KSII TRANSACTIONS ON INTERNET AND INFORMATION SYSTEMS VOL. 3, NO. 5, October 2009

A Minimum Sequence Matching Scheme for Efficient XPath Processing

Dong-min Seo[1], Myung-Ho Yeo[2], Myoung-Ho Kim[1] and Jae-Soo Yoo[2]
[1] Department of Computer Science, Korean Advanced Institute of Science and Technology, Daejen, Korea
[e-mail: {dmseo, mykim}@dbserver.kaist.ac.kr]
[2] Department of Computer and Communication Engineering, Chungbuk National University, Cheongju, Korea
[e-mail: {mhyeo, yjs}@chungbuk.ac.kr]
*Corresponding author: Jae-Soo Yoo

Received August 15, 2009; revised September 22, 2009; accepted September 28, 2009; published October 30, 2009

Abstract

Index structures that are based on sequence matching for XPath processing such as ViST, PRIX and LCS-TRIM have recently been proposed to reduce the search time of XML documents. However, ViST can cause a lot of unnecessary computation and I/O when processing structural joint queries because its numbering scheme is not optimized. PRIX and LCS-TRIM require much processing time for matching XML data trees and queries. In this paper, we propose a novel index structure that solves the problems of ViST and improves the performance of PRIX and LCS-TRIM. Our index structure provides the minimum sequence matching scheme to efficiently process structural queries. Finally, to verify the superiority of the proposed index structure with the minimum sequence matching scheme, we compare our index structure with ViST, PRIX and LCS-TRIM in terms of query processing of a single path or of a branching path including wild-cards (‘*’ and ‘//’).

Keywords: XML, XPath, query processing, sequence matching, index structure

A preliminary version of this paper appeared in ICCS 2007, LNCS 4489, May 27-30, Beijing, China. This version includes a concrete analysis and supporting experimental results with recent sequence schemes. This work was supported by the Ministry of Education, Science and Technology Grant funded by the Korean Government (The Regional Research Universities Program/Chungbuk BIT Research-Oriented University Consortium) and by the Brain Korea 21 Project, the School of Information Technology, KAIST in 2009.

DOI: 10.3837/tiis.2009.05.005

1. Introduction

As XML is gaining complete success in being adopted as universal data representation and exchange format. However, particularly on the World Wide Web, the problem of querying XML documents poses interesting challenges to database researchers. Recently, there have been many studies on the structural join methods to efficiently process the XML queries involving ancestor-descendant relationships [1][2][3]. The structural join methods can efficiently answer simple queries. However, twig queries involving branching structures usually have to be disassembled into multiple sub-queries. The results of these sub-queries are then combined by expensive join operations to produce final answers. To improve the existing structural join methods, Wang *et al*. have proposed a new method, called ViST, that transforms the XML data trees and queries into structure-encoded sequences [4]. ViST performs a subsequence matching on the structure-encoded sequences without breaking down the twig query into sub-queries to find twig patterns in XML documents. Also, to quickly determine the structural relationship between two nodes, each node of the structure-encoded sequences is assigned with a pair of numbers (n_x, $size_x$), where n_x is the tree traversal order by preorder, and where $size_x$ is the total number of descendants of a node, x, in the suffix tree [5]. Rao *et al*. have proposed PRIX that transforms XML data trees and queries into *LPS* (*Labeled Prüfer Sequence*) and *NPS* (*Numbered Prüfer Sequence*) to improve the drawbacks of query processing cost and *false alarms* of the ViST [6]. Tatikonda *et al*. have proposed LCS-TRIM that uses *CPS* (*Consolidated Prüfer Sequence*) instead of *LPS* and *NPS* of PRIX to reduce the sequence matching cost of PRIX [7]. However, ViST has other imminent drawbacks, and PRIX and LCS-TRIM require much processing time for structure matching of XML data trees and queries. In addition, the LCS-TRIM that's based on the main memory index is unsuitable for searching large XML documents.

In this paper, we propose an efficient index structure to solve the problems of the ViST, and we propose a novel query processing method that's suitable for the proposed index structure. The main contributions of this paper are summarized as follows. We propose a new structure-encoded sequence with the *Durable* numbering scheme [1] to support dynamic data insertion and deletion. We propose a minimum sequence matching scheme to speed up the subsequence match phase and to return correct answers without *false alarms* when a query is processed. Our approach is directly performed on the disk-based B+Tree and R-Tree, instead of relying on specialized data structures that are not well supported by DBMSs. We compare our index structure with ViST, PRIX and LCS-TRIM in terms of query processing costs to verify the superiority of our proposed index structure for being a minimum sequence matching scheme.

The rest of this paper is organized as follows. Section 2 discusses the background and motivations of our work. Section 3 proposes the new query indexing method, and section 4 presents our experimental results. Finally, section 5 summarizes the conclusion of this paper.

2. Background and Motivations

2.1 XML Numbering Schemes

The structural relationship between two element-nodes can be quickly determined by a region encoding scheme, where each element is assigned with a pair of numbers (*start*, *end*), based on the element's position in the data tree [2][3][8], with the following held: for any two distinct elements *u* and *v*, (1) the region of *u* is completely before or after *v*, or (2) the region of *u* completely contains *v* or is contained by the region of *v*. Formally, element *u* is an ancestor of element *v* iff $u.start < v.start$ and $v.end < u.end$. Since regions of two distinct elements never partially intersect, the formula can be simplified as $u.start < v.start < u.end$. Usually, region codes for element nodes can be effectively generated by a depth-first traversal of the tree and sequentially assigning a number at each visit.

