CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2013; **25**:1604–1621
Published online 24 July 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.2906

LAF: a new XML encoding and indexing strategy for keyword-based XML search

Zhi-Hong Deng*,†, Yong-Qing Xiang and Ning Gao

Key Laboratory of Machine Perception (Ministry of Education), School of Electronics Engineering and Computer Science, Peking University, Beijing, China

ABSTRACT

As a large number of corpuses are represented, stored and published in XML format, how to find useful information from XML databases has become an increasingly important issue. Keyword search enables web users to easily access XML data without the need to learn a structured query language or to study complex data schemas. Most existing indexing strategies for XML keyword search are based upon Dewey encoding. In this paper, we proposed a new encoding method called Level Order and Father (LAF) for XML documents. With LAF encoding, we devised a new index structure, called two-layer LAF inverted index, which can greatly decrease the space complexity compared with Dewey encoding-based inverted index. Furthermore, with two-layer LAF inverted index, we proposed a new keyword query algorithm called Algorithm based on Binary Search (ABS) that can quickly find all Smallest Lowest Common Ancestor. We experimentally evaluate two-layer LAF inverted index and ABS algorithm on four real XML data sets selected from Wikipedia. The experimental results prove the advantages of our index method and querying algorithm. The space consumed by two-layer LAF index is less than half of that consumed by Dewey inverted index. Moreover, ABS is about one to two orders of magnitude faster than the classic Stack algorithm. Concurrency and Computation: Practice and Experience, 2012.© 2012 Wiley Periodicals, Inc.

Received 19 January 2011; Revised 27 June 2012; Accepted 27 June 2012

KEY WORDS: XML keyword search; LAF; two-layer index; ABS; SLCA

1. INTRODUCTION

XML is a new kind of markup language emerging from 1998. Because XML provides a basic syntax that can be used to share information between different kinds of computers, different applications, and different organizations without needing to pass through numerous layers of conversion, XML has become the standard to represent, exchange, and share data on the web, leading to the rapidly increasing of documents published in XML format.[‡] How to store and retrieve XML documents has become an urgent and popular problem in the field of data management [1]. Keyword-based XML search is a proven user-friendly way of querying XML documents. It allows users to find information they are interested in without the need to learn a complex query language or prior structure knowledge of the underlying data.

Different from traditional keyword search, XML keyword search returns, instead of full documents, relevant elements that contain given query keywords as searching results [2].

Therefore, XML keyword search brings several new challenges. The ultimate challenge is how to improve the efficiency and effectiveness of keyword search by considering the hierarchical structure of XML documents. At present, there are many studies on XML keyword search. Most of the

*Correspondence to: Zhi-Hong Deng, Peking University, Beijing, China.
†E-mail: zhdeng@cis.pku.edu.cn
‡http://www.w3.org/XML

studies focus on the query semantics and their corresponding algorithms [3,2,4–6], the efficiency of algorithms [7, 8], result generation and presentation [9–14], and top-K keyword search [15,5,16]. There are few works on the index methods for XML keyword search. Since Dewey encoding was first introduced to label and index XML elements by [2], it has become the dominant encoding schema to store and index XML documents because it contains rich structure information for implementing XML keyword search efficiently. In addition to the index methods based on Dewey encoding, some other indexing methods [17–21] have been proposed for indexing XML documents. However, these indexing methods are not convenient for computing Lowest Common Ancestor (LCA) [2], the fundamental component of XML keyword search. Therefore, most XML keyword search systems [22, 23, 2,24,4, 5, 25,6] adopt index methods based on Dewey encoding.

However, Dewey encoding has two obvious disadvantages. First, it may cause indexing redundancy. The reason lies in the following: (i) with the increasing of elements' depth in XML tree, the Dewey ID of elements get longer and longer and (ii) each element keeps its parent's Dewey ID as code prefix. Second, it is inefficient for Dewey encoding to compare two Dewey IDs. To obtain the LCA of two Dewey IDs, we have to compare every dimension of the common prefix of these IDs.

To solve the aforementioned problems caused by Dewey encoding, we present a new labeling schema and indexing solution: Level Order and Father (LAF) encoding and two-layer LAF inverted index. In addition, a new XML searching algorithm based on two-layer LAF inverted index called Algorithm based on Binary Search (ABS) is proposed. The experiment results show that two-layer LAF inverted index outperforms Dewey inverted index in space complexity comparison and ABS is much better than the classic Stack algorithm in efficiency.

Although LAF was first proposed briefly in [26], this paper makes great progress as follows.

