Published by :
http://www.ijert.org

The Compression Method of Complete Path Representation (CPR) and Postings for Efficient Retrieval XML Documents

Hsu-Kuang Chang
Department of Information Engineering
I-Shou University
Kaohsiung, Taiwan

***Abstract*—The CPR dictionary and the inverted index presented as the central data structures in XML information retrieval [1]. It can be seen that the index structure could be very large since the CPR for each tree can be generated from many path combinations. For a larger and deeper tree, the index structure could grow exponential. Therefore, the proposed method of representing XML document with complete paths practical and scalable for use in web environment in which XML structure could be complicated. In this paper, we employ a number of compression techniques for CPR dictionary and inverted index that are important for efficient XML IR systems. There are two subtle benefits of compression. The first is increased use of caching. Search systems use some parts of the dictionary and the index much more than others. For example, if we cache the XML postings list of a frequently used query paths p, then the computations necessary for responding to the paths query p can be entirely done in memory. With compression, we can fit a lot more information into main memory. Instead of having to expand a disk seek when processing a query with p, we instead access its XML postings in memory and decompress it. The second more subtle advantage of compression is faster transfer of data from disk to memory. For instance, we can reduce input/output (I/O) time by loading much smaller compressed XML postings, even when you add on the cost of decompression. So, in most cases, the retrieval system runs faster on compressed XML postings than on uncompressed postings lists.**

***Keywords*—CPR, XML, Posting, dictionary**

I. INTRODUCTION

The amount of data in our world has been exploding exponentially as a result of the rapid development of the Internet and messaging-related technology, so a variety of different data formats has resulted. Unfortunately, the different data formats cause significant difficulties in the exchange of information destined for data storage and file formats used commercially or for business purposes. Therefore W3C [1] defines XML as a standard information description language widely used in the exchange of information and data storage. The increased amounts of data transfer have also increased the demand for efficacious XML data query search processing. Therefore, the use of the tree model for XML files has been the core query search process to satisfy user requirements. In order to describing the XML query language, first proposed by the Xpath [2] and XQuery [3], the process is based on path expressions. In order to process XML queries, all portions matching the query tree pattern in the XML document must be found. These patterns include the most important query axes: parent child (PC, / for short) and ancestor descendant (AD, // for short). Some methods, related to [4] and [5], match all query results in the contents of the XML document, but this is a time-consuming task, especially considering the increasingly large amount of data. Therefore, research is required to improve the efficiency of path expressions [6-16]. Path approaches use sub-paths as their feature, and represent each XML datum as a binary vector. An element in the binary vector denotes whether the datum involves a corresponding feature, where such features can be defined as tree nodes [7], two-node sub-paths (i.e. node-pair (NP)) [8-9], or whole paths (WP) [10-11]. To improve search efficiency, several path representation modifications have been proposed. Yang et al. [12] used the content instead of the leaf node for node representation. Liu et al. [13] presented a hybrid definition that combines NP and WP for XML data description.
The rest of the paper is organized as follows: first, it examines XML CPR representation. Second, it presents compression method for CPR paths and XML postings. Next, it shows the experiment results of handling compressing posting using variant approaches. Conclusions are drawn in the final section.

II. XML DATA REPRESENTATION USING COMPLETE PATH ELEMENTS

An example of level definition is shown in Figure 1, where four CPE sets: CP_{L-1}, CP_{L-2}, CP_{L-3}, and CP_{L-4}, can be defined. The elements of the four CPE sets are shown in Table 1. Traditional WP and NP representations, with their lack of linking information, cannot serve such queries as (/B/*/*/*/*) and (/*/I/A/L) where * means “don’t care” about the node name. For efficient query service, CPR describes XML data with the CPEs of all SSTs. The CPE 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-i} denote a set of CPEs starting from the *i*th level, where the root level is defined as one and is increased toward the leaves.
Essentially, this algorithm is an exhaustive search that is guaranteed to find all of the branches of a SST. The CPR for the description of SSTs can be defined as:

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

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 14 Issue 06, June - 2025

where ∪ denotes union operation and L is the maximum level number. Considering a database comprised of the three XML data shown in Figure 1, the CPR can be found in Table 1. In Figure 1, 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 sub-path elements /M/I/* and /M/I/*/* in the CP_{L-1}.