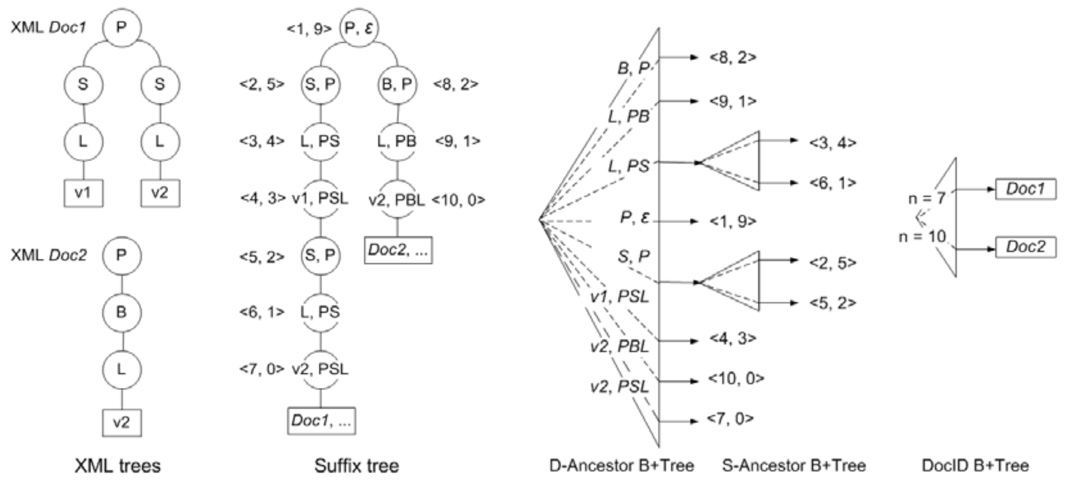
There are other approaches to numbering XML element nodes. One *Dietz's* numbering scheme uses tree traversal orders [9]. A tree node is assigned a pair of (*preorder*, *postorder*) tree traversal orders. Element *u* is an ancestor of element *v* iff $u.preorder < v.preorder$ and $v.postorder < u.postorder$. The limitation of this approach is the lack of flexibility. That is, the preorder and postorder may need to be recomputed for many tree nodes when a new node is inserted. The *Durable* numbering scheme is proposed to more efficiently deal with the dynamic updates of XML data. This approach assigns a pair of (*order*, *size*) to each node in tree [1]. The *order* is an extended preorder and *size* is an arbitrary integer that's larger than the total number of the current descendants of each node in a tree to gracefully accommodate future insertions. Element *u* is an ancestor of element *v* iff $u.order < v.order < u.order+u.size$.

2.2 Problems of ViST

Wang *et al.* have proposed a new method called ViST that transforms XML data trees and twig queries into structure-encoded sequences [4]. The structure-encoded sequence is a two-dimensional sequence of (*symbol*, *prefix*) pairs $\{(a_1, p_1), (a_2, p_2), \ldots, (a_n, p_n)\}$ where a_i represents a node in the XML document tree, and pi represents the path from the root node to a node a_i. The nodes $a_1, a_2, \ldots, a_n$ are in preorder. Also, to quickly determine the structural relationship between two nodes, each node of structure-encoded sequences has a pair of numbers $(n_x, size_x)$, where n_x is the tree traversal order by preorder, and $size_x$ is the total number of descendants of a node *x* in the suffix tree. If *u* and *v* are labeled $(n_u, size_u)$ and $(n_v, size_v)$ respectively, then node *u* is an ancestor of node *v* iff $n_v \in (n_u, n_u+size_u]$. ViST has a D-Ancestor B+Tree, S-Ancestor B+Trees and a DocID B+Tree. The D-Ancestor B+Tree indexes the structure-encoded sequences with their (*symbol*, *prefix*) as keys. Each S-Ancestor B+Tree indexes the n_x and $size_x$ values with the same (*symbol*, *prefix*). Moreover, a DocID B+Tree, using the n_x values of suffix tree's leaf nodes as keys, indexes the document's IDs involving the structure-encoded sequence of each path in a suffix tree. **Fig. 1 (a)** shows the suffix tree in ViST for XML Doc1 and Doc2. **Fig. 1 (b)** shows ViST on the suffix tree in **Fig. 1 (a)**.

Suppose a node *x*, labeled $(n_x, size_x)$, is one of the nodes matching a query $q_1, \ldots, q_{i-1}$. To match the next element q_i in the query, ViST consults the D-Ancestor B+Tree using q_i as a key. The D-Ancestor B+Tree returns the root of an S-Ancestor B+Tree. ViST then issues a range query $n_x < n \leq n_x+size_x$ on the S-Ancestor B+Tree to immediately find the descendants of node *x*. For each descendant, ViST uses the same process to match symbol q_{i+1}, until ViST reaches the last element of the query. If a node, *y,* is one of the nodes that matches the last element in the query, then ViST performs a range query $(n_y, n_y+size_y]$ on the DocID B+Tree to retrieve all the documents' IDs for *y* or *y*'s descendants. **Fig. 2** shows the query processing phase of ViST in **Fig. 1**.

ViST performs the subsequence matching of the structure-encoded sequences to optimize twig query processing without breaking a twig and it merges the results of sub-queries in XML documents. However, ViST has imminent drawbacks. One of those drawbacks, called Unacceptable Accesses, is that determining the structural relationship between two nodes is incorrect because the numbering scheme of ViST is associated after all the nodes of a XML document tree are indexed on a path of a suffix tree.



(a) The suffix tree for *Doc1* and *Doc2* (b) The structure of ViST

Fig. 1. The ViST for *Doc1* and *Doc2*

Example 1 The child node of (S, P) with n=2 is only (L, PS) with n=3 in *Doc1* of **Fig. 1**. However, when the range query (*symbol*, *prefix*) of (L, PS) involved in (S, P) is executed, (L, PS) with n=6 is also answered as the child of (S, P) according to $n_{(S,\ P)} < n_{(L,\ PS)} \leq n_{(S,\ P)}+size_{(S,\ P)}$.

Another drawback of ViST, called as *Unnecessary Accesses*, is that ViST accesses unnecessary nodes when processing a query. The *prefix* of the structure-encoded sequence represents the path from the root node to its parent node. If ViST utilized the characteristic of *prefix*, then ViST would reduce many I/O costs of determining the structural relationship between two nodes.

Example 2 With the *PSL* of (*v1*, *PSL*) in **Fig. 1**, we can determine that (*v1*, *PSL*) has (L, PS) as its parent node, and (P, ε) and (S, P) as its ancestor nodes. Therefore, the range queries of (P, ε), (S, P) and (L, PS) involving (*v1*, *PSL*) are unnecessary, as is shown in **Fig. 2**.

Also, the query processing strategy of ViST may result in *false alarms*. **Fig. 3** illustrates such a case. The structure-encoded sequence of the twig query Q is a subsequence of *Doc1* and *Doc2*. However, the twig pattern, Q, occurs only in *Doc1*, and the match that's detected in *Doc2* is a *false alarm*.

