)$, t_1 and t_2 are the same subtree. That is, r_1 and r_2 are the same, and n_1 and n_2 are in the same subtree. Due to the basic XML fragmentation rule, for a path type *P*, there could be at most one instance of an XML node in a subtree whose path type is *P*. As such, there is not a node *m* in t_1 such that $P(m)=P(n_1)$. Similarly, there is not a node *m* in t_2 such that $P(n)=P(n_2)$. In other words, n_1 is the unique node in t_1 whose path type is $P(n_1)$, and n_2 is the unique node in t_2 whose path type is $P(n_2)$. As such, n_1 and n_2 must be on the same path *p* because $P(n_1)$ is a prefix of $P(n_2)$ where *p* is as follows: In case that $range(t_1)$ properly contains $range(t_2)$, *p* is the path: the root of D-r_1-n_1-n_L-r_2-n_2 (cf. **Fig. 10-(a)**). In case that $range(t_1)$ is

equal to *range(t_2)*, *p* is the path: the root of *D*- $r_1(r_2)$-n_1-n_2 (cf. **Fig. 10-(b)**). In all, n_1 is an ancestor of n_2.

(only if) Suppose n_1 is an ancestor of n_2. Then, we show that the above two conditions hold. t_1 and t_2 are either the same subtree or different ones. In case that t_1 and t_2 are the same, *range(t_1)* is equal to *range(t_2)*. In case that t_1 and t_2 are different, let r_1 and r_2 be the root node of t_1 and t_2, respectively. Since n_1 is an ancestor of n_2, they are on the same path: the root of D-r_1-n_1-n_L-r_2-n_2 where n_L is a leaf node in t_1 which is the parent of r_2 (cf. **Fig. 10-(a)**). Since *range(t_1)* properly contains or is equal to *range(n_L)*, *range(n_L)* properly contains *range(r_2)*, and *range(r_2)* is *range(t_2)*, *range(t_1)* properly contains *range(t_2)*. Therefore, the first condition holds. As for the second condition, *P(n_1)* must be a prefix of *P(n_2)* because n_1 is an ancestor of n_2. □

Theorem 4.1 is the basis of the subtree-based structural join. As described in the previous subsection, for each twig node in the path-based twig pattern for an XPath expression, the list of <path ID, subtree> pairs is retrieved for structural join. Each pair <ID_i, t_i> corresponds to an XML node n_i that belongs to the subtree t_i. The path type of n_i can be obtained by looking up the XPFS tree with its path ID ID_i. The (begin:end) range of t_i is *a priori* given because what represents t_i in the pair < ID_i, t_i > is the (begin:end) range of the root node of t_i. As such, the structural relationship between two XML nodes n_1 and n_2 in an XML document stored in subtrees can be determined given two pairs <ID_1, t_1> and <ID_2, t_2> which respectively correspond to n_1 and n_2 without parsing t_1 or t_2.

Given a path-based twig pattern for an XPath expression *e*, each of its twig nodes is first provided with the input list of <path ID, subtree> pairs. Then, a series of subtree-based structural joins between two twig nodes N_p and N_c such that N_p is the parent of N_c is conducted. That obtains the pre-result of *e*, which is a set of <path ID, subtree> pairs. Each pair corresponds to an XML node in the node set that would be produced as the final result of *e*.

4.4 Correctness of Subtree-based XPath Processing

The correctness of the processing of an XPath expression against a subtree-based XML storage can be defined as follows:

Definition 4.2
In an XML document *D*, a <path ID, subtree> pair <*I*, *t*> where *t* is a subtree of *D* is said to *uniquely cover* an XML node *n* of *D* iff

1. *n* exists in *t* and
2. *n* is the only XML node in *t* whose path ID is *I*. □

Definition 4.3
In an XML document *D*, let S_T and S_n be a set of <path ID, subtree> pairs and a set of XML nodes, respectively. S_T is said to *uniquely cover* S_n iff the elements of S_T are in one-to-one correspondence with those of S_n such that for each corresponding pair (<ID_i, t_i>, n_i) where <ID_i, t_i> ∈ S_T and n_i ∈ S_n, <ID_i, t_i> uniquely covers n_i. □

Definition 4.4
For an XML document *D*, its XPFS tree, and an XPath expression *e*, let R_n be the XML node set which is the final result of *e* retrieved against the node-based storage of *D* and let PR_T be the set of <path ID, subtree> pairs which is the pre-result of *e* retrieved against a subtree-based storage of *D* derived by the XPFS tree. PR_T is *correct* iff PR_T uniquely covers R_n.