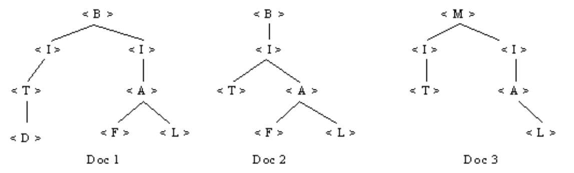

Figure 1. The SSTs of three XML documents where <x> denotes the node x.

Table 1. The CPR for the description of the three XML data shown in Fig. 1.

CPs	Complete path elements (CPEs)																
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/*/*	/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														

III. INDEXING COMPLETE PATH ELEMENTS

A CPE with the tree characteristic is a high dimensional feature. Traditional B-tree indexing 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 1 can be represented with a tree structure, as shown in Figure 2, where Pi denotes a CPE subset with path length equal to i. The CPEs in Figure 2 are inherent with the hierarchical information involving path length (Pi) 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 efficiently indexing the elements of NP and WP, but cannot provide hierarchical information. To facilitate the inference of semantic information, the inverted index structure with additional fields is applied for CPE indexing.

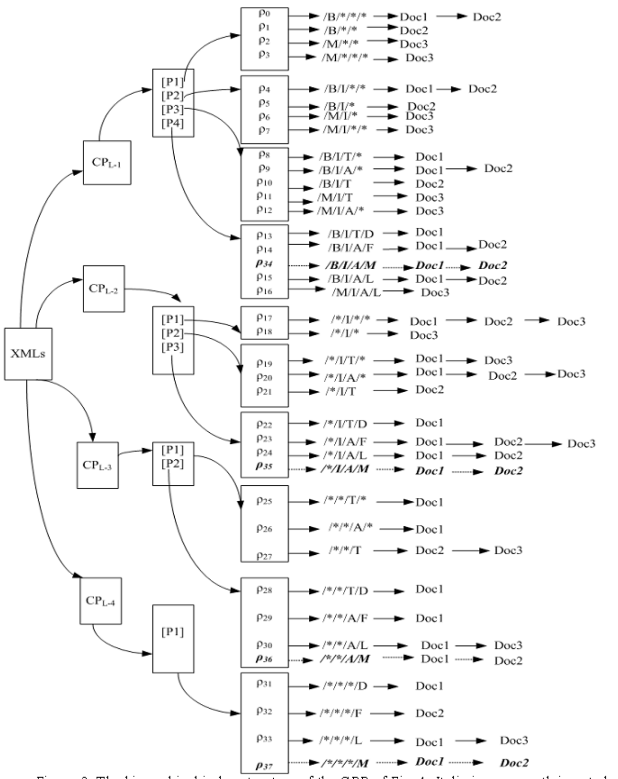

Figure 2. The hierarchical index structure of the CPR of Fig. 1. Italic is a new path inserted.

The path elements with AD relationships can be easily obtained from the CPEs with the path length field filled in Pi for $i \geq 3$, i.e., path length ≥ 3. For the example of Figure 2, there are two kinds of AD relationship shown in Table 2, 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 with $\delta_0 \sim \delta_{11}$. 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 3. The tree structure index, including semantic information, is illustrated in Figure 3, where SB and CN indexing requires fewer levels than the indexing of AD.

Table 2. The template-based hierarchical labeling for the AD path elements of Figure 2

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 3. The hierarchically template-based index and label of the SB and CN nodes inferred from 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}			