2.3 The Problems of PRIX and LCS-TRIM

Rao *et al.* have proposed PRIX to improve drawbacks of the query processing cost and the *false alarms* of ViST [6]. PRIX transforms XML data trees and twig queries into *LPS* and *NPS* as in **Fig. 4**. In PRIX, each node of a XML document tree and a twig query tree has a unique number according to the postorder numbering scheme. *LPS* and *NPS* can be then constructed according to the node removal method. To construct a sequence from a tree T_n with n nodes labeled from 1 to n, the node removal method works as follows. From T_n, delete a leaf with the smallest label to form a smaller tree T_{n-1}. Let a_1 denote the label of the node that was the parent of the deleted node.

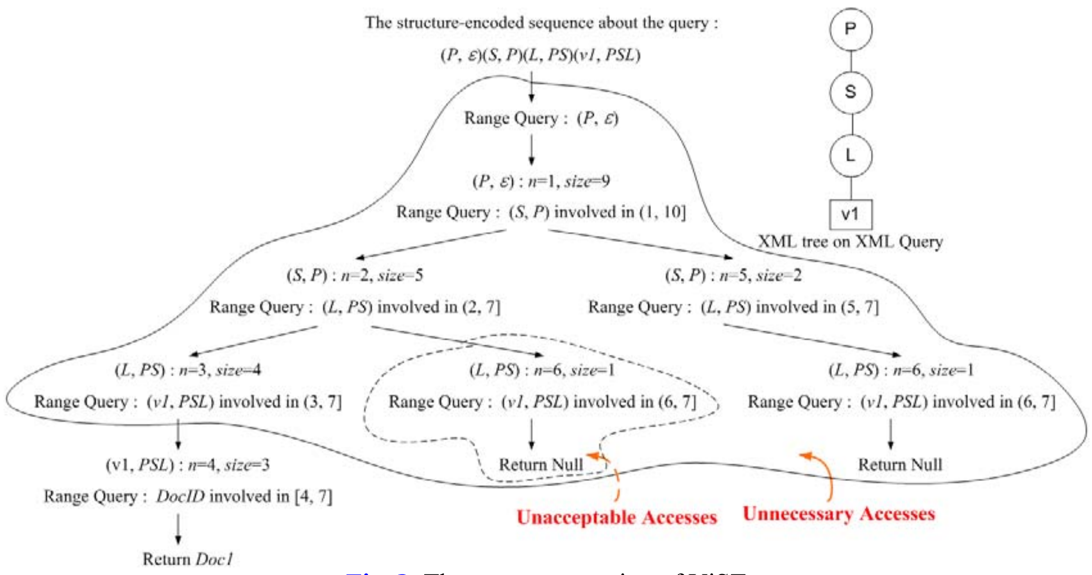
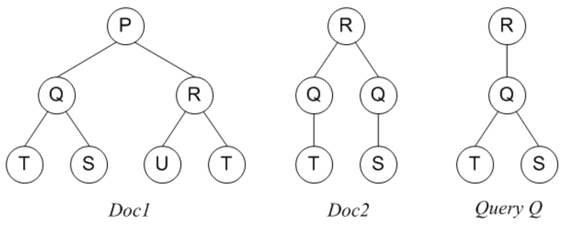

The structure-encoded sequence about the query :
(P, ε)(S, P)(L, PS)(v1, PSL)
Range Query : (P, ε)
(P, ε) : n=1, size=9
Range Query : (S, P) involved in (1, 10]
(S, P) : n=2, size=5
Range Query : (L, PS) involved in (2, 7]
(S, P) : n=5, size=2
Range Query : (L, PS) involved in (5, 7]
(L, PS) : n=3, size=4
Range Query : (v1, PSL) involved in (3, 7]
(L, PS) : n=6, size=1
Range Query : (v1, PSL) involved in (6, 7]
(L, PS) : n=6, size=1
Range Query : (v1, PSL) involved in (6, 7]
(v1, PSL) : n=4, size=3
Range Query : DocID involved in [4, 7]
Return Doc1
Return Null
Return Null
Unacceptable Accesses
Unnecessary Accesses
P
S
L
v1
XML tree on XML Query

Fig. 2. The query processing of ViST



$Doc1 = (\underline{P, \varepsilon})(\underline{Q, P})(\underline{T, PQ})(\underline{S, PQ})(R, P)(U, PR)(T, PR)$

$Doc2 = (\underline{P, \varepsilon})(\underline{Q, P})(\underline{T, PQ})(Q, P)(\underline{S, PQ})$

$Q = (\underline{P, \varepsilon})(\underline{Q, P})(\underline{T, PQ})(\underline{S, PQ})$

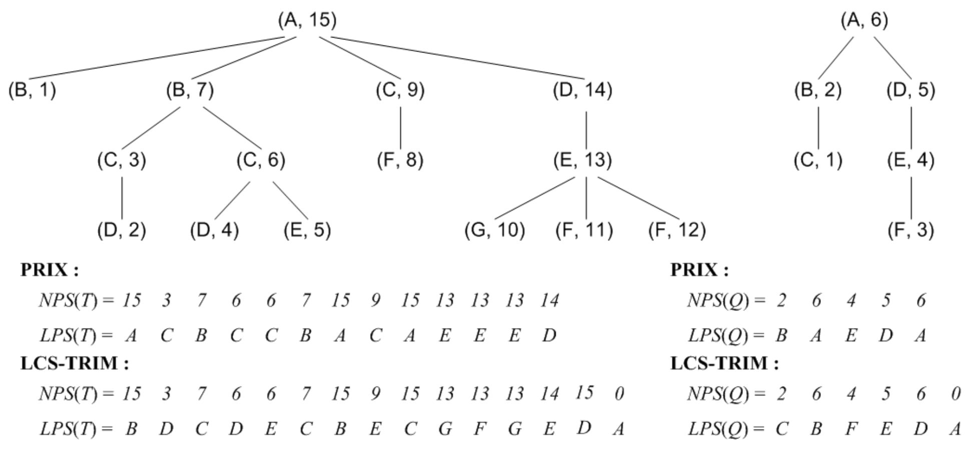
Fig. 3. *False Alarms* by ViST

Repeat this process on T_{n-1} to determine a_2 (the parent of the next node to be deleted), and continue until only two nodes that are joined by an edge are left. The sequence (a_1, a_2, a_3, …, a_{n-2}) is called the *Prüfer* sequence of T_n. If the *Prüfer* sequence is constructed with numbers assigned to nodes, then it is called *NPS*. If the *Prüfer* sequence is constructed with labels assigned to nodes, then it is called *LPS*.