1. We thoroughly discuss the work related to LAF and two-layer LAF inverted index. The content used to explaining and analyzing these two parts gets doubled compared with [26].
2. We explicitly introduce a new algorithm, ABS, which is very efficient in finding the query results according to Smallest LCA (SLCA) semantic by fully making use of two-layer LAF inverted index. This part is not shown in [26].
3. Additional experimental results on efficiency have been reported extensively. We examine the efficiency of our ABS algorithm with the classic Stack algorithm on four data sets extracted from Wikipedia with different sizes.

The remainder of the paper is organized as follows. Section 2 introduces the relevant concepts. Section 3 introduces LAF encoding, a new kind of encoding schema. Section 4 presents a new indexing structure called two-layer LAF inverted index, based on LAF encoding. In Section 5, we propose a new algorithm ABS to find the query results quickly according to SLCA semantic. Experimental results are presented in Section 6. Section 7 summarizes our study and points out some future research issues.

2. BASIC PRINCIPLES

In this section, we firstly introduce the data structure of XML documents briefly. Then, an introduction to SLCA, the currently popular query semantics model for XML keyword query, is given.

2.1. Data structure

As is known, XML is a hierarchical format for data representation and exchange. An XML document consists of nested XML elements starting with the root element. Each element can contain attributes and values. Figure 1 is an XML document representing the introduction of a book. The <book> element in the second line is the root, in which <name>, <author>, and <chapter> are the underlying subelements. The <chapter> element contains an attribute ID whose value is '1'. The <section> element in line 10 is a subelement of <subchapter>, with text value 'Information Retrieval at the Center of the Stage'.

DOI: 10.1002/cpe

```
<?xml version="1.0" encoding="utf-8">
<book>
  <name>Modern Information Retrieval</name >
  <author>Ricardo Baeza-Yates </author>
  <author>Berthier Ribeiro-Neto</author>
  <Chapter id="1">
    <name>introduction</name>
    <pages>1 to 17</pages>
    <subchapter name="Motivation">
      <section>Information Retrieval at the
              Center of the Stage</section>
      <section> Information Retrieval versus
              Data Retrieval </section>
      <section>Focus of the Book</section>
    </subchapter>
    <subchapter name="Past, Present and Future">
      <section>Early Developments</section>
      <section>Information Retrieval in the
                              library</section>
      <section>
          The web and Digit Libraries
      </section>
      <section>Practical Issues</section>
    </subchapter>
  </chapter>
  <reference>
    <name>Models and Theories of Information
                          Retrieval</name>
    <date>April 1998</date>
    <author>Jim Green</author>
  </reference>
  <reference>
    <name>How to Retrieval Information From
                            Database</name>
    <date>June 1975</date>
    <author>Franck Obma</author>
  </reference>
</book>
```

Figure 1. An XML document example.

XML documents are ordinarily modeled as XML trees because of their hierarchical structure. In Document Object Model, an XML document is also treated as a Document Object Model tree. The root element is mapped to the root node of the tree. The containing relationships of elements are modeled as parent–child relations in the tree. Figure 2 renders the tree structure of XML document in Figure 1, labeled with Dewey IDs [6].

2.2. *Smallest Lowest Common Ancestor*

This section presents SLCA, one of the predominant definitions for the results of XML keyword search. To satisfy a query Q, each of the required terms in Q must be matched. SLCA exploits XML tree model to represent XML data. In the following, we will introduce the fundamental concepts LCA [6] and SLCA [6].