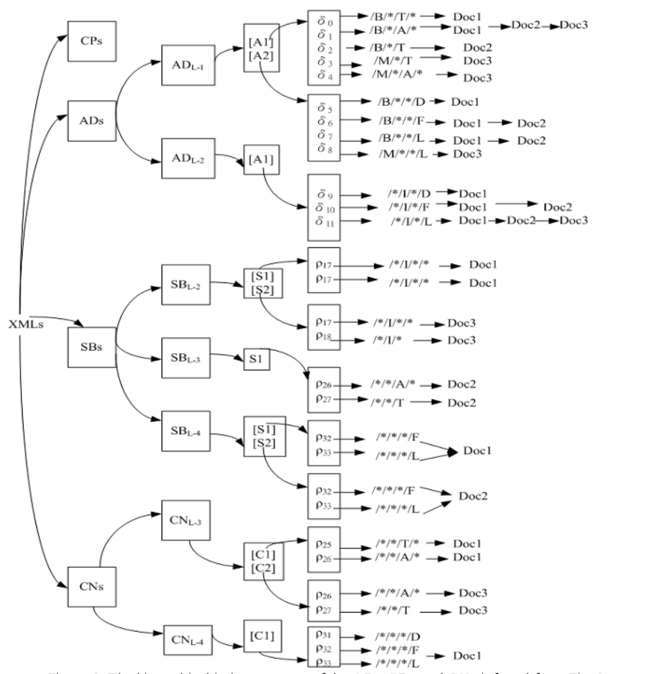

Figure 3. The hierarchical index structure of the ADs, SBs, and CNs inferred from Fig. 2.

IV. COMPESSION METHODS OF CPR PATHS AND XML POSTINGS

This section presents a series of CPR dictionary structures that achieve increasingly higher compression ratios. The techniques of compression in the CPR dictionary are presented in fixed-width entries, CPR dictionary as string, block storage with *k*, block & CPR coding methods, and CPR Huffman Coding (CPR HC).

4.1 Compession Methods Of Cpr Paths

CPR dictionary as fixed-width entries

In Figure 2 and Figure 3 for CPR dictionary, we need *M* * (20+4+4) = 63*28=1764 bytes (*B*), where *M* is paths of CPEs, ADs, SBs, and CNs for storing the CPR dictionary in this scheme as shown in the Table 4. Using fixed-width entries for the CPR dictionary is clearly wasteful.

Table 4. CPR dictionary compression for XML datasets of Figure 2 and Figure 3.

Data structure	CPEs	ADs	SBs	CNs	Size in bytes (*B*)
CPR Fixed-Width	34*(20+4+4)=952 *B*	12*(20+4+4)=336 *B*	10*(20+4+4)=280 *B*	7*(20+4+4)=196 *B*	1764 *B*
CPR Path String	34*(4+4+3+8)= 646 *B*	12*(4+4+3+8)=228 *B*	10*(4+4+3+8)=190 *B*	7*(4+4+3+8)=133*B*	1197 *B*
CPR Block Pointer ($k=4$)	646 - 43′ = 603 *B*	228 – 15″ = 213 *B*	190 – 13 = 177 *B*	133 – 9 = 124 *B*	1117 *B*
CPR Block & Coding	$34*(4+4+8)+(\sum_{i=1}^{L} i)*3=544+30=574\ B$	$12*(4+4+8)+(\sum_{i=1}^{L-2} i)*3=192+9=201\ B$	$10*(4+4+8)+(\sum_{i=2}^{L} i)*3=160+9=169\ B$	$7*(4+4+8)+(\sum_{i=3}^{L} i)*3=112+6=118\ B$	1062 *B*
CPR Static Huffman Coding (SHC)	331/8+34*(4+4) = 313 *B*	119/8 +12*(4+4) =111 *B*	40/8+6*(4+4)=47 B	44/8+6*(4+4)=48 B	519 *B*

In Figures 2~3, 34 CPEs, 12 ADs, 10 SBs, and 7 CNs paths are derived. ′ Eliminate *k*-1 path pointers but additional *k* bytes for storing the length of each path, saves (*k*-1)*3=9 bytes, and additional 4 bytes for length that is 9 – 4 = 5 saved, $34\times\frac{1}{4}\times 5=43\ B$ reduced.

″ Similarly for ADs, $12\times\frac{1}{4}\times 5=15B$ saved.

