A Complete Path Representation Method with a Modified Inverted Index for Efficient Retrieval of XML Documents

Hsu-Kuang Chang [*, **], King-Chu Hung [**], and I-Chang Jou [*]

[*]Department of Information Engineering,

I-Shou University,

No.1, Sec. 1, Syuecheng Rd., Dashu District, Kaohsiung City 84001

Taiwan, R.O.C.

[**] Department of Electronic Engineering,

National Kaohsiung First University of Science and Technology,

2 Jhuoyue Rd., Nanzih, Kaohsiung City, 811

Taiwan, R.O.C.
hkchang@isu.edu.tw, kchung@ccms.nkfust.edu.tw, and icjou@isu.edu.tw

Abstract: - Compiling documents in extensible markup language (XML) increasingly requires access to data services which provide both rapid response and the precise use of search engines. Efficient data service should be based on a skillful representation that can support low complexity and high precision search capabilities. In this paper, a novel complete path representation (CPR) associated with a modified inverted index is presented for the provision of efficient XML data services, where queries can be versatile in terms of predicates. CPR can completely preserve hierarchical information, and the new index is used to save semantic information. The CPR approach can provide template-based indexing for fast data search. An experiment is also conducted for the evaluation of the CPR approach.

Key-Words: - XML, DTD, Complete path representation (CPR), Structural summary tree (SST), versatile query

1 Introduction

By recommendation of the World Wide Web Consortium (W3C) [1], XML has served as a standard information description language widely used in communications of computers. This facility demands skillful XML documents (or data) representation and indexing for fast data search in a large XML database [2]. Since XML data can be uniquely described with a semantically structured tree (i.e., a hierarchical structure associated with relationships among nodes), both structure and semantics are significant feature elements of XML data representation.

Traditional XML data representations can be categorized into string [3-4] and path [5-15] groups. String representation can be derived with a preorder traversal algorithm; it requires dynamic programming for edit distance measurement. This approach, with its lack of structure information, may lead to indeterminate search results. Path approaches use sub-paths as feature elements, and represent each XML datum with a binary vector. An element in the binary vector denotes whether the datum involves a corresponding feature, where such features can be defined as start-end tags [5-6], two-node sub-paths (i.e., node-pair (NP)) [7], Xpaths and whole paths (WP) [8-10]. For search efficiency improvement, modified path approaches were also proposed. Yang *et al*. [11] used the content instead of the leaf node for node representation. Liu *et al.* [12] combined NP and WP for XML data description. Based on the determined finite automata, Mustafa *et al*. [13] and Lee *et al*. [14] presented a path-embedded string representation.

For improving common Xpath efficiency and principal component analysis, Li [15] presented a modified WP with limited-length paths.

The elements of NP and WP are commonly indexed with inverted index list [24] or B*-tree structure [19] [27] [28]. These two indices only recording tag names and pointers do not provide enough information useful for inferring the existence undefined features, e.g., sub-paths by level, ancestor-descendant (AD) paths, sibling (SB) and cousin (CN) relations. These features are usually required for versatile queries. Based on XML QBE [16], XQuery [17] and query pattern tree (QPT) [19], several stack-based parser algorithms, PathStack, TwigStack, and TJFast [18] [20] [23], have been proposed for the search of AD elements (i.e., sub-paths). These stack-based parsers can check only one document for each parsing. This query service manner will be time-consuming for large database. For versatile queries service, stack-based parsers can be also used for extracting all the AD elements of a large database. However, this approach do not provide any indexing for fast AD element search. This implies that exhaustive search of AD features is necessary for each query service. Moreover, stack-based parsers are inefficient for searching the features with SB and CN relations.

In this paper, a novel path approach referred to as complete path representation (CPR) is proposed. The CPR scheme that uses complete path elements (CPEs) as representation features can well preserve the hierarchical structure of XML data. A modified inverted index structure is also presented for recording the CPEs and hierarchically semantic data, i.e., path lengths and levels. The semantic data are useful for extracting the feature elements with AD, SB, and CN relationships. The modified index can also provide a template-based indexing for fast XML data search.

This paper is organized as follows: In Section 2, the CPR is proposed for XML data description. In Section 3, the modified inverted index is described. Section 4 shows the performance evaluation results of handling versatile queries by using variant approaches. Finally, conclusions are drawn in Section 5.

2 XML DATA REPRESENTATION USING COMPLETE PATH ELEMENTS

Any XML datum defined with the document type definition (DTD) can be modeled as an ordered label tree [3]. In this section, the hierarchical tree information is extracted by a pre-ordered traversal process performed with a document object model (DOM) API [1]. Following this, the CPE extraction based on the SSTs of XML data is described and the CPR is presented.

2.1 The Extraction of Structural Summary

SST is the XML tree skeleton commonly used for XML data representation [3]. The SST of an XML document can be extracted by four functional processes, as follows:

Step 1. This step performs a tree conversion using Java DOM (JDOM), where the tree element values are neglected. For the DTD-formatted XML datum of Example 1, the tree conversion process result is illustrated in Fig. 1.

Step 2. For efficient matching, we symbolize the name of the tree node with an abbreviated character order, as shown in Figure 2.

Step 3. Based on the pre-order traversal process [3], SST extraction requires two simplification procedures:

a) For each node, examine whether the current node's name is equal to an ancestor's name. If it is, set the current node's sub-tree to be a child of the ancestor; otherwise, check the next node. The purpose of this procedure is to remove nested sub-paths, as shown in Fig. 3.

b) Exhaustive searching based on a Hash table is applied for discovering and eliminating repeated branches. Fig. 4 shows the repeated branch elimination result where the simplified tree is the SST.

Step 4. For the extraction of CPEs, it is necessary to construct the adjacent-linked (AL) lists of the SSTs of all XML data. An AL list is a data structure that records the linking information of each node and facilitates the pre-order traversal process. The AL list of the SST in Fig. 4 is given in Table 1 where $\delta_i[n]$ denotes the *n*th

head node of the *i*th XML datum.