A twig matching of PRIX can be found by performing a subsequence matching on the set of *LPS* and *NPS*, and by performing a structure matching with gap consistent, frequency consistent and matching leaf nodes [6]. The twig matching is faster than that of ViST. However, the structure matching of PRIX needs many disk I/O and time.

Tatikonda *et al*. have proposed LCS-TRIM to reduce the query processing cost of PRIX [10]. LCS-TRIM uses *CPS* that consists of *LS* (*Label Sequence*) and *NPS*. The *NPS* of LCS-TRIM is similar to that of PRIX. However, PRIX doesn't take the sequence of the root node, but LCS-TRIM takes it. Also, *LS* takes the labels of the deleted nodes instead of their parent node labels. The sequence matching by using *LS* outperforms that by using *LPS* because *LPS* additionally performs the matching of leaf nodes. Moreover, LCS-TRIM uses *LF*

(*Label Filtering*) and *DM* (*Dominant Match*) to reduce the number of unnecessary node accesses during structure matching. However, when processing queries with wild-cards, *LF* and *DM* significantly degrade the query performance. Moreover, The LCS-TRIM that's based on main memory index is unsuitable for searching large XML documents.



PRIX :

NPS(T) = 15 3 7 6 6 7 15 9 15 13 13 13 14

LPS(T) = A C B C C B A C A E E E D

LCS-TRIM :

NPS(T) = 15 3 7 6 6 7 15 9 15 13 13 13 14 15 0

LPS(T) = B D C D E C B E C G F G E D A

PRIX :

NPS(Q) = 2 6 4 5 6

LPS(Q) = B A E D A

LCS-TRIM :

NPS(Q) = 2 6 4 5 6 0

LPS(Q) = C B F E D A

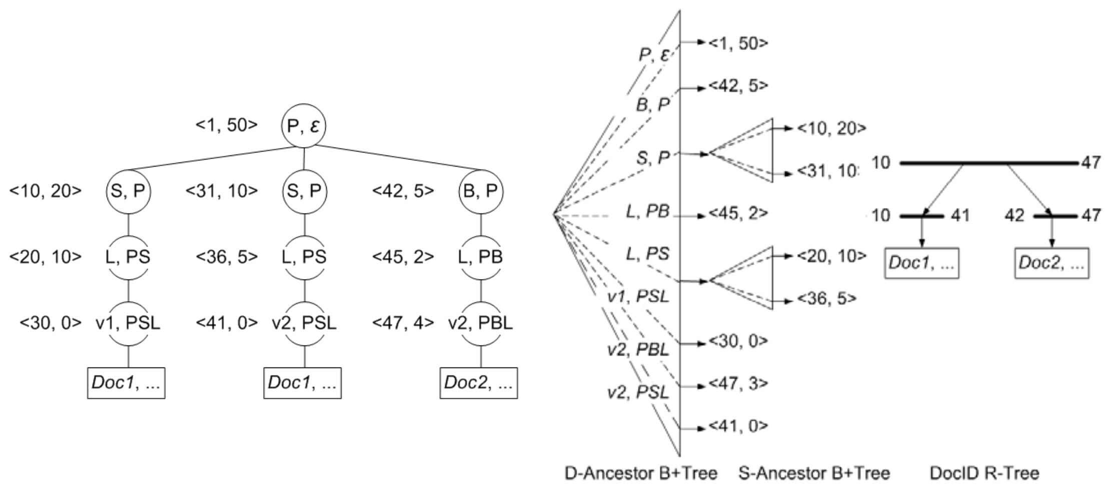
(a) XML document tree *T* (b) Twig query tree *Q*

Fig. 4. Sequences for a XML document tree and a twig query tree with PRIX and LCS-TRIM

3. The Proposed Minimum Sequence Matching Scheme

3.1 The Proposed Index Structure

Our index structure is based on ViST. One notable difference is that we use the *Durable* numbering scheme [1] to avoid the *Unacceptable Accesses* of ViST and to more efficiently deal with dynamic updates of XML data. **Fig. 5** shows our proposed index structure on XML *Doc1* and *Doc2* of **Fig. 1**.



(a) The suffix tree using the Durable numbering scheme (b) Our index structure

Fig. 5. Our proposed index structure.

Another difference is that each path of the suffix tree used in our index structure is composed of each of the paths of all XML data trees, whereas each path of the suffix tree used in ViST is composed of all the nodes of each XML data tree. The D-Ancestor B+Tree and the S-Ancestor B+Trees of our index structure are the same as those of ViST. However, each S-Ancestor B+Tree of our index structure indexes the $order_x$ and $size_x$ values assigned by the *Durable* numbering scheme. Also, we use DocID R-Tree that uses the second smallest n_x values and the largest n_x values of each document as keys. Therefore, our index can find all the DocIDs with n_x values of all nodes.

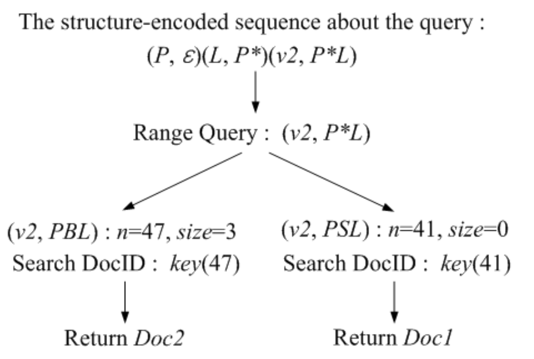
3.2 The Proposed Minimum Sequence Matching Schemes

If we use the characteristic of *prefix* as mentioned above, the query performance can be significantly improved. We propose achieving the minimum sequence matching by using the characteristic of *prefix* and the bottom-up query processing method. In the case of a single path query, as shown in **Fig. 6**, our index only executes the range query of the last node of the structure-encoded sequence of a query. Moreover, as shown in example 3, our query processing method is very efficient for wild-cards queries because the number of node-accesses by our minimum sequence matching is smaller than that by ViST.