The correctness of PR_T implies that if PR_T is correct, it is guaranteed to correctly get R_n out of PR_T in STEP 4 of the subtree-based XPath processing.

Example 4.4
For XPath expression *e*, */a/b[@id='dog']//d*, the result of *e* against the XML tree of **Fig. 2** obtained by the conventional node-based XPath processing is the node set $R_n = \{d_1, d_2\}$. The pre-result of *e* against the subtree-based XML storage shown in **Fig. 3** obtained by the subtree-based XPath processing is the set PR_T = {<path ID of d_1, subtree rooted at d_1>, <path ID of d_2, subtree rooted at e_1>}. We note that PR_T and R_n are in one-to-one correspondence such that <path ID of d_1, subtree rooted at d_1> uniquely covers d_1 and that <path ID of d_2, subtree rooted at e_1> uniquely covers d_2. Thus, PR_T uniquely covers R_n, and PR_T is correct. □

Theorem 4.2
For an XML document *D*, its XPFS tree, and an XPath expression *e*, the pre-result of *e* for the path-based twig pattern of *e* against a subtree-based XML storage of *D* derived by the XPFS tree, obtained by STEP 2 and then by STEP 3 of the previous subsections without parsing the subtrees involved, is *correct*. □

Proof
Let $R_n(e)$ and $PR_T(e)$ be the final result of *e* obtained against the node-based storage of *D* and the pre-result of *e* obtained against the subtree-based storage of *D*. Let *T(e)* be the path-based twig pattern of *e*. For each twig node of *T(e)*, let L_n and L_T be the input XML node list retrieved from the node-based storage of *D* and the input list of <path ID, subtree> pairs retrieved from the subtree-based storage of *D*, respectively. Because of the basic XML fragmentation rule described in Section 2 and the tasks of STEP 2, we see that L_T uniquely covers L_n. As such, we see that due to Theorem 4.1, $PR_T(e)$ obtained in STEP 3 by a series of structural joins uniquely covers $R_n(e)$. Thus, by Definition 4.4, $PR_T(e)$ is correct. □

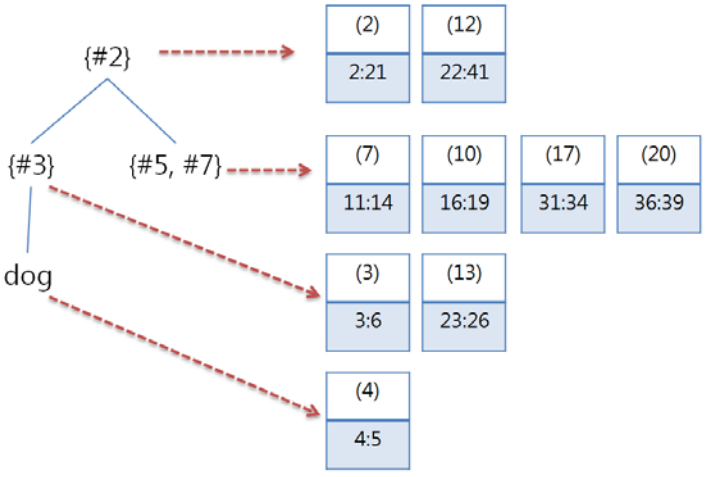

Fig. 11. Input XML Node Lists

Example 4.5
Let us consider an XPath expression *e*, */a/b[@id='dog']//d*. **Fig. 11** shows the path-based twig pattern for *e* with its input *XML node* lists retrieved from the node-based XML storage of **Fig. 6**. (Note that **Fig. 11** is the node-based version of **Fig. 9**. The latter shows the same twig pattern

with its input lists of <path ID, subtree> pairs retrieved from the subtree-based XML storage of **Fig. 5**). For the twig nodes labeled with {#2}, {#3}, and {#5,#7}, two, two, and four XML nodes are retrieved, respectively. Each box corresponds to an XML node *n* denoted by <(*i*) *j*:*k*> where (*i*) is the reference number of *n* (in *no.*column of **Fig. 6**), and *j*:*k* is the (begin:end) range of *n*.