Example 1.

<!ELEMENT books (books*, intro*)+>

<!ELEMENT intro (title, author)>

<!ELEMENT title (#PCDATA)>

<!ELEMENT author (firstname, lastname)>

<!ELEMENT firstname (#PCDATA)>

<!ELEMENT lastname (#PCDATA)>

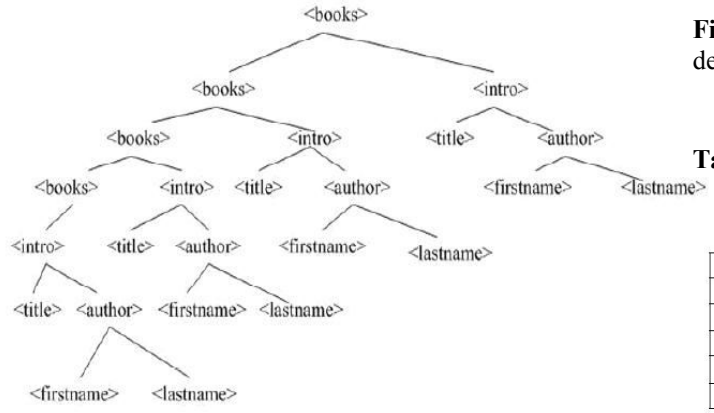


Fig.1. Tree representation of Example 1 based on the JDOM

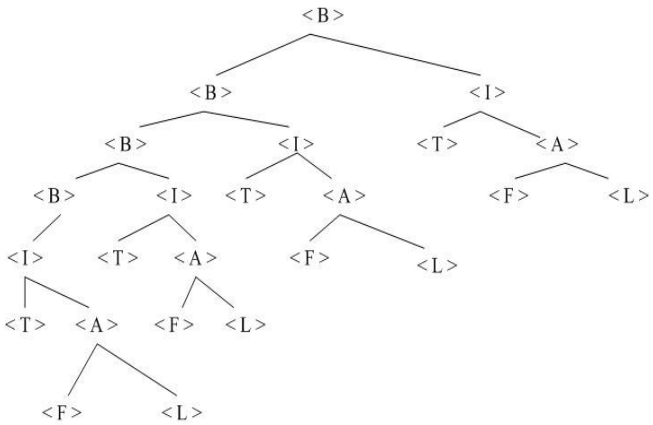


Fig.2. Symbolization process for the XML tree from Example 1

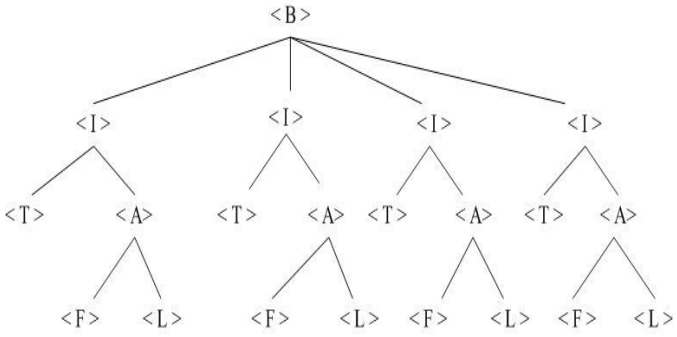


Fig.3. The XML tree of Example 1 with nested nodes removed

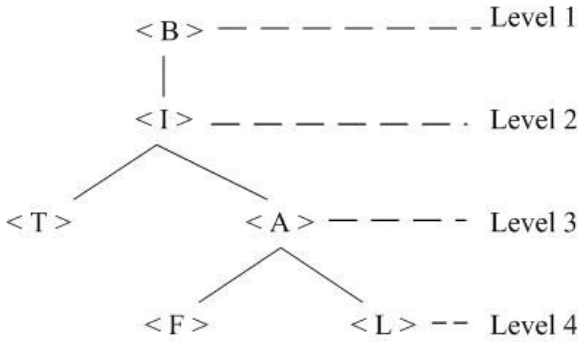


Fig.4. The SST of Example 1 where the level of root is defined as 1 and increased to the leaves

Table 1: The AL list of the SST of Example 1

$\delta_i[0]$	$\delta_i[1]$	$\delta_i[2]$	$\delta_i[3]$	$\delta_i[4]$	$\delta_i[5]$
< B >	< I >	< T >	< A >	< F >	< L >
↓	↓	↓	↓	↓	↓
< I >	< T >	Nil	< F >	Nil	Nil
	↓		↓		
	< A >		< L >		

2.2 Complete Path Element Extraction and Representation

Traditional WP and NP representations, with their lack of linking information, cannot serve such queries as (/B/~/~/~/~) and (/~/I/A/L). For efficient query service, CPR describes XML data with the complete paths (CPs) of all SSTs. The CPs set of a tree is defined as all the branches, (i.e., sub-paths) starting from each level to the leaves. For convenience, let $CP_{L\text{-}i}$ denote a set of CP elements starting from the *i*th level, where the root level is defined as one and is increased toward the leaves. An example of level definition is shown in Fig. 4, where four CP sets: $CP_{L\text{-}1}$, $CP_{L\text{-}2}$, $CP_{L\text{-}3}$, and $CP_{L\text{-}4}$, can be defined. The elements of the four CP sets are shown in Table 2.