DOI: 10.1002/cpe

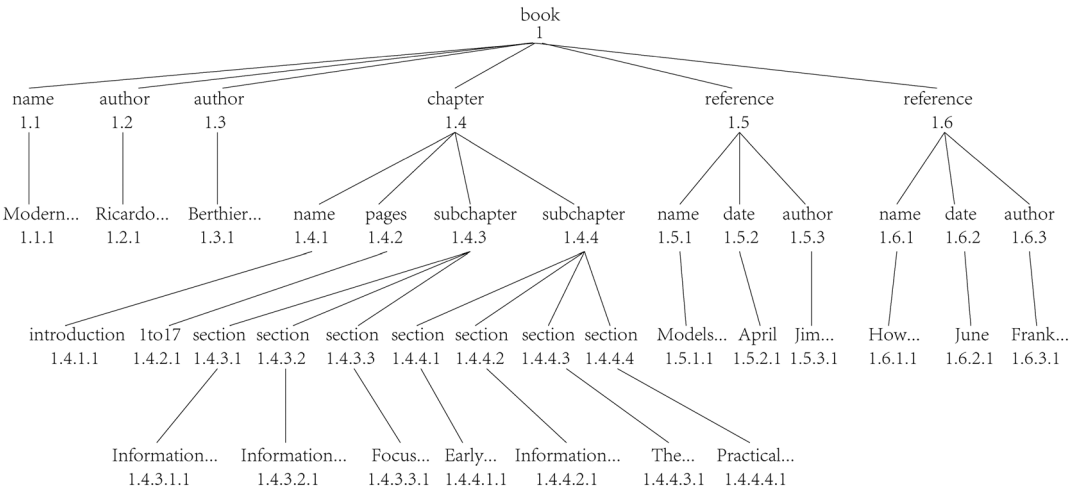

Figure 2. An XML tree labeled with Dewey IDs.

Definition 2.1 Lowest Common Ancestor

Given a keyword query $Q = \{k_1, \ldots, k_m\}$, where k_i $(1 \le i \le m)$ is a keyword, and an XML tree X_{tree}, we assume that V_i $(1 \le i \le m)$ is the set of nodes that directly contains keyword k_i in X_{tree}. LCA(Q, X_{tree}) is defined as follows:

$$\text{LCA}(Q, X_{\text{tree}}) = \{n | \exists (v_1 \in V_1, \ldots, v_m \in V_m),\ \mathsf{n} \text{ is the LCA of } v_1, \ldots, v_m\}.$$

Definition 2.2 Smallest Lowest Common Ancestor

Given a keyword query $Q = \{k_1, \ldots, k_m\}$, an XML tree X_{tree}, SLCA(Q, X_{tree}) is defined as follows:

$$\text{SLCA}(Q, X_{\text{tree}}) = \left\{ n | n \in \text{LCA}(Q, X_{\text{tree}}) \hat{} \neg \left(\exists n' \in \text{LCA}(Q, X_{\text{tree}}), n \succ n' \right) \right\},$$
$$\text{where } n \succ n' \text{ means is } n \text{ an ancestor of } n'$$

3. LEVEL ORDER AND FATHER ENCODING

At present, the most widely used encoding method is the Dewey encoding. In this section, we will firstly introduce the Dewey encoding. Then, we propose our new XML document encoding strategy called LAF.

3.1. Dewey encoding

Dewey encoding was introduced by [2] to store and query XML documents at the first time. With Dewey order, each node is assigned a vector that represents the path from the document's root to itself. Each component of the path represents the local order (with local order, each node is assigned a number that represents its relative position among its siblings) of an ancestor node. Figure 2 is an XML document tree labeled with Dewey order encoding for the XML document showed in Figure 1. It is easy to conclude the most crucial benefit of Dewey encoding: it is easy to obtain the LCA node of arbitrary nodes and judge the relationship of arbitrary two nodes in XML tree.

DOI: 10.1002/cpe

Specifically, the LCA node's Dewey ID is the longest common prefix of these node's Dewey IDs. To give an example, Node 1.4.1.1, Node 1.4.3.1.1, and Node 1.4.4.3.1 are three nodes in Figure 2. The longest common prefix of these nodes is 1.4, so that Node 1.4 is their LCA.

Nevertheless, Dewey order encoding suffers two obvious disadvantages: (i) every subnode keeps its parent node's Dewey encoding as its prefix, causing storing redundancy and increasing the space complexity of XML indices, and (ii) the query-processing algorithm based on Dewey encoding may cause low efficiency.

3.2. Level Order and Father encoding schema

To overcome the aforementioned two disadvantages of Dewey encoding, we put forward a new encoding schema for XML documents called LAF, which is short for LAF encoding. LAF encoding is built on level order tree traversal, a global traversal strategy. When traversing an XML tree by level order, we firstly visit the root of the tree and then traverse the tree level by level until all of the nodes in the tree are visited. Figure 3 labels the XML tree by its order of level order traversal. The level order sequence of XML tree in figure 3 is A, B, C, D, E, F, G, H, I, J; hence, the level order encoding of these ten nodes should be 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

A LAF is a novel labeling schema for XML tree. Each node in the XML tree is signed by a unique LAF ID. As indicated in Figure 4, a LAF ID is made up by three parts, represented as the level order sequence number of current node, the level order sequence number of current node's parent node, and the level number of current node. Especially, if current node is the root node, its parent node's level order sequence number is set to -1. The level number initials from 0, and it increases level by level. The structure of LAF encoding is illustrated as Figure 4.