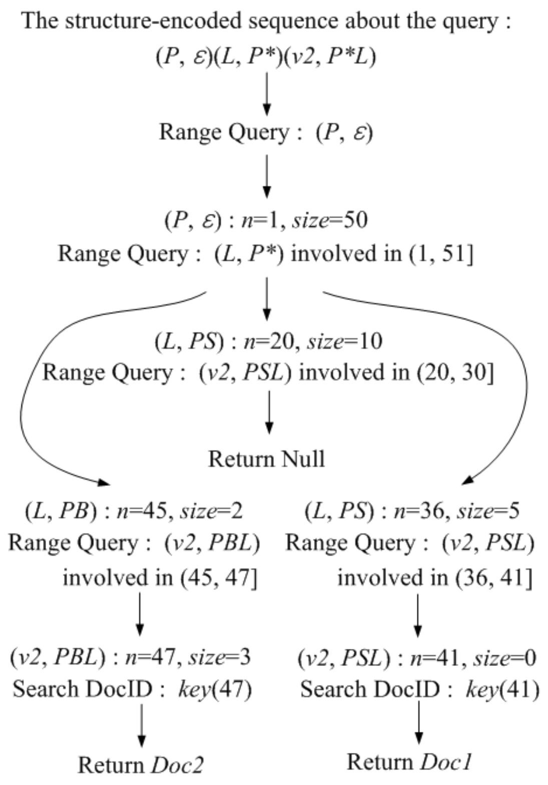
```
Input : Q : q_1, …, q_k, a query sequence
    D-Ancestor B+Tree, index of (symbol, prefix) pairs
    S-Ancestor B+Tree, index of (order, size) labels
    DocID R-Tree, mapping between the order values in node labels and document IDs
Output : all occurrences of Q in the XML data

LinearSearch(Q, k)  /* k is the number of a query sequence's nodes */

Function LinearSearch(Q, k)
1 : while not found T do
2 :   T ← retrieve the S-Ancestor B+Tree that represents q_k from the D-Ancestor B+Tree;
3 :   N ← retrieve all nodes with range inside (order, order+size) from T;
4 :   for each node c ∈ N do
5 :      Perform a query with key(order) on the DocID R-tree to output all document
6 :      IDs in that key;
7 :   end
8 : end
```

Fig. 6. The minimum sequence matching of a single path query

Example 3 Consider a query "*/P/*/L/v2*". The structure-encoded sequence of this query is (P, ε)(L, $P*$)($v2$, $P*L$) and the last node sequence of this query is ($v2$, $P*L$). Therefore, as shown in **Fig. 7 (a)**, our method performs only a range query to match ($v2$, $P*L$) in the D-Ancestor B+Tree of **Fig. 5**. We find ($v2$, PBL) and ($v2$, PSL) as the result of the range query. Then, to find documents involving (*v2*, *PBL*) and (*v2*, *PSL*), range queries are executed in each S-Ancestor B+Tree of (*v2*, *PBL*) and (*v2*, *PSL*). However, the query processing method of ViST has many disk I/O and time because of the top-down query processing method, as is shown in **Fig. 7 (b)**.



(a) Query processing by proposed minimum sequence matching



(b) Query processing by ViST

Fig. 7. Single path query processing phases of our method and ViST

If a twig query is processed by our minimum sequence matching for a single path query, then *false alarms* occur because the structural relationship among multiple sub-queries disassembled from a twig query is not determined. Therefore, we also propose the efficient minimum sequence matching for a twig query; it only executes the range queries about the leaf nodes and branch nodes of a twig query tree, as is shown in example 4.

Example 4 Consider a query "*/P/S[L/v1]/L/v2*". The structure-encoded sequence of this query is (*P*, *ε*)(*S*, *P*)(*L*, *PS*)(*v1*, *PSL*)(*L*, *PS*)(*v2*, *PSL*). The sequence nodes for the leaf nodes of this twig query tree are (*v1*, *PSL*) and (*v2*, *PSL*). And the sequence node for the branch node is (*S*, *P*), as is shown in **Fig. 8 (a)**. **Fig. 8 (b)** shows the twig query processing phases by the proposed minimum sequence matching of a twig query. First, our method performs range queries to match (*v1*, *PSL*), (*v2*, *PSL*) and (*S*, *P*) in the D-Ancestor B+Tree of **Fig. 5**. We find the S-Ancestor B+Trees of (*v1*, *PSL*), (*v2*, *PSL*) and (*S*, *P*) as the results of the above range queries. Then, by range queries of the above S-Ancestor B+Trees, we find (*30*, *0*) as the numbering of (*v1*, *PSL*), (*41*, *0*) as the numbering of (*v2*, *PSL*), and (*10*, *20*) and (*31*, *10*) as the numbering of (*S*, *P*). Finally, to determine the structural relationship among above numbering values, we find the numbering values of branch node sequence involving the numbering values of each leaf node sequence by equation (1). For example, we compute {(30, 30+0]∈10 & (41, 41+0]∈10} and {(30, 30+0]∈31 & (41, 41+0]∈31}. However, since *Doc1* and *Doc2* do not have the sequences of this twig query, the result of this twig query does not exist. As a result, our minimum sequence matching has no *false alarms*.

$$\begin{gathered}(order_{(v1,\,PSL)},\ order_{(v1,\,PSL)}+size_{(v1,\,PSL)}] \in order_{(S,\,P)} \\ \& \\ (order_{(v2,\,PSL)},\ order_{(v2,\,PSL)}+size_{(v2,\,PSL)}] \in order_{(S,\,P)}\end{gathered} \tag{1}$$

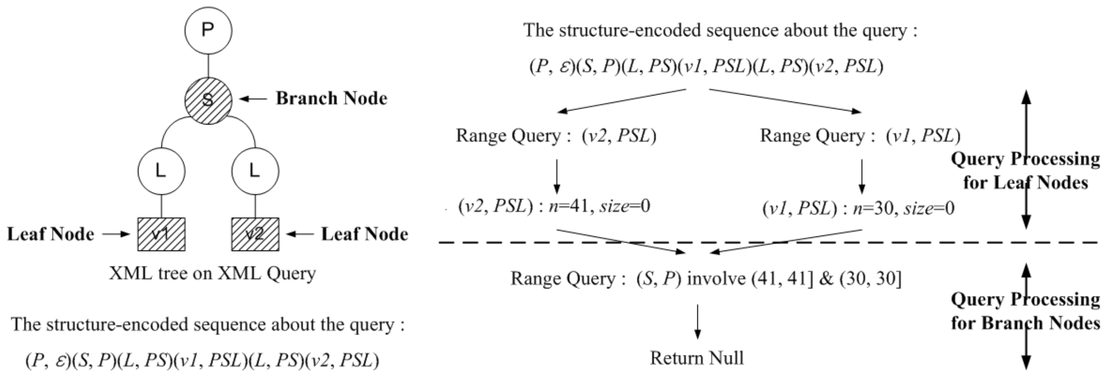

P
S
Branch Node
L
L
Leaf Node
v1
v2
Leaf Node
XML tree on XML Query
The structure-encoded sequence about the query :
(P, ε)(S, P)(L, PS)(v1, PSL)(L, PS)(v2, PSL)
The structure-encoded sequence about the query :
(P, ε)(S, P)(L, PS)(v1, PSL)(L, PS)(v2, PSL)
Range Query : (v2, PSL)
Range Query : (v1, PSL)
(v2, PSL) : n=41, size=0
(v1, PSL) : n=30, size=0
Query Processing for Leaf Nodes
Range Query : (S, P) involve (41, 41] & (30, 30]
Return Null
Query Processing for Branch Nodes

Fig. 8. The twig query processing phases of our method.