CPR Block storage with *k*

We can further compress the CPR dictionary by grouping paths in the string with blocks of size *k* and keeping a path pointer only for the first path of each block. We store the length of the path in the string as an additional byte at the beginning of the path. We thus eliminate *k*-1 path pointers, but need additional *k* bytes for storing the length of each path. For *k* = 4, we save (*k*-1)*3 = 9 bytes for path pointer, but need an additional *k* = 4 bytes for path length. So the total space requirements for the CPEs dictionary of CPR are reduced by 5 bytes per four-path block, or total of $34\times\frac{1}{4}\times 5=42.5$ for CEPs and $12\times\frac{1}{4}\times 5=15B$ for ADs respectively, bringing us down to 1117 *bytes* as shown in the Table 4.

Block & CPR coding

We presented new method, the blocking & CPR coding with CPEs dictionary, totally needs block pointers, $\sum_{i=1}^{L} i=1+2+3+4=10$, which can be denoted as $\sum_{i=1}^{L}\sum_{j=1}^{L-i+1}\rho_{i,j}$.

The CPEs dictionary of the size in can be denoted as $34*(4+4+8)+\left(\sum_{i=1}^{L} i\right)*3=544+10*3=574\ B$, which 4 bytes for frequency and posting pointer, 8 bytes for average path length, and 3 bytes for block pointer. Similarly for ADs dictionary, with blocking & CPR coding, it totally needs block pointers $\sum_{i=1}^{L-2} i=1+2=3$, which can be denoted as $\sum_{i=1}^{L-2}\sum_{j=1}^{L-i-1}\delta_{i,j}$. The CPR for this scheme of need size is down to 1062 *B* as shown in the Table 4.

CPR Huffman Coding

We presented modified Huffman Coding method for CPEs and ADs paths in Figure 2 and Figure 3. The Huffman Code algorithm for CPR is presented in the Table 5. First, the frequency of the path elements can be derived as the Table 6, for example, according to the shown frequency, the path element F can be compressed Huffman code as <u>11001</u> (5 bits), and I as <u>101</u> (3 bits). Secondly, the complete path elements (CPEs) in different length of each level can be derived in the Table 7. For example, the Huffman Coding (HC) of /B/I/T/D (path of length-4 in level-1: CP_{L-1}/P4) can be represented as the <u>1111</u> <u>101</u> <u>1110</u> <u>11000</u>. Thirdly, the ancestor-descents (ADs) path in different generation of each level can be derived in the Table 8. For example, the Huffman Coding (HC) of /B/~ /~ /D (AD path of 2- generations in level-1: AD_{L-1}/A2) can be represented as the <u>1111</u> <u>0</u> <u>0</u> <u>11000</u>. Fourthly, the compression ratio (CR) of complete paths representation (CPR) can be calculated as CR = Bytes of (CPEs+ADs) * 8 bits / Bits of Huffman Coding (CPEs+ADs), CR = (62+62+50) * 8 bits / (184+147+119) bits = 3.09 as shown in the Table 9. That is, for example, the path /B/~/~/~, occupies

Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 14 Issue 06, June - 2025

32 bits (4-byte * 8 bis) while the Huffman coding is 7 bits only (1111 0 0 0).

Table 5. Algorithm Huffman Code for CPR - Creating Tree

```
Huffman Code (CPR)
// Huffman Code encoding for CPR
// CPR : Complete Path Representation.
1. Count each symbol (element) frequency of the CPR
       Put each symbol (element) into the list in ascending frequency
2. Place each symbol in leaf
       Weight of leaf = symbol frequency
3. Repeat
4.     Select two trees L and R (initially leafs)
           Such that L, R has lowest frequencies in tree
5.     Create new (internal) node
           L ← Left child
           R ← Right child
           New frequency ← frequency(L) + frequency(R)
6. Until all nodes merge into one tree
```

Table 6. Path elements of Frequency CPR

CPR			
Symbol	Present	Frequency	Huffman Encoding
F	6	6/173	11001
D	6	6/173	11000
L	8	8/173	1000
M	10	10/173	1001
T	13	13/173	1110
A	13	13/173	1101
B	16	16/173	1111
I	24	24/173	101
~	77	77/173	0

Table 7. The Huffman Coding of the complete path elements (CPEs).