The LAF IDs can be stored in a vector with only three dimensions. In Figure 3, It is easy to know that Node A's LAF ID is 0–1.0. The reason lies in the following: (i) node A's level order sequence number is 0, so the first dimension of LAF vector is 0; (ii) node A is the root of XML tree; consequently, its parent node's level order sequence number is set to -1; and (iii) node A is at the first level of XML tree; therefore, its level number is 0. By parity of reasoning, we can label the tree in Figure 2 as Figure 5 by using LAF IDs.

Note that the LAF ID for each node in an XML tree is unique because of its solitary level order sequence number. On the other hand, given the LAF IDs of nodes, it is effortless to build the XML tree. This is to say, there is a one-to-one mapping relationship between the LAF IDs and the nodes in an XML tree. The following are some useful properties of LAF encoding.

4. TWO-LAYER INVERTED INDEXING BASED ON LEVEL ORDER AND FATHER ENCODING

An inverted index (also addressed as posting file or inverted file) is an index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a

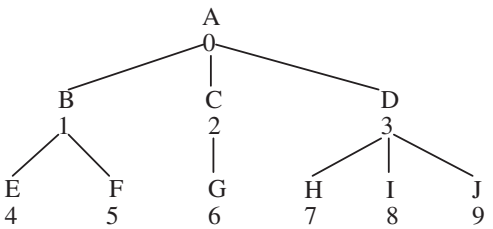

Figure 3. Level order encoding on XML tree.

Level order sequence number	Parent node's level order sequence number	Level number

Figure 4. The structure of Level Order and Father encoding.

DOI: 10.1002/cpe

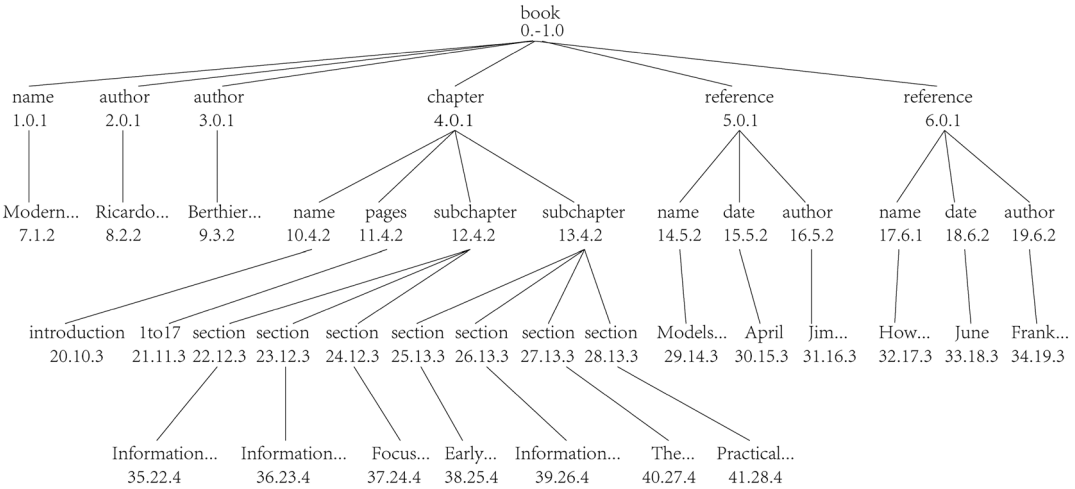

Figure 5. Level Order and Father encoding example.

document or a set of documents. Inverted index is the most widely used data structure in document retrieval systems. This section will introduce Dewey encoding-based inverted index firstly; then, we propose a new index structure based on LAF encoding.

4.1. Dewey inverted index

Dewey inverted index, also referred to as Dewey inverted list, is now the most popular index structure used in XML document retrieval systems, such as XRANK [4] and XKSearch [10]. The inverted list for a keyword k contains the Dewey IDs of all the XML elements that directly contain the keyword k. To handle multiple documents, the first dimension of each Dewey ID should be set as the document ID. Associated with each Dewey ID entry in Dewey inverted list is the rank of the corresponding XML element and the list of positions where the keyword k appears in certain element. The rank here denotes the objective importance of the XML node and can be calculated by a ranking function such as ElemRank [4]. Figure 6 shows an example Dewey inverted list. Here 'Retrieval' and 'Ricardo' are two keywords in the document showed in Figure 1. The first column corresponds to the Dewey IDs of the nodes where the keyword occurs. The second column contains the ElemRank of the particular occurrences. Moreover, the third column keeps the position of the occurrence within the node. The entries are sorted by the Dewey IDs.