Table 2: The four level complete paths of the SST of Example 1

Four CP sets	CP elements							
CP_{L-1}	/B/I/T	/B/I/A/F	/B/I/A/L	/B/I/A/~	/B/I/~/~	/B/I/~	/B/~/~/~	/B/~/~
CP_{L-2}	/~/I/T	/~/I/A/F	/~/I/A/L	/~/I/A/~	/~/I/~/~	/~/I/~		
CP_{L-3}	/~/~/A/F	/~/~/A/L	/~/~/T	/~/~/A/~				
CP_{L-4}	/~/~/~/F	/~/~/~/L						

For the extraction of CP elements, a recursive depth-first search (DFS) algorithm based on the AL list is developed to search all branches, starting from each level. The CP element extraction algorithm is described as follows:

BUILDCPR Algorithm

// AL_i :: AL list of each XML tree.
// δ_i :: Head Node for AL list.
// *stack* :: store the nodes of each level for $AL_i : \delta_i$.
// *CP* :: Collections of Complete Path.
// *BCP* :: Collections of Back-Track CP.
// *CPs* :: Collections of unique CPR by the level.
// Count-Tree-Level :: A function returns the tree level for $AL_i : \delta_i$.
// Count-Level-Nodes :: A function counts the nodes of the certain level for $AL_i : \delta_i$.
// Build-Path :: A function constructs the CP for node with the level.
// PUSH(), POP() :: Two operations for stack.

```
1.  for each XML tree Xi, i← 1 to k do          //each XML tree in AL :: δ1 ~ δk
2.      Tree-Levels = Count-Tree-Level (ALi : δi) // Count tree level
3.      for Level ← 1 to Tree-Levels do          // trace level by level
4.          stack ←Nil   // clear stack
5.          Nodes = Count-Level-Nodes(ALi : δi) // Count node this tracing level
6.          PUSH(Nodes, stack)                   // Push parsing nodes into the stack
7.          while not Empty(stack) do
8.              V = POP(stack)
9.              CP ← Build-Path (Level, V)       // Build CP by level
10.             BCP ← <>                         // Initiate back-track CP
11.             DFSCP ELEMENTS ( δi, V, Level, CP, BCP) //Call DFS recursively
12.         end-while
13.     end-for Level
14.     CPs = Remove-Repeated(CP,BCP)            //Remove repeated paths
15. end-for XML
16. end BUILDCPR
```

DFSCP ELEMENTS (X_i, v, ι, ρ, bρ)

// X_i : AL list.
// v: parsing node.
// ι: current level.
// ρ: complete path.
// bρ: backtrack CP.
// *Link (v)* : child node of v.

```
1.  if Link(v) = Nil then
2.      return
3.  for each vertex ω adjacent to v do
4.      Build-ρ (ρ, ι, v) // function build up the path.
5.      DFSCOMPLETEPATH (Xi, ω, ι+1, ρ, bρ)
6.      if ι ≠ Level then
7.          Build-bρ (ρ, ι, "/~") //function build up the backtrack path.
8.  endfor
end DFSCOMPLETEPATH
```

Essentially, DFS is an exhaustive search algorithm that is guaranteed to find all of the branches of an XML tree. Once this is done, the CPR for the description of considerable XML data can be defined as:

$\mathrm{CP}_S = \{\bigcup_{i=1}^{L} CP_{L-i} \mid CP_{L-i}$ is a set involving the i-th level CPs of all XML data$\}$,

where ∪ denotes union operation and L is the maximum level number. Considering a database comprised of the three XML data shown in Fig. 5, the CPR can be found in Table 3. In Fig. 5, there are two I nodes for both DOC 1 and 3. The two nodes with different children are distinct and cannot be merged. The two sub-paths /B/I/T in DOC 2 and /B/I/T/~ in DOC 1 have the same path length equal to 3, but have distinct distances from leaf node. The same distinctions also exist between the two elements /M/I/~ and /M/I/~/~ in the CP_{L-1} .

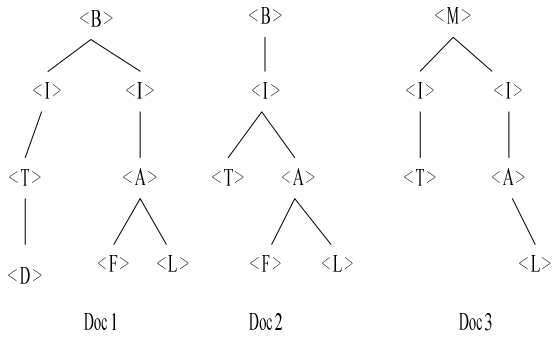


Fig.5: The SSTs for three XML documents

Table 3: The CPR for the description of the three XML data shown in Fig. 5

CPR	complete path elements																
CP_{L-1}	/B/I/T/D	/B/I/A/F	/B/I/A/L	/M/I/A/L	/B/I/T/~	/B/I/A/~	/B/I/~/~	/B/~/~/~	/B/~/~	/M/I/T	/M/I/~/~	/M/I/~	/M/I/A/~	/M/I/~/~	/M/~/~/~	/B/I/~	/B/I/T
CP_{L-2}	/~/I/T/D	/~/I/A/F	/~/I/A/L	/~/I/T/~	/~/I/A/~	/~/I/~/~	/~/I/~	/~/I/T									
CP_{L-3}	/~/~/T/D	/~/~/A/F	/~/~/A/L	/~/~/T/~	/~/~/A/~	/~/~/T											
CP_{L-4}	/~/~/~/D	/~/~/~/F	/~/~/~/L														

3 INDEXING THE COMPLETE PATH ELEMENTS

A CPE with the tree characteristic is a high dimensional feature. Traditional B-tree indexing [27] [28] based on node relationships is suitable for WP, NP and twig queries, but is inefficient for CPR, which regards each CPE as a feature element. In this section, a new index with feature similarity structure (FSS) is presented for CPE management. The FSS provides a fast template-based hierarchical indexing. The CPEs of Table 3 can be represented with a tree structure, as shown in Fig. 6, where P*i* denotes the subset of the CPEs with path length equal to *i*. The CPEs in Fig. 6 are inherent with the hierarchical information involving path length (P*i*) and level (CP_{L-l}) that are available for inferring semantic relations, e.g., ancestor-descendant (AD), sibling (SB) and cousin (CN) relationships. B-tree index with a key design can achieve balanced binary tree structure for efficient NP and WP element indexing, but cannot provide hierarchical information. To facilitate the inference of semantic information, the inverted index structure with additional fields is applied for CPE indexing. These additional fields are used for recoding nodes' children and the CPR level defined in the tree representation of Fig. 6. The modified inverted index referred to as the FSS is defined as follows:

```
typedef struct FSSelem *PFSS; // Pointer points to node FSSelem
struct FSSelem {
    int Children;          // Number of children of this node.
    int Descendants;       // Number of descendants.
    int Level;             // Level of this node in the tree representation.
    int Update-Count;      //Record of modification times.
    String Element;        // Element name of this Node.
        union PointerToChild {
            FSSelem *Child-Array-Pointer [Children]; // Child pointer.
            XML-Listings *DocPointer;    // pointer points to XML Listings.
        }
};
```