Let us compute $R_n(e)$ in **Fig. 11** by a series of structural joins. Joining two twig nodes {#3} and 'dog' (i.e., joining {(3), (13)} vs. {(4)}) produces the XML node (3) for the twig node {#3} as an intermediate result. Joining twig nodes {#2} and {#3} produces the XML node (2) for the twig node {#2}. Finally, joining twig nodes {#2} and {#5,#7} produces the final result node set $R_n(e)$={(7), (10)}, which corresponds to d_1 and d_2 in **Fig. 2**.

Now let us compute $PR_T(e)$ in **Fig. 9**. The whole process is the same as above except that the path IDs are checked as well to see if one path type is a prefix of the other. Joining two twig nodes {#3} and 'dog' (i.e., joining {<#3, subtree (2)>, <#3, subtree (5)>} vs. {<#3, subtree (2)>}) produces the <#3, subtree (2)> for the twig node {#3} as an intermediate result. All the path IDs here are #3. Since they are the same, the prefix condition is satisfied. Joining twig nodes {#2} and {#3} produces the <#2, subtree (2)> for the twig node {#2}. Here, the path IDs involved are #2 and #3. Looking up the XPFS tree of **Fig. 4**, the prefix condition also holds. Finally, joining twig nodes {#2} and {#5,#7} produces the pre-result subtree set $PR_T(e)$={<#5, subtree (3)>, <#7, subtree (4)>}, which corresponds to {<path ID of d_1, subtree rooted at d_1>, <path ID of d_2, subtree rooted at e_1>} in **Fig. 3**. As examined in Example 3.4, $PR_T(e)$ uniquely covers $R_n(e)$. Thus, $PR_T(e)$ is correct. □

5. Implementation and Experiments

5.1 Implementation

We have implemented in Java two XML storing and XPath processing systems as described in previous sections. One is with the node-based XML storage and XPath processing (*System/N*), and the other is with the subtree-based XML storage and XPath processing (*System/T*). For ease of implementation, an RDBMS, Oracle 11g, was employed for storing and indexing in both systems. In System/N, each node is stored as a record of a table called *NODE* whereas in System/T, each subtree is stored as as a record of a table called *SUBTREE*.

The *auction.xml* generated in XMark [10] was used as the XML document. XPath expressions that appear in the XQuery FLWOR expressions in XMark were used to confirm the correctness of the proposed subtree-based XPath processing. They are listed in **Table 1**. The XPath processing time in System/T and System/N was measured and compared. Also the space requirement in System/T and System/N was measured and compared. The implementation and experiments were carried out on a Windows XP server with Intel Core 2 Duo 6600 (2.40Ghz) CPU and 2 GB memory.

Table 1. XPath Expressions

Query	XPath Expressions
Q1	/site/people/person[homepage]/name
Q2	/site/closed_auctions/closed_auction/price
Q3	/site//item[description]/name
Q4	/site/regions/europe/item[@id]/name
Q5	/site/people/person[profile/interest/@category]/address/city

5.2 XPath Processing Time

For all the XPath expressions in **Table 1**, the correctness of the proposed subtree-based XPath processing was confirmed with auction.xml of size 1.16 MB. Though the goal of the technique proposed in this paper is *not* for improving query performance, we checked query performance of System/T in comparison with that of System/N by carrying out two experiments. The two differ only in System/T. In the first experiment, for every query, the final result XML nodes are set to be the descendants of the root of their subtrees. In the second experiment, for every query, the final result XML nodes are set to be the *root* of their subtrees. The difference is the necessity of the parsing of the subtrees in the final stage of subtree-based XPath processing (i.e., STEP 4) in System/T: parsing is needed in the first experiment, whereas it is *not* in the second. **Table 2** shows the time in System/N (T_N) and in Syetem/T (T_T) out of the first experiment and **Table 3** out of the second. Each of the measured time is the average of 100 runs. As shown in **Table 2**, System/T took more time than System/N because of the parsing overhead. However, as shown in **Table 3**, System/T outperforms System/N.

Table 2. Query Processing Time (ms)
(result node: *inside* the subtree)

Query	T_N	T_T	T_T / T_N
Q1	65.90	97.61	1.48
Q2	46.56	80.08	1.72
Q3	154.71	261.92	1.69
Q4	48.13	88.84	1.85
Q5	51.52	59.04	1.15

Table 3. Query Processing Time (ms)
(result node: *root* of the subtree)

Query	T_N	T_T	T_T / T_N
Q1	66.37	19.45	0.29
Q2	49.68	15.17	0.31
Q3	154.62	19.96	0.13
Q4	48.26	16.57	0.34
Q5	51.52	22.22	0.43

There are mainly two reasons for the superior query performance of System/T in the second experiment. First, as shall be described shortly, the size of the table NODE is much larger than that of SUBTREE. As such, the time to retrieve the input node lists for the twig nodes in System/N took much longer than the time to retrieve the input subtree lists for the twig nodes in System/T. Secondly, the extra work done in the subtree-based XPath processing compared with the node-based counterpart is not burdensome, calling the root function in retrieving the input subtree lists and checking if one path type is a prefix of another in structural join.