Dewey encoding-based inverted index is effective for XML document indexing. However, each node in the tree restores its parent's Dewey ID as prefix, leading to redundancy and increasing the space complexity of XML indices. To solve these disadvantages caused by Dewey encoding, in the following, we introduce a new index structure called two-layer LAF inverted index.

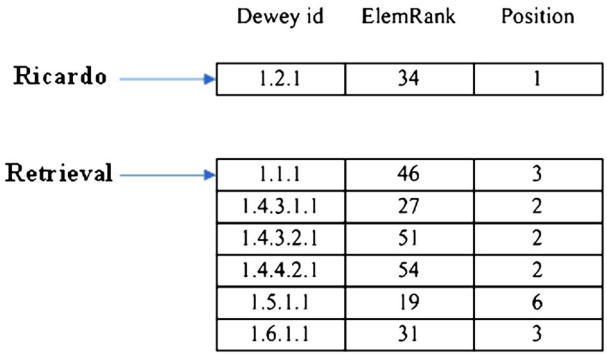

Figure 6. Dewey inverted list.

DOI: 10.1002/cpe

4.2. Two-layer Level Order and Father inverted index

As is known, the ultimate difference between retrieval on plain text and XML document is that XML documents contain structure information besides content. Therefore, the objective of XML querying is the elements rather than the entire document. In fact, each XML document contains both plain text and structure information. Forasmuch, the target of our novel two-layer inverted index is supporting these two kinds of information to speed up querying efficiency.

Two-layer inverted index includes three components: (i) the LAF encoding table for each XML documents; (ii) the first layer index, similar to common inverted index; and (iii) the second layer index, similar to Dewey inverted index. We now present the details of these three parts.

Each LAF encoding table stores all the LAF IDs in the corresponding XML tree. If an XML tree contains n nodes, its relevant LAF encoding table also consists of n entries. LAF IDs in LAF table are sorted according to their level order number. LAF encoding table is a crucial part because it stores the structure information of the XML tree. Table I is an example of LAF table for the XML tree in Figure 5.

The first layer index is an inverted index built on XML document, similar to common inverted index except that it should contain a pointer pointing to the second layer index. The first layer index is used to store the term information in an XML document, such as the document ID and document URL. Figure 7 shows the structure of the first layer index.

The second layer index is similar to Dewey inverted index. The only difference is that we store the sequence number of level order of each node instead of its Dewy ID in the second layer index. The second layer index is used to store the term information in an XML element, such as ElemRank value. The inverted list in the second layer index is sorted by the level order number. Figure 8 shows the structure of the second layer index.

Table I. Level Order and Father encoding table.

Level order number	Parent node's level order number	Level number
0	−1	0
1	0	1
2	0	1
3	0	1
4	0	1
5	0	1
6	0	1
7	1	2
8	2	2
9	3	2

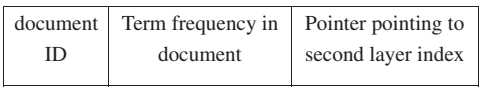

Figure 7. Structure of first layer index.

Figure 8. Structure of second layer index.

DOI: 10.1002/cpe

Figure 9 is an example of two-layer inverted index to store keywords 'Ricardo' and 'Retrieval' in Figure 1, whose document ID is set as 1000. The sequence number of level order of nodes containing these two keywords can be respectively acquired from the XML tree labeled with LAF IDs in Figure 5.

5. ABS: AN ALGORITHM BASED ON BINARY SEARCH TO COMPUTE SMALLEST LOWEST COMMON ANCESTORS

In Section 4, we propose a new index structure for XML documents. Originally, index is used to speed up the query process. Consequently, in this section, we will introduce a new algorithm called ABS, which is the short for ABS to do keyword query process over XML documents.

5.1. Query processing

The ABS is designed based on two-layer LAF inverted index. In this paper, we consider that the returned results for a given query should contain all keywords' matches. Therefore, the key procedure of ABS is as follows. Firstly, we select the document that contains all keywords through the first layer index. This process can reduce enormous quantity of useless elements contained by those irrelevant documents. Secondly, for each selected relevant document, the corresponding inverted element information list for each keyword is procured through the second layer index. Thirdly, we merge and sort these lists by descending order of the level order number for all the keywords in each document. Finally, the procedure goes through the union list in a single pass to obtain the SLCAs for query Q inner each document. Figure 10 shows the pseudo-code of ABS.