As shown in Table 4, the FSS can be used to define either an internal node or a leaf node. The difference between the two data structures is the setting of the active field.

Table 4: Illustration of the FSS indexes for two example nodes

	For internal node	For leaf node
Element =	"/CPs/CP_{L-1}"	"/CPs/CP_{L-1}/P1/ρ_0/B/~/~/~"
Children =	4	0
Descendants =	17	0
CPR_Level =	2	5
Update_Count =	0	0
Active field	Child-Array-Pointer [i] = a pointer pointing to P*i+1*	DocPointer = a pointer pointing to the disk files Doc1 and Doc2

The FSS with feature similarity provides a template-based hierarchical query service. This query service method can effectively reduce the searching complexity induced by the path element increment of CPR, compared to that of the NP and WP representations. Utilizing the one-to-one property of ρ_i , XML documents can be uniquely described with a feature vector (FV), defined as:

$$\mathrm{FV}_{DOC} = [\rho_0, \rho_1, \cdots, \rho_{N-1}], \rho_i \in \{0,1\}, \qquad (1)$$

where *N* denotes the number of CPEs. The element ρ_i = 1 implies that the document involves the *i*th labeled CPE. With the FV description, CPEs can be labeled with a hierarchical structure, as shown in Table 5. This labeling provides a template-based hierarchical query service. Let CPsT(*l*, *i*) be a query

template involving the CPEs of CP_{L-l} and P_i. A query template with $(l,i)=(1,1)$ can be defined as:

$$CPsT(1,1)=\begin{bmatrix} SW & 0 & 0 & 0 & 0 \\ \rho_{\#} & \rho_0 & \rho_1 & \rho_2 & \rho_3 \end{bmatrix}_{P_1}^{CP_{L-1}}$$

where SW denotes switch. Setting a field of SW to one indicates that the corresponding CPE is selected. For the example, a query template defined by:

$$CPsT(1,4)=\begin{bmatrix} SW & 1 & 0 & 1 & 1 \\ \rho_{\#} & \rho_{13} & \rho_{14} & \rho_{15} & \rho_{16} \end{bmatrix}_{P_4}^{CP_{L-1}}$$

will yield a response as:

ρ_{13} = */B/I/T/D* in Doc1

ρ_{15} = */B/I/A/L* in Doc1 and Doc2
ρ_{16} = */M/I/A/L* in Doc3.

Like the Region [22] and Dewey [26] methods, the CPE index can be easily updated with numerical labeling, as shown in Table 6. Updating the Dewey method is based on the extended Dewey labeling [25] [27, 28] which uses modular function to reserve even numbers for the insertion of new path elements. On the other hand, the updating of the CPE index only needs to increase the label in a template. Suppose that a new CPE /B/I/A/M will be added between /B/I/A/F and /B/I/A/L, as in Fig. 6. This updating will introduce four new CPEs: /CP_{L-1}/P4/"*/B/I/A/M*", /CP_{L-2}/P3/"*/~/I/A/M*", /CP_{L-3}/P2/"*/~/~/A/M*" and /CP_{L-4}/P1"*/~/~/~/M*" (the italic type in Fig. 6), and lead to some modifications: CPsT(1, 4), CPsT(2, 3), CPsT(3, 2), and CPsT(4, 1), as shown in Table 6, where only the content's order of CPsT(1, 4) needs to be rearranged (i.e., the new labels in parentheses).

Table 5: The template-based hierarchical labeling for the 34 CPEs of Fig. 6

CPs	P1		P2		P3		P4	
CP_{L-1}	Path	ρ #	Path	ρ #	Path	ρ #	Path	ρ #
	/B/~/~/~	ρ_0	/B/I/~/~	ρ_4	/B/I/T/~	ρ_8	/B/I/T/D	ρ_{13}
	/B/~/~	ρ_1	/B/I/~	ρ_5	/B/I/A/~	ρ_9	/B/I/A/F	ρ_{14}
	/M/~/~	ρ_2	/M/I/~	ρ_6	/B/I/T	ρ_{10}	/B/I/A/L	ρ_{15}
	/M/~/~/~	ρ_3	/M/I/~/~	ρ_7	/MI/T	ρ_{11}	/M/I/A/L	ρ_{16}
					/M/I/A/~	ρ_{12}		
CP_{L-2}	Path	ρ #	Path	ρ #	Path	ρ #		
	/~/I/~/~	ρ_{17}	/~/I/T/~	ρ_{19}	/~/I/T/D	ρ_{22}		
	/~/I/~	ρ_{18}	/~/I/A/~	ρ_{20}	/~/I/A/F	ρ_{23}		
			/~/I/T	ρ_{21}	/~/I/A/L	ρ_{24}		
CP_{L-3}	Path	ρ #	Path	ρ #				
	/~/~/T/~	ρ_{25}	/~/~/T/D	ρ_{28}				
	/~/~/A/~	ρ_{26}	/~/~/A/F	ρ_{29}				
	/~/~/T	ρ_{27}	/~/~/A/L	ρ_{30}				
CP_{L-4}	Path	ρ #						
	/~/~/~/D	ρ_{31}						
	/~/~/~/F	ρ_{32}						
	/~/~/~/L	ρ_{33}						