Table 4. Space Overhead for XPFS Tree

Scaling factor for auction.xml	S_X Size of auction.xml (KB)	S_T Size of XPFS Tree (KB)	$(S_T / S_X) * 100$ (%)
0.01	1,155	49.2	4.26
0.02	2,330	51.6	2.21
0.04	4,727	56.4	1.19
0.06	6,984	57.9	0.83
0.08	9,371	58.6	0.63
0.1	11,597	59.1	0.51
0.2	23,365	60.5	0.26
0.4	46,395	60.5	0.13
0.6	69,892	60.5	0.09
0.8	92,975	60.5	0.07
1	115,775	60.5	0.05

5.3 Space Requirement

The subtree-based XML storage is much more compact than the node-based counterpart. In the first experiment mentioned above, for auction.xml of size 1.16 MB, there are 21,051 records stored in the table NODE, and 4,791 records in the table SUBTREE (Note that when the XML tree of **Fig. 1-(b)** is stored, there are 23 nodes in the node-based XML storage in **Fig. 6** while there are just 7 subtrees in the subtree-based counterpart in **Fig. 5**. The space occupied by NODE and SUBTREE in the table space of Oracle is roughly 0.98 MB and 0.51 MB, respectively. We see that a significant space saving is achieved with the subtree-based XML storage.

As for the XPFS tree, it is mandatory in System/T. In System/N, it is optional. (If the path-based XML storage and XPath processing is employed in System/N as described in this paper, the XPFS tree is mandatory even in System/N.) We generated 11 auction.xml documents of various sizes in XMark. For each of them, we measured the size of the XPFS tree implemented as an *n*-ary tree stored as a file. **Table 4** shows that compared with the size of the auction.xml source document, the space required for the XPFS tree is negligible.

6. Related Work

Much work has been conducted on storing XML data. Several techniques that employ an RDBMS were proposed first [6][8][11][12]. Since the range numbering schemes were proposed in [1][4], the techniques of XML storing have been centered around XML node labeling. As such, the XML node labeling schemes had received much attention, and the schemes ORDPATHS [13], QED [14], and so on were developed. As for XPath processing, the structural join based on XML node labeling has been established as a core technique [1] [2][4]. As we pointed out in this paper, however, the current state-of-the-art technique of XML storing and XPath processing is node-based.

In STORED [6], the irregular parts of the XML data was treated as overflow when the XML data is mapped to relational schema. Such overflow data could be stored as a subtree. In [7], a hybrid design of object-relational schema to store XML data was investigated. The irregular parts and/or infrequently accessed parts of the XML data are stored as subtrees while all the XML nodes in the rest are separately stored. In these work, however, the capability of dealing with the stored subtrees without parsing in query processing was not provided.

In Natix [15] and SystemRX [16], which provide the native XML data management functions, the nodes close to each other (e.g., parent-child) or the nodes in a subtree are put to the same page (in the slotted page structure). In Natix, such subtrees are identified by fragmenting the XML documents with the use of virtual nodes as connector between subtrees. However, those subtrees are stored in some parsed form so that the nodes in them can be accessed. Strictly speaking, they use node-based storage with node clustering capability. In our work, the storage unit is a subtree, and the subtrees are not parsed during XPath processing, and thus, it can be stored as just plain text (byte stream).

The techniques of XPath processing over a stream of XML subtrees fragmented out of a source XML document in a resource-limited client devices were proposed [17][18]. In those techniques, however, each received subtree is parsed in query processing. In [5], active XML processing was investigated where some portions of XML data are dynamically handled as XML subtrees especially in the web service applications. However, the capability of dealing with the subtrees without parsing in query processing was not provided.

Techniques of exploiting XML path schema had received little attention. XRel [8] and XParent [19] are early systems with such a feature. Recently, in [20], a native XML data

management techniques that exploit the path information called path synopsis with which a very space-efficient XML storage can be achieved were proposed. These work, however, does not feature subtree-based XML storage and XPath processing with the path information. The XPFS tree in our work plays the similar role as the well-known DataGuide [21] or the path synopsis [20] which summarizes the path structure of a semistructured or XML data except that the XPFS tree 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 [22].