5.2. Case study

The core data structure of ABS is the union list (line 6) sorted by element's level order number in descending order. We now walk through the algorithm by using an example. Consider the two-layer LAF inverted index shown in Figure 9 and a keyword query 'Ricardo and Retrieval'. Firstly, the procedure selects the document lists of these two keywords through the first layer index (line 2). It is easy to know that $L_1 = \{1000\}$, $L_2 = \{1000, 1001, 1002\}$. Secondly, the intersection L_0 of these two document lists is computed to {1000} (line 3). Thirdly, for each document in L_0(line 4), document 1000 for example, the procedure obtains the inverted element list for keyword 'Ricardo' as Figure 11 and 'Retrieval' as Figure 12 (line 5); finally, these two element inverted lists are merged as Figure 13 (line 6).

Now, the algorithm processes the entries in the union list in sequence (lines 8–22). The algorithm initially scans the first entry with the largest level order number and then removes it from the union list, so element 39 (here we label an element by its level order number) is removed from union list (line 9). Then, the algorithm checks that element 39 only contains one keyword 'Retrieval' (line 10). Afterwards, the algorithm obtains the parent's level order number of element 39 through the LAF

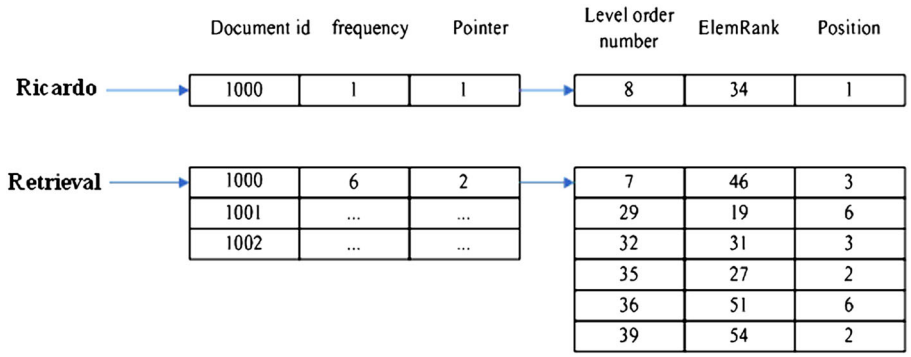

Figure 9. Two-layer Level Order and Father index.

DOI: 10.1002/cpe

```
1 SLCA_Query_Process (K1, K2, …, Kn ){
2    For each keyword Ki, get its document list Li through the first layer index;
3    L0 = intersection(L1, L2, …, Ln); // get the intersection of L1, L2, …, Ln
4    For each document Di in L0 {
5       For each keyword Ki, get its element inverted list ELi through the second layer index;
6       EL0 = union(EL1, EL2, …, ELn); // merge EL1, EL2, …, ELn and is EL0 ordered from
                                          // big to small according to level order number
7       Get the LAF encoding table laf_table_i for Di by the ID of Di;
8       While EL0 is not empty {
9          Get and remove the first element Efirst from EL0;
10         If Efirst is not marked with slca_flag and contains all the keywords
11            Add Efirst to resultList, mark with Efirst slca_flag.
12         Get Efirst's parent Eparent;
13         If Eparent's level order number = -1;
14            Return resultList;
15         Else
16            add Efirst's status to Eparent;
17         Pos = BinarySearch(Eparent; EL0); // return the right position of Eparent
                                             // through binary search on EL0.
18         If(EL0[Pos]== Eparent)
19            add Eparent's status to EL0[Pos];
20         Else
21            Insert Eparent into EL0[Pos]
22      }
23   }
24}
```

Figure 10. The pseudo-code of Algorithm based on Binary Search.

8	34	1

Figure 11. Inverted element list for Ricardo.

7	46	3
29	19	6
32	31	3
35	27	2
36	51	6
39	54	2

Figure 12. Inverted element list for Retrieval.

level order number	retrieval		Ricardo		slca_flag
39	54	2			
36	51	6			
35	27	2			
32	31	3			
29	19	6			
8			34	1	
7	46	3			

Figure 13. Union inverted element list of Ricardo and Retrieval.

DOI: 10.1002/cpe

encoding table of document 1000. From Figure 5, we can know that the parent's level order number of element 39 is 26. Therefore, the procedure adds the status of element 39 to element 26, in which the entry for element 26 is shown in Figure 14 (lines 13–16). Then, the algorithm searches for the appearance of element 26 in the union list through binary search (line 17), discovering that there is no existing entry for it. Therefore, a new entry of element 26 is inserted into the union list right between element 29 and element 8 according to the descending sequence of the list. The modified union list is as figure 15.

As long as the union list is not empty, the algorithm continues processing the next entry element 36. The process of dealing with element 36 is the same as to element 39. After the process, the new union list is shown in Figure 16.