CPEs	P1	P2	P3	P4
CP_{L-1}	/B/~/~/~ 1111 0 0 0	/B/I/~/~ 1111 101 0 0	/B/I/T/~ 1111 101 1110 0	/B/I/T/D 1111 101 1110 11000
	/B/~/~ 1111 0 0	/B/I/~ 1111 101 0	/B/I/A/~ 1111 101 1101 0	/B/I/A/F 1111 101 1011 11001
	/M/~/~ 1001 0 0	/M/I/~ 1001 101 0	/B/I/T 1111 101 1110	/B/I/A/L 1111 101 1101 1000
	/M/~/~/~ 1001 0 0 0	/M/I/~/~ 1001 101 0 0	/M/I/T 1001 101 1110	/M/I/A/L 1001 101 1101 1000
			/M/I/A/~ 1001 101 1101 0	
CP_{L-2}	/~/I/~/~ 0 101 0 0	/~/I/T/~ 0 101 1110 0	/~/I/T/D 0 101 1110 11000	
	/~/I/~ 0 101 0	/~/I/A/~ 0 101 1101 0	/~/I/A/F 0 101 1101 11001	
		/~/I/T 0 101 1010	/~/I/A/L 0 101 1101 1000	
CP_{L-3}	/~/~/T/~ 0 0 1110 0	/~/~/T/D 0 0 1110 11000		
	/~/~/A/~ 0 0 1101 0	/~/~/A/F 0 0 1101 11001		
	/~/~/T 0 0 1110	/~/~/A/L 0 0 1011 1000		
CP_{L-4}	/~/~/~/D 0 0 0 11000			
	/~/~/~/F 0 0 0 11001			
	/~/~/~/L 0 0 0 1000			

Table 8. The Huffman Coding of the ancestor-descent (ADs).

ADs	A1	A2
AD_{L-1}	/B/~/T/~ : 1111 0 1110 0	/B/~/~/D : 1111 0 0 11000
	/B/~/A/~ : 1111 0 1101 0	/B/~/~/F : 1111 0 0 11001
	/B/~/T : 1111 0 1110	/B/~/~/L : 1111 0 0 1000
	/M/~/T : 1001 0 1110	/M/~/~/L : 1001 0 0 1000
	/M/~/A/~ : 1001 0 1101 0	
AD_{L-2}	/~/I/~/D : 0 101 0 11000	
	/~/I/~/F : 0 101 0 11001	
	/~/I/~/L : 0 101 0 1000	

Table 9. The compression ratio (CR) in Huffman Coding for CPR.

CPEs	Huffman Coding	CPEs	Huffman Coding	ADs	Huffman Coding	
/B/~/~/~	100 0 0 0	/~/I/~	0 110 0	/B/~/T/~	100 0 1010 0	
/B/~/~	100 0 0	/~/I/T/~	0 110 1010 0	/B/~/A/~	100 0 1011 0	
/M/~/~	11111 0 0	/~/I/A/~	0 110 1011 0	/B/~/T	100 0 1010	
/M/~/~/~	11111 0 0 0	/~/I/T	0 110 1010	/M/~/T	11111 0 1010	
/B/I/~/~	100 110 0 0	/~/I/T/D	0 110 1010 11101	/M/~/A/~	11111 0 1011 0	
/B/I/~	100 110 0	/~/I/A/F	0 110 1011 11100	/B/~/~/D	100 0 0 11101	
/M/I/~	11111 110 0	/~/I/A/L	0 110 1011 11110	/B/~/~/F	100 0 0 11100	
/M/I/~/~	11111 110 0 0	/~/~/T/~	0 0 1010 0	/B/~/~/L	100 0 0 11110	
/B/I/T/~	100 110 1010 0	/~/~/A/~	0 0 1011 0	/M/~/~L	11111 0 0 11101	
/B/I/A/~	100 110 1011 0	/~/~/T	0 0 1010	/~/I/~/D	0 110 0 11101	
/B/I/T	100 110 1010	/~/~/T/D	0 0 1010 11101	/~/I/~/F	0 110 0 11100	
/M/I/T	11111 110 1010	/~/~/A/F	0 0 1011 11100	/~/I/~/L	0 110 0 11110	
/M/I/A/~	11111 110 1011 0	/~/~/A/L	0 0 1011 11110			
/B/I/T/D	100 110 1010 11101	/~/~/~/D	0 0 0 11101			
/B/I/A/F	100 110 1011 11100	/~/~/~/F	0 0 0 11100			
/B/I/A/L	100 110 1011 11110	/~/~/~/L	0 0 0 11110			
/M/I/A/L	11111 110 1011 11110					
/~/I/~/~	0 110 0 0					
62 bytes	184 bits	62 bytes	147 bits	50 bytes	119 bits	CR=3.09