Table 6: A mapping for numerically labeling the 34 CPEs of Fig. 6

CPs	P1		P2		P3		P4	
CP_{L-1}	/CPs/CP_{L-1}/P1		/CPs/CP_{L-1}/P2		/CPs/CP_{L-1}/P3		/CPs/CP_{L-1}/P4	
	ρ #	Labels	ρ #	Labels	ρ #	Labels	ρ #	Labels
	ρ_0	1.1.1	ρ_4	1.2.1	ρ_8	1.3.1	ρ_{13}	1.4.1
	ρ_1	1.1.2	ρ_5	1.2.2	ρ_9	1.3.2	ρ_{14}	1.4.2
	ρ_2	1.1.3	ρ_6	1.2.3	ρ_{10}	1.3.3	*(ρ_{34})*	*(1.4.3)*
	ρ_3	1.1.4	ρ_7	1.2.4	ρ_{11}	1.3.4	ρ_{15}	1.4.3 *(1.4.4)*
					ρ_{12}	1.3.5	ρ_{16}	1.4.4 *(1.4.5)*
CP_{L-2}	/CPs/CP_{L-2}/P1		/CPs/CP_{L-2}/P2		/CPs/CP_{L-2}/P3			
	ρ #	Labels	ρ #	Labels	ρ #	Labels		
	ρ_{17}	2.1.1	ρ_{19}	2.2.1	ρ_{22}	2.3.1		
	ρ_{18}	2.1.2	ρ_{20}	2.2.2	ρ_{23}	2.3.2		
			ρ_{21}	2.2.3	ρ_{24}	2.3.3		
					(ρ_{35})	*(2.3.4)*		
CP_{L-3}	/CPs/CP_{L-3}/P1		/CPs/CP_{L-3}/P2					
	ρ #	Labels	ρ #	Labels				
	ρ_{25}	3.1.1	ρ_{28}	3.2.1				
	ρ_{26}	3.1.2	ρ_{29}	3.2.2				
	ρ_{27}	3.1.3	ρ_{30}	3.2.3				
			(ρ_{36})	*(3.2.4)*				
CP_{L-4}	/CPs/CP_{L-4}/P1							
	ρ #	Labels						
	ρ_{31}	4.1.1						
	ρ_{32}	4.1.2						
	ρ_{33}	4.1.3						
	(ρ_{37})	*(4.1.4)*						

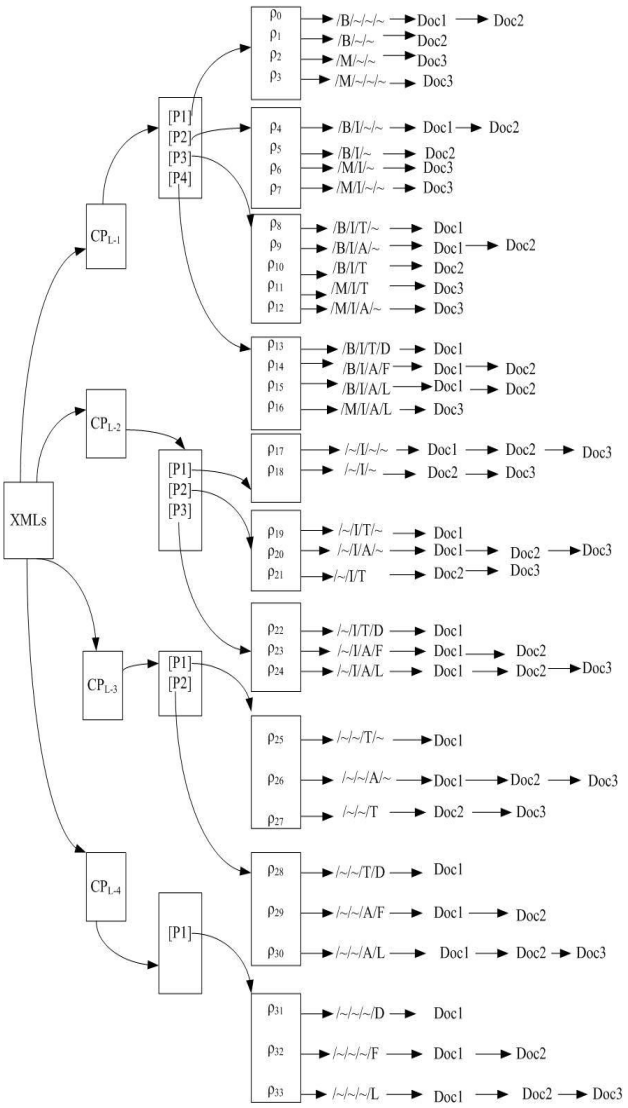


Figure 6: The index structure of the tree representation of Fig. 5. Italic is a new path inserted

The FSS with path length and level also allows the inference of semantic information. The path elements with AD relationships can easily be obtained from the CPEs with the path length field filled in Pi for $i \geq 3$, i.e., path length ≥ 3. For the example in Fig. 5, there are two kinds of AD relationship shown in Table 7, where A1 involves the path elements with one-generation AD, and A2 involves the path elements with two-generation AD. Note that these path elements are different from CPE, and are labeled as $\delta_0 \sim \delta_{11}$. SB and CN are relations among nodes, where these nodes have different descendants but have the same father and grandfather node, respectively. For SB, the father nodes can be found in levels CP_{L-l} for $1 \leq l \leq L-1$. Furthermore, the search of CN nodes is to verify whether their father nodes are inherent with a SB relationship. The hierarchical labeling templates of SB and CN relations are shown in Table 8. The tree structure index, including semantic information, is illustrated in Fig. 7, where SB and CN indexing requires fewer levels than the indexing of AD.