The next entry is element 35. The parent's level order number of element 35 is element 22. The process of element 35 is the same as to element 39, too. After the process, the new union list is shown in Figure 17.

Then, the algorithm processes the elements 32, 29, and 26 in sequence. After processing, the status of the union list is shown in Figure 18.

Because elements 23 and 22 share the same parent, the size of union list reduces by 1, and the new union list after removing these two elements is shown in Figure 18. Element 12 is the common parent of elements 23 and 22; however, element 12 only keeps the status of element 22 because elements 23 and 22 contain the same keyword and element 22 is nearer to element 12 (22–12 $<$ 23–12). Figure 19 shows the union list after processing elements 23 and 22.

Then, the algorithm goes on processing elements 17, 14, 13, 12, 8, and 7. The result is shown in Figure 20.

26	54	2			

Figure 14. Parent element of element 39.

level order number	retrieval		Ricardo		slca_flag
36	51	6			
35	27	2			
32	31	3			
29	19	6			
26	54	2			
8			34	1	
7	46	3			

Figure 15. Union list after processing element 39.

level order number	retrieval		Ricardo		slca_flag
35	27	2			
32	31	3			
29	19	6			
26	54	2			
23	51	6			
8			34	1	
7	46	3			

Figure 16. Union list after processing element 36.

DOI: 10.1002/cpe

level order number	retrieval	Ricardo	slca_flag
32	31	3	
29	19	6	
26	54	2	
23	51	6	
22	27	2	
8		34	1
7	46	3	

Figure 17. Union list after processing element 35.

level order number	retrieval	Ricardo	slca_flag
23	51	6	
22	27	2	
17	31	3	
14	19	6	
13	54	2	
8		34	1
7	46	3	

Figure 18. Union list after processing elements 32, 29, and 26.

level order number	retrieval	Ricardo	slca_flag
17	31	3	
14	19	6	
13	54	2	
12	27	2	
8		34	1
7	46	3	

Figure 19. Union list after processing elements 23 and 22.

level order number	retrieval	Ricardo	slca_flag
6	31	3	
5	19	6	
4	27	2	
2		34	1
1	46	3	

Figure 20. Union list after processing 17, 14, 13, 12, 8, and 7.

After processing elements 6, 5, and 4, the result is shown in Figure 21.

Then, the algorithm processes element 2, after removing element 2 from the union list and adding its status to its parent element 0. The result is shown in Figure 22.

Then, the algorithm processes element 1, after removing element 1 from the union list and adding its status to its parent element 0. The result is shown in Figure 23.

DOI: 10.1002/cpe

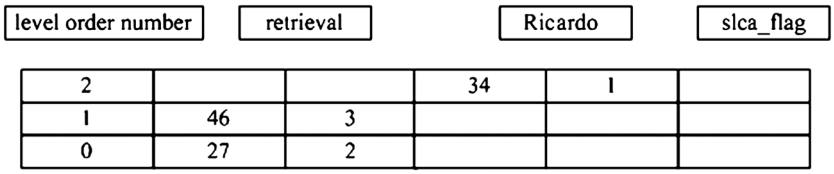

Figure 21. Union List after processing 6, 5, and 4.

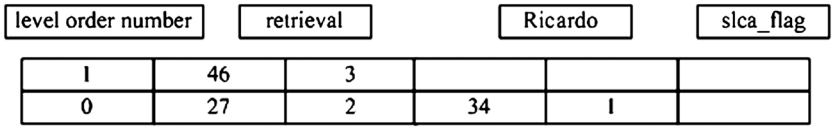

Figure 22. Union list after processing element 2.

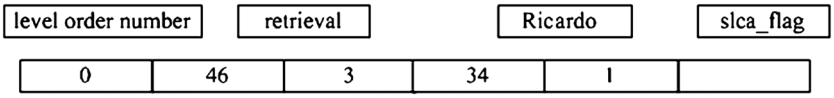

Figure 23. Union list after processing element 1.

Afterwards, the algorithm processes element 0. The algorithm removes element 0 from the union list and then detects that element 0 contains all the keywords. Next, element 0 is added to result list and marked with slca_flag (lines 10–11). Finally, the algorithm returns the result list in respect that the parent of element 0 is -1 (line 14).

Because the document list L_0 contains only one document, the algorithm finishes processing the query. Eventually, the SLCA result for Q (Ricardo and Retrieval) is {1000.0}. Here, 1000 is the document ID, and 0 is the level order number of element in document 1000.

5.3. Analysis

5.3.1. Correctness. The elements computed out by the program SLCA_Query_Process must be SLCA nodes, and the SLCA nodes for the certain relevant document D must be exported by SLCA_Query_Process.