4. Performance Evaluation

4.1 Experimental Environment

To determine the effectiveness of our minimum sequence matching scheme, we compare the performance of our index structure with that of ViST, PRIX and LCS-TRIM. We implemented our XML indexing in C++. The implementation uses the B+Tree API provided by GiST [11]. Also, we get the ViST algorithm, the PRIX algorithm and LCS-TRIM algorithm from [12], [13] and [10]. We run our experiments on 3.0GHz Pentium IV processor with 1GB RAM running on Linux kernel 2.6. The code is compiled using the GNU g++ compiler, version 4.0.2. A page size of 8KB is used, and 8-byte number ranges are used to label the nodes. However, the LCS-TRIM used the buffer size of 200MB because the index is based on main memory.

We provided a comparison with LCS-TRIM to verify that the structure matching of LCS-TRIM can cost more. We experiment with the datasets that are obtained from the University of Washington's XML repository [14]. We choose these three datasets since each has a different characteristic, as is shown in **Table 1**. The document tree in the DBLP dataset has good similarity in structure and is shallow. The document tree in the SWISSPROT dataset is bushy and shallow. The document tree in the TREEBANK dataset is skinny and has deep recursions of element names. **Table 1** provides additional information such as the maximum depth, the number of elements and so on for the datasets.

Table 1. The datasets used in our experiments

Data Name	**DBLP**	**SWISSPROT**	**TREEBANK**
Size in MB	134	115	86
# of Elements	3332130	2977031	2437666
# of Attributes	404276	2189859	1
Max-depth	6	5	36
# of Sequences	328858	50000	56385

The XPath queries listed in **Table 2** are tested in our experiments. These queries have different characteristics in terms of selectivity, presence of values and twig structure. Since the values are encrypted, we choose queries without values (character data) for the TREEBANK dataset

Table 2. The XPath queries used in our experiments

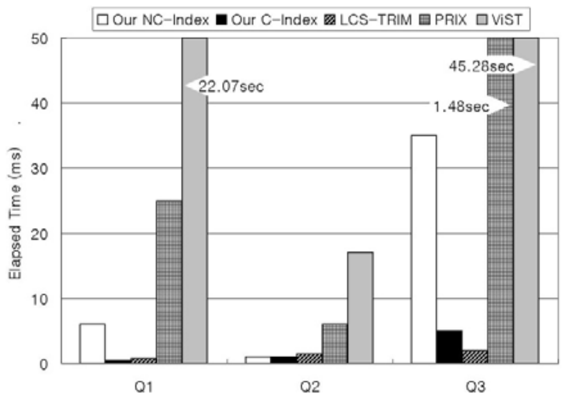
No.	**Query**	**Dataset**
Q_1	//article/author="E. F. Codd"	DBLP
Q_2	//phdthesis[/year][/number]	DBLP
Q_3	//inproceedings[/author="Jim Gray"][/year="1990"]	DBLP
Q_4	//Ref/Author="Moss J"	SWISSPROT
Q_5	//Entry[/Org="Piroplasmida"][/Ref/Author="Kemp D.J"]	SWISSPROT
Q_6	//Entry[/PFAM[@prim-id="PF00304"][//SIGNAL//Descr]	SWISSPROT
Q_7	//S//NP/SYM	TREEBANK
Q_8	//NP[/RBR-OR-JJR]/PP	TREEBANK
Q_9	//NP/PP/NP[/NNS-OR-NN][/NN]	TREEBANK

4.2 Experimental Results

For evaluating various experiments, our index has two types. One, called *C*-index, indexes the structure-encoded sequences and the numbering values of character data on a D-Ancestor B+Tree and S-Ancestor B+Trees like ViST. The second, called *NC*-index, does not index the structure-encoded sequences and the numbering values of character data on a D-Ancestor B+Tree and S-Ancestor B+Trees like PRIX. Therefore, *NC*-index stores the values of character data in a database and each leaf node with character data in S-Ancestor B+Trees points each tuple with its value in a database. If *NC*-index has a query with character data, then the *NC*-index first executes the minimum sequence matching of this query without character data, and the *NC*-index then executes the matching character data between the leaf nodes of this query and the tuples of the database.

Fig. 9 shows the performance results of the total time elapsed and physical I/O (pages read from disk) to process queries Q_1, Q_2 and Q_3 of the DBLP dataset. The physical I/O of

LCS-TRIM represents the size of the used main memory. Q_1 is a single path query with two element nodes and one character data node. ViST and PRIX execute range queries on all nodes of the structure-encoded sequence for Q_1. LCS-TRIM requires scanning all of the main memory to construct *R-matrix* for processing queries [10]. However, *C*-index executes the range query of the leaf node of the structure-encoded sequence of Q_1. Therefore, our index is better than the other indexes. *C*-index is better than *NC*-index because *NC*-index additionally requires the matching character data between the leaf node of Q_1 and the tuple of the database. Q_2 and Q_3 are twig queries with three element nodes and two branches, and with three element nodes, two character data nodes and two branches, respectively. Processing a single path query is better than processing a twig query because processing a twig query additionally requires determining the structural relationship between multiple sub-queries disassembled from a twig query. The performance results of *NC*-index and *C*-index to process Q_2 are the same because Q_2 has no character data. Also, the performance results of Q_2 are better than those of Q_1 because 'year' element and 'number' element as a child of 'phdthesis' element are more than 'author' element as a child of 'article' element in the DBLP dataset. The performance results of *NC*-index to process Q_3 are more faulty than those of the others because 'author' element and 'year' element as a child of 'inproceedings' element are plentiful in the DBLP dataset and they additionally require the matching character data to process 'Jim Gray' element and '1990' element. LCS-TRIM is better than our index because the structure matching of LCS-TRIM is executed in the main memory.