4.2 Compession Methods Of Xml Document Postings

bits 20-30 compression

Assuming we have 800,000 XML documents, 200 paths per document, six characters per path, and 100,000,000 XML postings where we define a posting as a docID in a postings list. XML document identifiers are $\log_2$ 800,000 ~= 20 bits long. Thus, the size of the collection is about 800,000*200*6 bytes=960 MB and the size of the uncompressed XML postings file is 100,000,000*32/8 = 400 MB (32-bit word) and 100,000,000*20/8 =250 MB (20-bit word) respectively as shown in the Table 10.

Table 10. Document postings compression for XML

data structure	Size in mega bytes (*MB*)
Size of collection	$8\times10^5\times200\times6$ bytes = 960 *MB*
Size of uncompressed posting file (32-bit word)	$1\times10^8\times32/8=400$ *MB*
Size of uncompressed posting file (20-bit word)	$1\times10^8\times20/8=250$ *MB*

Assumes 8×10^5 documents, 200 paths/per document, 6 chars/per path, and 1×10^8 XML postings.

To encode small numbers in less space than large numbers, we look at two types of methods: bytewise compression and bitwise compression. As the names suggest, these methods attempt to encode gaps with the minimum number of bytes and bits, respectively.

Variable byte (*VB*) encoding

Bytewise variable byte (*VB*) encoding uses an integral number of bytes to encode a gap. The last 7 bits of a byte are "payload" and encode part of the gap. The first bit of the byte is a *continuation bit*. It is set to 1 for the last byte of the encoded gap and to 0 otherwise. To decode a variable byte code, we read a sequence of bytes with *continuation bit* 0

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 14 Issue 06, June - 2025

terminated by a byte with *continuation bit* 1. We then extract and concatenate the 7-bit parts. Note the *VB* encoding 60% reduced compared with the 20-bits uncompressed can be denoted in the Table 12.

To devise a more efficient representation of the XML postings, the proposed methods that uses fewer than 20 bits per XML document, we observe that the XML postings for frequent paths are close together. Imagine going through the documents of a collection one by one and looking for a frequent path like */B/I/T/D*. We will find a document containing */B/I/T/D*, then we skip a few documents that do not contain it, then there is again a document with the path and so on (see Table 11). The key idea is that the gaps between XML postings are short, requiring a lot less space than 20 bits to store. In fact, gaps for the most frequent paths such as */B/~/~/~* are mostly equal to 1. But the gaps for a rare path that occurs only once or twice in a collection (e.g., */B/~/A/~* in Table 11) have the same order of magnitude as the docIDs and need 20 bits. For an economical representation of this distribution of gaps, we need a *variable encoding* method that uses fewer bits for short gaps.

Table 11. Encoding gaps instead of document IDs.

/B/~/~/~	docIDs	…	283042		283043		283044		283044	…
	gaps	…		1		1		1		…
/B/I/T/D	docIDs	…	283047		283154		283159		283202	…
	gaps	…		107		5		43		…
/B/~/A/~	docIDs	25200		50010						
	gaps	25200	24810							

Patched Frame-Of-Reference Delta (PForDelta)

The PForDelta use the compressing structures including the header, code, and exception which contain 32-bits for header, 8-bits for code (172 xml postings), and 4-byte for exception respectively. The CPEs for PForDelta is derived as 50*8-bits =400 bits/8=50 Bytes, ADs 19*8=152/8=19 Bytes. The total size bytes for PForDelta be 32B+50B+19B+8B+8B=88Bytes. This strategy for PForDelta is space saved and good performance of efficient retrieval. The PForDelta for XML posting is shown as the Table 12.