7. Conclusions and Future Work

In this paper, we proposed the subtree-based XML storage and XPath processing as a generalization of the state-of-the-art node-based XML storage and XPath processing. The main contributions of this paper are:

1. We proposed the technique of subtree-based XML storing and XPath processing whereby the parsing of the subtrees involved in the query is *not* required.
2. Through implementation and experiments, we showed that our proposed solution is a feasible generalization of the node-based solution both in time and in space.

As shown in the query performance experiments (**Table 3**), the frequently requested XML nodes (i.e., *hot* nodes) are desired to be the roots of the subtrees while the *cold* ones are placed inside the subtrees. As a future work, we are now considering a problem of *optimization* of fragmenting an XML document modeled as a tree into subtrees given a set of XPath expressions and their frequencies.

Since the contents of the stored subtrees except for their roots need not be accessed during query processing, another future work we are interested in is the application of the proposed subtree-based XML storage and XPath processing technique to *secure* XML data processing.

The primary focus in this paper was on the correctness of XPath processing without parsing the stored subtrees involved. However, updatability is also an important issue. The proposed technique should not undermine updatability. As for dynamic XML documents, they can also be supported with the proposed technique. A basic assumption in this paper is that the XML documents conform to some DTD. Also the basic XML fragmentation rule is enforced. It states that the repeating or optional elements specified with +, *, or ? in the DTD are to be the roots of different subtrees. As such, each of the nodes that reside inside a subtree (excluding the root) is the node that appear exactly once (at least once as well as at most once) as a subelement or an attribute of its parent. That is, their cardinality (i.e., the number of occurrences) is 1. Thus, the internal structure of any subtree instance is supposed to remain static, conforming to the schema. In other words, every non-root node in a subtree appears at least once (which means it cannot be deleted) and at most once (which means it is already inserted). Meanwhile, the repeating or optional nodes are the roots of some subtrees, and the entire subtree can be inserted or deleted as a static unit. To support the updates that would change the internal structure of a subtree, schema evolution needs to be considered. Although the types of allowed updates with our technique are restricted to some extent because of the assumption of having DTD and the basic XML fragmentation rule, they cover practical types of updates while conformance to the schema is preserved. Thus, our technique can be applied to a wide range of dynamic environment as well. An important issue requiring further investigation is the relationship among update types, schema evolution, and fragmentation.

Other issues that need to be addressed include transaction management in multi-user environments. As far as locking is concerned, for example, the new types of muti-user

concurrent operations include the followings: (1) Two users are to read different parts of the same subtree. (2) One is trying to read a node inside a subtree while the other intends to delete the same subtree, and so on. A locking protocol similar to the multiple granularity intent locking (acquiring a weaker lock on the subtree first, and then a stronger one on the nodes inside the subtree) could be devised for concurrency control. In the former case, both users are supposed to be in the shared lock mode, and not in conflict. Both users are required to acquire the shared lock on the subtree. In the latter, they are in conflict, which is detected as follows: With the node-based storage, a node n is uniquely identified with L_n (the label of n) whereas with our subtree-based storage, the same node n which might be residing inside a subtree t is uniquely identified with the pair $<L_r, I_n>$ where L_r is the label of the root of t, and I_n is the path ID of n. As such, an afore-mentioned intent locking-like protocol can reveal the conflict. Because to obtain a lock on n, the lock on t needs to be obtained first, both users are supposed to try to acquire an exclusive lock on t (i.e., the lock on name L_r).

References

[1] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman, “On Supporting Containment Queries in Relational Database Management Systems,” in *Proc. of ACM SIGMOD International Conf. on Management of Data*, pp. 425-436, 2001.

[2] S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, D. Srivastava, Y. Wu, “Structural Joins: A Primitive for Efficient XML Query Pattern Matching,” in *Proc. of 18th International Conf. on Data Engineering*, pp. 141-152, 2002.

[3] N. Bruno, N. Koudas, and D. Srivastava, “Holistic Twig Joins: Optimal XML Pattern Matching,” in *Proc. of ACM SIGMOD International Conf. on Management of Data*, pp. 310-321, 2002.

[4] Q. Li and B. Moon, “Indexing and Querying XML Data for Regular Path Expressions,” in *Proc. of 27th International Conf. on Very Large Data Bases*, pp. 361-370, 2001.