Table 7: The template-based hierarchical labeling for the AD path elements of Fig. 5

ADs	A1		A2	
AD_{L-1}	Path	δ #	Path	δ #
	/B/~/T/~	δ_0	/B/~/~/D	δ_5
	/B/~/A/~	δ_1	/B/~/~/F	δ_6
	/B/~/T	δ_2	/B/~/~/L	δ_7
	/M/~/T	δ_3	/M/~/~/L	δ_8
	/M/~/A/~	δ_4		
AD_{L-2}	Path	δ #		
	/~/I/~/D	δ_9		
	/~/I/~/F	δ_{10}		
	/~/I/~/L	δ_{11}		

Table 8: Sibling (SB) and Cousin (CN) relations for Fig. 6

SBs			CNs		
SB_{L-2}	Path	ρ #	CN_{L-3}	Path	ρ #
	/~/I/~/~	ρ_{17}		/~/~/T/~	ρ_{25}
	/~/I/~/~	ρ_{18}		/~/~/A/~	ρ_{26}
SB_{L-3}	Path	ρ #		/~/~/T	ρ_{27}
	/~/~/A/~	ρ_{26}	CN_{L-4}	Path	ρ #
	/~/~/T	ρ_{27}		/~/~/~/D	ρ_{31}
SB_{L-4}	Path	ρ #		/~/~/~/F	ρ_{32}
	/~/~/~/F	ρ_{32}		/~/~/~/L	ρ_{33}
	/~/~/~/L	ρ_{33}			

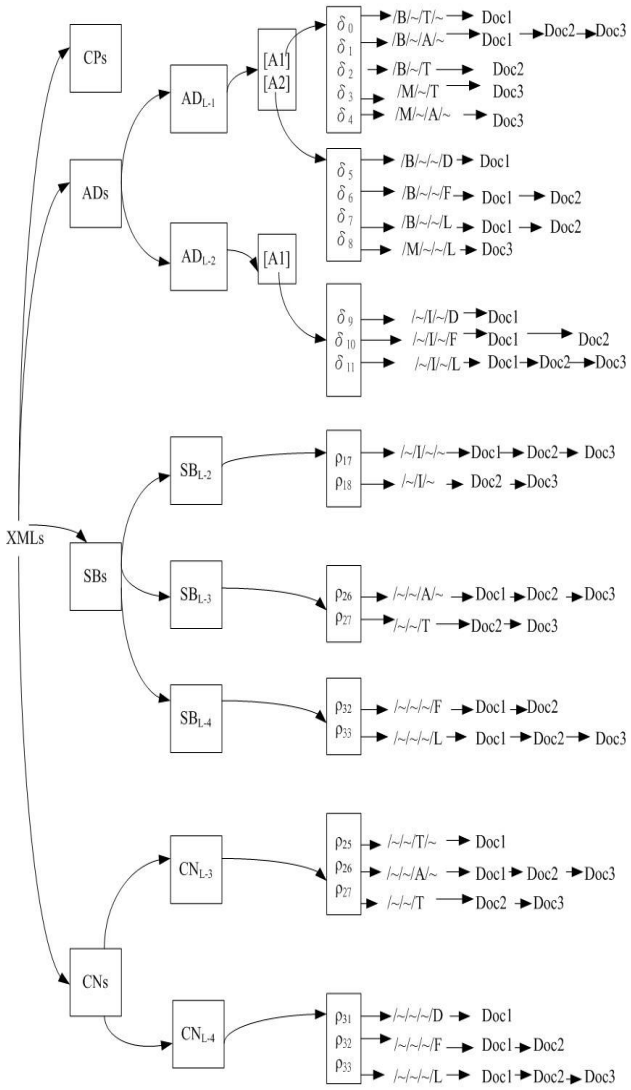


Figure 7: The index structure of the ADs, SBs, and CNs of Fig. 6

4 EXPERIMENTAL RESULTS

For the data service efficiency analysis of CPR, an experiment using the simple dataset of Fig. 5 was performed. In this dataset, WP and NP have 6 and 10 feature elements respectively. For CPR, the feature elements of CPE and AD relation are 34 and 13, respectively. Some queries shown in Table 9 are designed for the simulation of versatile client requests. These queries can be categorized into CPE (TPQ_1~TPQ_8), AD (TPQ_9~TPQ_A), and SB&CN (TPQ_B~TPQ_D) groups, where TPQ_1~TPQ_5 belong to WP and NP types. TPQ_D is special due to the distinct I nodes. Decoded with the query parser [23], these statements can be translated into compound tree-pattern queries. Two commonly used indices: searching complexity and accuracy, are applied for performance evaluation. The searching complexity (SC) is defined with the total checking times required for matching all of the query paths. Here, we suppose that all of the path elements (in dataset) fitting query conditions should be checked in each query path matching. For TPQ_1, there are four query paths with level=1 and path length=4. The level and path length determine the selection of the query template: $CPsT(1,4)$, where four path elements: $\rho_{13} \sim \rho_{16}$, satisfy the conditions. Considering exhaustive matching, each query path should be matched four times. Thus the query service of CPR requires a complexity of $SC = 4 \times 4 = 16$, and the SW fields of $\rho_{13} \sim \rho_{16}$ will be set to 1:

$$CPsT(1,4) = \begin{bmatrix} SW & 1 & 1 & 1 & 1 \\ \rho_{\#} & \rho_{13} & \rho_{14} & \rho_{15} & \rho_{16} \end{bmatrix}_{P_4}^{CP_{L-1}}.$$