Table 12. Postings compression for XML datasets of Figure2-3.

Data structure	CPEs	ADs	SBs	CNs	Size in bytes (*B*)
32-bit word uncompressed	50×32/8bits=200B	19×32/8bits=76B	8×32/8bits=32B	7×32/8bits=28B	336 *B*
20-bit word uncompressed	50×20/8bits=125B	19×20/8bits=48B	8×20/8bits=20B	7×20/8bits=18B	211 *B*
VB encoding compressed	50×1 B=50 B	19×1 B=19 B	8×1 B=8 B	7×1 B=7 B	84 *B* (60% reduced)
γ encode compressed	50×3 bits/8bits=19 B	19×3 bits/8bits=7 B	8×3 bits/8bits=3 B	7×3 bits/8bits=3 B	32 *B* (62% reduced)
PForDelta	50×8 bits/8bits=50 B	19×8 bits/8bits=19B	8×8 bits/8bits=8 B	7×8 bits/8bits=7 B	88 B

Note that Figures 2-3 have 86 XML postings entirely, 52 postings in CPEs, 19 in ADs, 8 in SBs, and 7 in CNs respectively.

V. CONCLUSION

For efficiently serving versatile queries, a new XML data representation referred to as CPR Dictionary and XML postings have been presented in this paper. The proposed method of representing XML document with complete paths is practical and scalable for the usage in web environment. We employ a number of compression techniques for CPR dictionary and XML postings that are important for efficient XML IR systems and give a satisfied experiment result.

REFERENCES

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

[2] Berglund A, Boag S, and Chamberlin D, XML Path Language (XPath) 2.0, W3C Recommendation, http://www.w3.org/TR/xpath20/ (January 2007).

[3] Robie J and Hat R. XML Processing and Data Integration with XQuery. IEEE Internet Computing 2007; 11: 62-67.

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

[5] Nierman A and Jagadish HV. Evaluating Structural Similarity in XML Documents. Fifth International Workshop on the Web and Databases 2002.

[6] Flesca S, et al. Fast Detection of XML Structural Similarity. IEEE Transactions on Knowledge and Data Engineering 2004; 17: 160-175.

[7] Lian W, et al. An Efficient and Scalable Algorithm for Clustering XML Documents by Structure. IEEE Transactions on Knowledge and Data Engineering 2004; 16: 82-96.

[8] Kozielski M. Improving the Results and Performance of Clustering Bit-encoded XML Documents. Sixth IEEE International Conference on Data Mining-Workshop 2006.

[9] Yuan JS, Li XY and Ma LN. An Improved XML Document Clustering Using Path Feature. Fifth International Conference on Fuzzy Systems and Knowledge Discovery 2008; 2: 400-404.

[10] Leung HP, Chung FL, Chan SCF and Luk R. XML Document Clustering Using Common Xpath. Proceedings of the International Workshop on Challenges in Web Information Retrieval and Integration 2005; 91 -96.

[11] Termier A, Rousset MC and Sebag M. treefinder: a first step towards XML data mining. Proceedings of IEEE International Conference on Data Mining 2002; 450-457.

[12] Yang J, Cheung WK and Chen X. Learning the Kernel Matrix for XML Document Clustering. IEEE International Conference on e-Technology, e-Commerce and e Service 2005; 353-358.

[13] Liu J, Wang JTL, Hsu W and Herbert KG. XML Clustering by Principal Component Analysis. Proceedings. Of the 16th IEEE International Conference on Tools with Artificial Intelligence. 2004; 658-662.

[14] Lee JW, Lee K and Kim W. Preparation For Semantic-Based XML Mining. IEEE International Conference on Data Mining 2001, pp. 345-352.

[15] Qureshi MH and Samadzadeh MH. Determining the Complexity of XML Documents. Proceedings of the International Conference on Information Technology: Coding and Computing 2005; 2: 416-421.

[16] Li XY. Using Clustering Technology to Improve XML Semantic Search. Proceedings of the Seventh International Conference on Machine Learning and Cybernetics 2008; 5: 2635-2639.