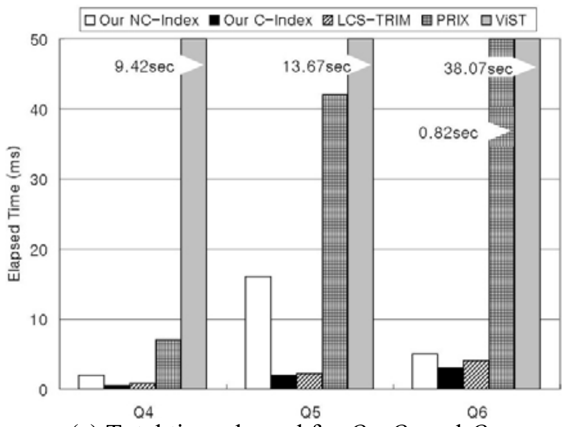
(a) Total time elapsed for Q_1, Q_2 and Q_3

	***NC*-Index**	***C*-Index**	**PRIX**	**ViST**	**LCS-TRIM**
Q_1	8 pages	1 pages	23 pages	2,280 pages	23.61 MB
Q_2	2 pages	2 pages	8 pages	17 pages	23.61 MB
Q_3	39 pages	7 pages	185 pages	3,543 pages	23.61 MB

(b) Total physical I/O of Q_1, Q_2 and Q_3

Fig. 9. The performance results of Q_1, Q_2 and Q_3.

Fig. 10 shows the performance results of processing queries Q_4, Q_5 and Q_6 of the SWISSPROT dataset. Q_4 is the same single path query as Q_1. Processing Q_4 is better than processing Q_1 because the elements that are related to Q_1 are more than those related to Q_4 of the SWISSPROT dataset. The performance results of *C*-index to process Q_1 and Q_4 are the same because the number of character data of Q_1 and Q_4 is same. Q_5 is the same twig query as Q_3. Also, as shown in the performance results to process Q_1 and Q_4, processing Q_5 is better than the processing Q_3 because the elements that are related to Q_3 are more than those related to Q_5. Q_6 is a twig query with an attribute node and a wild-card '//'. The performance results to process Q_6 are more erroneous than those to process Q_4 and Q_5 because of the many range queries to process the wild-card. But *NC*-index to process Q_6 is better than that to process Q_5 because the '@prim_id' attribute and 'Descr' element that are related to Q_6 are more than 'Org' and 'Author' elements that are related to Q_5 of the SWISSPROT dataset. If a query has wild-cards, then *LF* and *DM* of LCS-TRIM don't construct the small *R-matrix* because almost all the nodes of main memory are inserted into *R-matrix* to find the ancestor nodes of the nodes that are involved in the query. Therefore, the cost of structure matching is increased.



(a) Total time elapsed for Q_4, Q_5 and Q_6

	NC-Index	*C*-Index	PRIX	ViST	LCS-TRIM
Q_4	2 pages	1 pages	9 pages	1,657 pages	23.14 MB
Q_5	18 pages	2 pages	49 pages	1,885 pages	23.14 MB
Q_6	7 pages	4 pages	83 pages	4,367 pages	23.14 MB

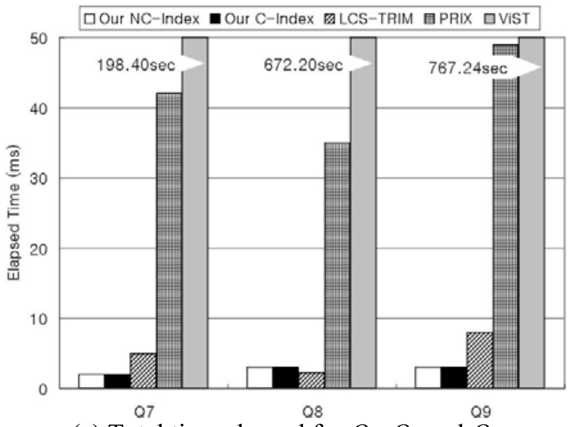
(b) Total physical I/O of Q_4, Q_5 and Q_6

Fig. 10. The performance results of Q_4, Q_5 and Q_6.

Fig. 11 shows the performance results of processing queries Q_7, Q_8 and Q_9 of the TREEBANK dataset. Q_7 is a single path query with two wild-cards '//'. To process Q_7, ViST performs the range queries (*S*, *//*), (*NP*, *//S//*) and (*SYM*, *//S//NP*). The results of these range queries cause many other range queries to produce final answers. PRIX performs the range queries 'S', 'NP' and 'SYM'. Also, PRIX performs the subsequence matching and refinement phases on the set of *LPS* and *NPS* of the results of these range queries. Our index only performs the range query (*SYM*, *//S//NP*). Therefore, our index clearly outperforms other indexes. The TREEBANK dataset has no character data because the character data are encrypted. The performance results of *NC*-index and *C*-Index to process each query are the same. Q_8 and Q_9 are the twig queries with a wild-card '//'. As shown in **Fig. 11**, the performance results of our index to process a single query are better than those to process a twig query because our index additionally requires the range queries of vertex nodes to process a twig query. However, the performance results of PRIX to process Q_8 are better than those to process Q_7 because the number of *LPS* and *NPS* of processing Q_7 are more than those of processing Q_8. Also, in LCS-TRIM, if the depth of the document tree is high and a query has wild-cards, then the size of *R-matrix* is much increased.

Fig. 12 shows the performance results of *false alarms*. PRIX performs a series of refinement phases with *gap consistent* and *frequency consistent* to avoid the *false alarms* of ViST.

LCS-TRIM performs structure matching by *LF* and *DM*. Our index performs the minimum sequence matching according to the *Durable* numbering scheme. As shown in **Fig. 12**, PRIX, LCS-TRIM and our index have no *false alarms*. However, ViST has many *false alarms*. For example, the DBLP dataset has 72 ‘phdthesis’ elements, 72 ‘year’ elements as a child of ‘number’ and 3 elements as a child of ‘phdthesis’. Also, there are 3 ‘phdthesis’ elements with ‘year’ element and ‘number’ element as a child. However, ViST has 4602 ‘phdthesis’ elements as the result of Q_2 because if a node, *y,* is a child or a descendant of a node *x* on ViST, then ViST treats node *y* as a child or a descendant of node *x* in spite of the fact that node *y* is not a child or a descendant of node *x* in a XML document.