The complexities required for serving TPQ_2~TPQ_9 are evaluated in Table 10, where the symbol '-' denotes that this representation method cannot serve the query. For TPQ_A, there are three 1-level query paths involving two one-generation AD and one two-generation AD. The level and AD relations determine the selection of two query templates: $ADsT(1,1)$ and $ADsT(1,2)$, where the former has five elements ($\delta_0 \sim \delta_4$), and the latter has four elements ($\delta_5 \sim \delta_8$). Also considering exhaustive search, the SC of TPQ_A can be found as SC = 5 * 2 + 4 = 14. The query templates are set by:

$$ADsT(1,1) = \begin{bmatrix} SW & 0 & 1 & 1 & 0 & 0 \\ \delta_{\#} & \delta_0 & \delta_1 & \delta_2 & \delta_3 & \delta_4 \end{bmatrix}_{A_1}^{AD_{L-1}},$$

$$ADsT(1,2) = \begin{bmatrix} SW & 1 & 0 & 0 & 0 \\ \delta_{\#} & \delta_5 & \delta_6 & \delta_7 & \delta_8 \end{bmatrix}_{A_2}^{AD_{L-1}}.$$

For $TPQ_{B\sim D}$, the level and semantic relations will determine the selection of the three query templates: $SBsT(3)$, $CNsT(3)$, and $SBsT(2)$. By using exhaustive search, the SC of the three queries can be found as SC = 4(2*2), 9(3*3), and 4(2*2) respectively. The query templates are set by:

$$SBsT(3)=\begin{bmatrix} SW & 1 & 1 \\ \rho_{\#} & \rho_{26} & \rho_{27} \end{bmatrix}^{SB_{L-3}},$$

$$CNsT(3)=\begin{bmatrix} SW & 1 & 1 & 1 \\ \rho_{\#} & \rho_{25} & \rho_{26} & \rho_{27} \end{bmatrix}^{CN_{L-3}}, \text{ and}$$

$$SBsT(2)=\begin{bmatrix} SW & 1 & 1 \\ \rho_{\#} & \rho_{17} & \rho_{18} \end{bmatrix}^{SB_{L-2}}.$$

Searching accuracy (SA) is defined with two bi-levels: Success and Fail, indicating whether or not the document can be found. With WP and NP element queries, the documents satisfying the conditions of TPQ_1~TPQ_5 can easily be retrieved for the WP and NP approaches respectively. For queries TPQ_6~TPQ_8 that request sub-paths starting from different levels, neither NP nor WP can handle these queries due to a lack of level information. The experiment clearly shows that NP and WP are subsets of the CPR. Nevertheless, with hierarchical template search, the increased feature elements do not reduce the searching efficiency of CPR at all. With the semantic relation inference capability, CPR can also easily serve the queries with inherent AD, SB and CN relationships. The SC of TPQ_9 and TPQ_D are shown in Table 10. However, neither WP nor NP can handle these queries due to a lack of level and path length information.

Table 9: Some queries for the simulation of versatile client requests

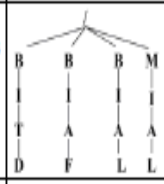
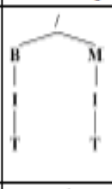
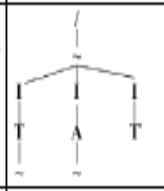
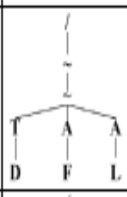
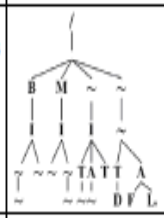
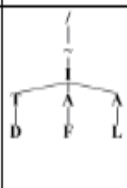
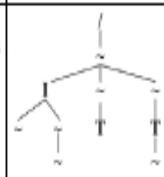
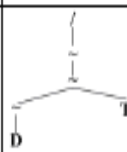
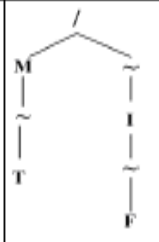
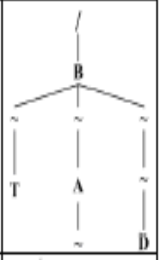
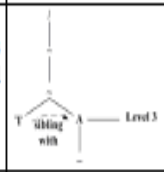
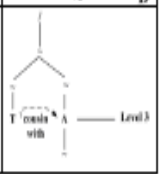
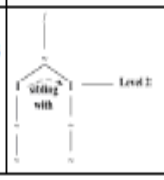
Query	Description	Tree Pattern Query	Query	Description	Tree Pattern Query
TPQ_1	Find documents with four 4-length whole paths.		TPQ_2	Find documents with two 3-length whole paths.	
TPQ_3	Find documents with three level 2 NPs.		TPQ_4	Find documents with three level 3 NPs.	
TPQ_5	Find documents with the NPs in all levels.		TPQ_6	Find documents with three 3-length path elements in level 2.	
TPQ_7	Find documents with nodes I and T in level 2 and 3.		TPQ_8	Find documents with leaf nodes T and D.	
TPQ_9	Find documents with M being ancestor of T and I being ancestor of F.		TPQ_A	Find documents with B being ancestor of T, A, and D.	
TPQ_B	Find documents with T and A being sibling on level 3.		TPQ_C	Find documents with T and A being cousin on level 3.	
TPQ_D	Find documents with I and I being sibling on level 2.				

Table 10: A comparison of the XML data service performances of WP, NP and CPR approaches for the queries given in Table 9

Tree Pattern Queries	SC			SA		
	WP	NP	CPR	WP	NP	CPR
TPQ_1	16	-	16	S	F	S
TPQ_2	10	-	10	S	F	S
TPQ_3	-	9	9	F	S	S
TPQ_4	-	9	9	F	S	S
TPQ_5	-	34	34	F	S	S
TPQ_6	-	-	9	F	F	S
TPQ_7	-	-	10	F	F	S
TPQ_8	-	-	6	F	F	S
TPQ_9	-	-	8	F	F	S
TPQ_A	-	-	14	F	F	S
TPQ_B	-	-	4	F	F	S
TPQ_C	-	-	9	F	F	S
TPQ_D	-	-	4	F	F	S

5 Conclusion

In this paper, a new XML data representation called CPR is presented as a means of providing an efficient and versatile query service. CPR uses complete path elements as XML data description features. In association with a modified inverted index, the CPR approach can preserve both structure and semantic information, as well as provide a template-based indexing for fast XML data search. Performance evaluation results show that the CPR can be an efficient kernel for XML data service.