[5] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda, "Lazy Evaluation for Active XML," in *Proc. of ACM SIGMOD International Conf. on Management of Data*, pp. 227-238, 2004.

[6] A. Deutsch, M. Fernandez, D. Suciu, “Storing Semistructured Data with STORED,” in *Proc. of ACM SIGMOD International Conf. on Management of Data*, pp. 431-442, 1999.

[7] M. Klettke and H. Meyer, “XML and Object-Relational Database Systems – Enhancing Structural Mappings based on Statistics,” in *Proc. of 3rd International Workshop on Web and Databases*, pp. 63-68, 2000.

[8] Yoshikawa, T. Amagasa, T. Shimura, S. Uemura, “XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases,” *ACM Trans. on Internet Technology*, vol. 1, no. 1, pp. 110-141, 2001.

[9] J. Clark and S. DeRose, editors, XML Path Language (XPath) version 1.0, *W3C Recommendation*, Nov. 1999, http://www.w3.org/TR/xpath.

[10] A. Schmidt, F. Wass, M. Kersten, M. Carey, I. Manolescu, and R. Busse, “XMark: A Benchmark for XML Data Management,” in *Proc. of 28th International Conf. on Very Large Data Bases*, pp. 974-985, 2002.

[11] D. Florescu and D. Kossmann, “Storing and Querying XML Data Using an RDBMS,” *IEEE Data Engineering Bulletin*, vol. 22, no. 3, pp. 27-34, 1999.

[12] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. DeWitt, and J. Naughton, “Relational Databases for Querying XML Documents: Limitations and Opportunities,” in *Proc. of 25th International Conf. on Very Large Data Bases*, pp. 302-314, 1999.

[13] P. O’Neil, E. O’Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury, “ORDPATHs: Insert-Friendly

XML Node Labels," in *Proc. of ACM SIGMOD International Conf. on Management of Data*, pp. 903-908, 2004.

[14] C. Li and T. Ling, "QED: A Novel Quaternary Encoding to Completely Avoid Relabeling in XML Updates," in *Proc. of International Conf. on Information and Knowledge Management*, pp. 501-508, 2005.

[15] T. Fiebig, S. Helmer, C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann, "Anatomy of a Native XML Base Management System," *VLDB Journal*, vol. 11, no. 4, pp. 292-314, 2002.

[16] K. Beyer, R. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, R. Lyle, F. Ozcan, H. Pirahesh, N. Seemann, T. Truong, B. Van der Linden, B. Vickery, and C. Zhang, "System RX: One Part Relational, One Part XML," in *Proc. of ACM SIGMOD International Conf. on Management of Data*, pp. 374-358, 2005.

[17] S. Bose and L. Fegaras, "XFrag: A Query Processing Framework for Fragmented XML Data," in *Proc. of 8th International Workshop on Web and Databases*, pp. 97-102, 2005.

[18] H. Huo, G. Wang, X. Hui, R. Zhou, B. Ning, and C. Xiao, "Effiecient Query Processing for Streamed XML Fragments," in *Proc. of 11th International Conf. on Database Systems for Advanced Applications*, pp. 468-482, 2006.

[19] H. Jiang, H. Lu, W. Wang, and J. Yu, "XParent: An Efficient RDBMS-based XML Database System," in *Proc. of 18th International Conf. on Data Engineering*, pp. 335-336, 2002.

[20] C. Mathis, T. Härder, and K. Schmidt, "Storing and Indexing XML Documents Upside Down," *Computer Science - Research and Development*, vol. 24, no. 1-2, pp. 51-68, 2009.

[21] R. Goldman and J. Widom, "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases," in *Proc. of 23rd International Conf. on Very Large Data Bases*, pp. 436-445, 1997.

[22] P. Grosso and D. Veillard, editors, XML Fragment Interchange (XFI), *W3C Candidate Recommendation*, Feb. 2001, http://www.w3.org/TR/xml-fragment.

Ki Hoon Shin received the B.E. degree in Computer Science and Engineering from Chung-Ang University, Seoul, Korea in 2009. He is currently a M.S. student in Computer Science and Engineering at Chung-Ang University. His research interests include XML query processing and sensor network database.

Hyunchul Kang received the B.E. degree in Computer Engineering from Seoul National University, Seoul, Korea in 1983, and received the M.S. and Ph.D. degrees in Computer Science from the University of Maryland, College Park, U.S.A. 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, mobile data management, and sensor network database.