(a) Total time elapsed for Q_7, Q_8 and Q_9

	NC-Index	*C*-Index	PRIX	ViST	LCS-TRIM
Q_7	3 pages	3 pages	46 pages	40,827 pages	11.73 MB
Q_8	4 pages	4 pages	35 pages	94,505 pages	11.73 MB
Q_9	4 pages	4 pages	55 pages	121,928 pages	11.73 MB

(b) Total physical I/O of Q_7, Q_8 and Q_9

Fig. 11. The performance results of Q_7, Q_8 and Q_9.

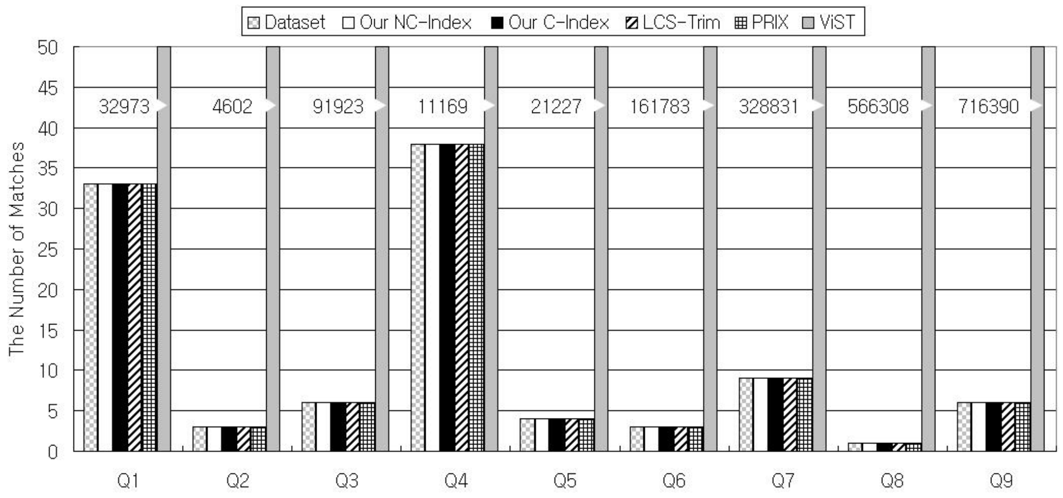


Fig. 12. The performance results of the *false alarms*

5. Conclusion

In this paper, we have presented an efficient paradigm for the XPath query processing. We have proposed a novel index structure that uses the *Durable* numbering scheme and the structure-encoded sequences of the XML tree for indexing XML data. Then, to quickly process XPath queries, we have proposed the minimum sequence matching scheme using the characteristic of the prefix. We have also provided empirical performance analysis to demonstrate the efficient processing of XML queries when using our index structure. In the future, we will study how to efficiently process the delimiters of the prefix schemes, to decrease the label size and to maintain low-label update.

References

[1] Q. Li and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions," *Proc. of 27th VLDB Conference*, pp.361-370, 2001.
[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," *Proc. of 18th IEEE International Conference on Data Engineering*, pp.141-152, 2002.
[3] S. Y. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo, "Efficient Structural Joins on Indexed XML Documents," *Proc. of 28th VLDB Conference*, pp.263-274, 2002.
[4] H. Wang, S. Park, W. Fan, and P. S. Yu, "ViST: A Dynamic Index Method for Querying XML Data by Tree Structures", *Proc. of 2003 ACM SIGMOD Conference*, pp.110-121, 2003.
[5] E. M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," *Journal of the ACM*, Vol. 23, pp.262-272, 1976.
[6] P. Rao and B. Moon, "Sequencing XML Data and Query Twig for Fast Pattern Matching," *ACM Transactions on Database Systems(TODS)*, pp.299-345, 2006.
[7] S. Tatikonda, S. Parthasarathy, and M. Goyder, "LCS-TRIM: Dynamic Programming Meets XML Indexing and Querying," *Proc. of 2007 VLDB Conference*, pp.63-74, 2007.
[8] C. Zhang, J. F. Naughton, D. J, DeWitt, Q. Luo, and G. M. Lohman, "On Supporting Containment Queries in Relational Database Management Systems," *Proc. of 2001 ACM SIGMOD Conference*, pp.425-436, 2001.
[9] P. F. Dietz, "Maintaining Order in a Linked List," *Proc. of the 4th Annual ACM Symposium on Theory of Computing*, pp.122-127, 1982.
[10] S. Tatikonada, "LCS-TRIM Project," http://www.cse.ohiostate.edu/~takidond, 2007.
[11] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, "Generalized Search Trees for Database Systems," *Proc. of the 21th VLDB Conference*, pp.562-573, 1995.
[12] H. Wang, "The ViST Algorithm," http://wis.cs.uda.edu/~hxwang/pub.html, 2003.
[13] B. Moon, "PRIX Project," http://www.cs.arizona.edu/prix, 2006.
[14] G. Miklau, "UW XML Repository," http://www.cs.washington.edu/research/xmldatasets, 2006.

Dong-Min Seo received his B.S., M.S. and Ph.D. in Information and Communication Engineering from Chungbuk National University, Korea in 2002, 2004 and 2008, respectively. He is now the postdoctoral in the Dept. of Computer Science, the Korean Advanced Institute of Science and Technology, Korea. His main research interests include MOD (Moving-Objects Database) system, WSN (Wireless Sensor Networks) and XML database system.

Myung-Ho Yeo received his B.S. and M.S. in Information and Communication Engineering from Chungbuk National University, Korea in 2004 and 2006, respectively. He is currently working towards his Ph.D. degree in the Dept. of Information and Communication Engineering from Chungbuk National University, Korea. His main research interests include main-memory database systems and WSN.

Myoung-Ho Kim received his B.S. and M.S. in Computer Engineering from Seoul National University, Korea in 1982 and 1984, respectively. He received his Ph.D. in Computer Science from Michigan State University, USA. He is now a professor in the Dept. of Computer Science, the Korean Advanced Institute of Science and Technology, Korea. His main research interests include database system, sensor data management, and storage management system.

Jae-Soo Yoo received his B.S. in Computer Engineering from Chunbuk National University, Korea in 1989, and he also received his M.S. and Ph.D. in Computer Science from the Korean Advanced Institute of Science and Technology, Korea in 1991 and 1995. He is now a professor in Information and Communication Engineering, Chungbuk National University, Korea. His main research interests include database systems, sensor data management, location based services, distributed computing and storage management system.