References:

[1] World Wide Web Consortium. The document object model. http://www.w3.org/DOM/

[2] Theodore Dalamagas, Tao Cheng, Klaas Jan Winkel, Timos Sellis, A Methodology for Clustering XML Documents by Structure, *Information Systems*, 31(3): 187-228, 2006.

[3] T. Dalamagas et al., "Clustering XML Documents using Structural Summaries", *EDBT Work-shop on Clustering Information over the Web (ClustWeb04)*, Heraklion, Greece, 2004.

[4] A. Nierman, H. V. Jagadish, "Evaluating Structural Similarity in XML Documents", *Fifth International Workshop on the Web and Databases (WebDB 2002)*.

[5] S. Flesca et al., "Fast Detection of XML Structural Similarity", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 17, No. 2, February 2004. pp 160-175.

[6] W. Lian et al., "An Efficient and Scalable Algorithm for Clustering XML Documents by Structure", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 16, No. 1, January 2004. pp 82-96.

[7] Michal Kozielski ,"Improving the Results and Performance of Clustering Bit-encoded XML Documents", Sixth *IEEE International Conference on Data Mining* - Workshops (ICDMW'06).

[8] Jin-sha Yuan, Xin-ye Li, Li-na Ma, "An Improved XML Document Clustering Using Path Feature," fskd, vol. 2, pp.400-404, 2008 Fifth *International Conference on Fuzzy Systems and Knowledge Discovery,* 2008.

[9] Ho-pong Leung, Fu-lai Chung, Chan, S.C.F, Luk, R., "XML Document Clustering Using Common Xpath", *Proceedings of the International Workshop on Challenges in Web Information Retrieval and Integration*, Tokyo ,pp. 91-96, April 2005.

[10] J. A. Termier, M-C. Rousset, M. Sebag, "treefinder: a first step towards XML data mining", *Proceedings of IEEE International Conference on Data Mining*, Maebashi, pp. 450-457, December 2002.

[11] Jianwu Yang, William K. Cheung, Xiaoou Chen, "Learning the Kernel Matrix for XML Document Clustering", *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05)*, Hong Kong, pp. 353-358, April 2005.

[12] Jianghui Liu, Wang, J.T.L, Hsu.W, Herbert,K.G., "XML Clustering by Principal Component Analysis", *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'04)*, Boca Raton, pp.658-662, November 2004.

[13] Mustafa H. Qureshi, Kozielski, M. H. Samadzadeh, "Determining the Complexity of XML Documents", Proceedings of the *International Conference on Information Technology: Coding and Computing* (ITCC'05) - Volume II - Volume 02, Pages: 416 – 421.

[14] Jung won lee, kiho lee, won kim, "Preparation For Semantic-Based XML Mining", The 2001 *IEEE International Conference on Data Mining*, San Jose, pp.345-352, November 2001.

[15] Xin-Ye Li, "Using Clustering Technology to Improve XML Semantic Search", *Proceedings of the Seventh International Conference on Machine Learning and Cybernetics:* Volume: 5, Page(s):2635-2639, July 2008.

[16] S. Zhang, J. T. L. Wang, and K. G. Herbert. , Xml query by example. *International Journal of Computational Intelligence and Applications*, 2(3):329–337, 2002.Jonathan

Robie and Red Hat**,** *IEEE Internet Computing***,** **"**XML Processing and Data Integration with XQuery", August 2007, (vol. 11 no. 4) pp. 62-67.

[17] Jonathan Robie and Red Hat**,** *IEEE Internet Computing***,** **"**XML Processing and Data Integration with XQuery", August 2007, (vol. 11 no. 4) pp. 62-67.

[18] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: optimal XML pattern matching, in: *Proceedings of the SIGMOD Conference*, 2002, pp. 310–321.

[19] Qiankun Zhao, Ling Chen, Sourav S. Bhowmick, Sanjay Kumar Madria: XML structural delta mining: Issues and challenges. *Data and Knowledge Engineering* 59(3): 627-651 (2006).

[20] S. Chen, H.G. Li., J. Tatemura, W.P. Hsiung, D. Agrawal, K.S. Candan, Twig^{2}Stack: bottom-up processing of generalized tree-pattern queries over XML documents, in: *Proceedings of the VLDB Conference*, 2006, pp. 283–294.

[21] L. Qin, J. Xu Yu, B. Ding, TwigList: make twig pattern matching fast, in *Proceedings of the DASFAA Conference*, 2007, pp. 850–862.

[22] J. Lu, T.W. Ling, C.Y. Chan, T. Chen, From region encoding to extended Dewey: on efficient processing of XML twig pattern matching, in: *Proceedings of the VLDB Conference*, 2005, pp. 193–204.

[23] Sayyed K. Izadi, Theo Härder and Mostafa S. Haghjo, S^3: Evaluation of tree-pattern XML queries supported by structural summaries, *Data & Knowledge Engineering*, Vol. 68, Issue 1, pp. 126-145, Jan. 2009.

[24] Barbara Catania and Anna Maddalena, **"**XML Document Indexes: A Classification", *IEEE Internet Computing*, October 2005, (vol. 9 no. 5) pp. 64-71.

[25] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury: ORDPATHs: Insert-Friendly XML Node Labels. *SIGMOD* (2004) 903-908.

[26] S. Tatarinov, K.S. Viglas., J. Beyer, E. Shanmugasun-daram, J. Shekita, C. Zhang, Storing and querying ordered XML using a relational database system. *SIGMOD* (2002) 204-215.

[27] T. Harder, M.P. Haustein, C. Mathis, M. Wagner: Node labeling schemes for dynamic XML documents reconsidered, *Data and Knowledge Engineering* 60 (1) (2007) 126–149.

[28] M.P. Haustein, T. Harder: An efficient infrastructure for native transactional XML processing, *Data and Knowledge Engineering* 61 (3) (2007) 500–523.