According to the procedure of ABS, the output elements absolutely contain all keywords in the query. On the other hand, only when an element does not contain all keywords in the query its parent element can be inserted into the union list. This rule guarantees that the no offspring of the output element is an SLCA. Thus, according to Definition 2.2, the output elements are SLCAs.

Given an SLCA node n in document D, there must be a vector $V=(k_1, k_2, \ldots k_n)$ that matches the keywords in query, respectively, in which k_i is an arbitrary match in element n for keyword K_i. When dealing with the element that directly contains k_i, the process will trace back to its parent element because it does not contain all keywords. The same tracing process happens to the parent element if it does not contain all keywords either. The tracing operation ends when detecting node n because it conforms to the definition of SLCA node. As a result, all SLCA nodes in D could be exported by the algorithm SLCA_Query_Process.

5.3.2. Time complexity. According to the program SLCA_Query_Process, the time complexity of computing the SLCA nodes of a relevant document D is $O(ml\log m)$, where m is the number of elements in D that contains keyword matches and l is the largest level number of these elements.

The process loop computing SLCA nodes starts at line 4 and ends at line 23. In line 6, the inverted lists of keywords are united by the elements' level order number in descending order. The time

DOI: 10.1002/cpe

Table II. Details of the data sets.

Number of documents	Data set size (KB)	Total elements	Total keywords
500	9505	210801	455612
1000	18775	421458	887186
1500	27775	622793	1313850
2000	37001	827797	1745227

complexity of the sorting process is $O(m\log m)$. The algorithm then needs to deal with each of the elements in the union list. The worst circumstance is that the root node is the only SLCA node. In this case, besides the elements that directly contains keywords, their ancestor nodes are added to the union list. Hence, the loop in line 8 could be possible to run $(ml + 1)$ times. While entering the loop, the number of elements in union list will not be more than m because only after removing an element, its parent element's entry can be inserted. Therefore, the time complexity of the BinarySearch operation in line 17 is $O(m)$. Thus, the while loop in line 8 represents $O(ml\log m)$ complexity. Overall, the time complexity of computing SLCA nodes for a relevant document D is $O(ml\log m)$.

6. EXPERIMENTAL STUDY

We now experimentally evaluate the techniques presented in this paper. Firstly, we introduce the experimental environment and the XML data set used in this paper. Second, we compare two-layer LAF inverted index and traditional Dewey encoding-based inverted index in aspect of space efficiency. Finally, we do a comparison on query performance between traditional Stack algorithm [6] and ABS. It should be noted that the goal of our experiment is to show that the LAF ID is an efficient coding method for XML keyword search. Therefore, we choose SLCA, one of predominant query semantic, and Stack algorithm, one of predominant query algorithm for SLCA in our experiment.

6.1. Experimental setup

We conduct our experiments on the XML data set of Wikipedia English, which is the standard data set used by the Initiative for the Evaluation of XML Retrieval.[§] The comparison experiments are based on four data sets, including 500, 1000, 1500, and 2000 XML documents, respectively. The corresponding sizes for each data set are 9505, 18775, 27775, and 37001 KB. Table II displays the details of the data sets we used. We select XML data from Wikipedia as experimental collection because these data sets are all made up by small pieces of XML documents. The average size of documents in the collection is about 19 KB.

We build traditional Dewey encoding-based index and two-layer LAF inverted index, both stored in Berkeley DB.[¶] Then, we implement the Stack algorithm to compute SLCA results on traditional index and ABS on two-layer LAF inverted index. The experiments are performed on a 1.8-GHz Pentium Dual processor running Microsoft Windows XP operating system with 2.0 GB memory and 160 GB of disk space. The system is implemented in C++.

6.2. Space performance

The space of two-layer LAF Inverted index is made up by the space of the first layer index, second layer index, and LAF tables. Table III shows the details of space performance of two-layer inverted index. From Table III, we can see that the total size of two-layer LAF inverted index for each data set is a little larger than the original data set. Figure 24 shows the space requirements for two-layer index based on LAF encoding and Dewey inverted index, noting that the size of Dewey inverted

[§]http://www.inex.otago.ac.nz/data/documentcollection.asp>.
[¶]http://sleepycat.com

DOI: 10.1002/cpe

Table III. Details of space performance of two-layer Level Order and Father (LAF) inverted index.

Data set size (KB)	Size of first layer index (KB)	Size of second layer index (KB)	Size of LAF tables (KB)	Total size of two-layer inverted index (KB)
9505	3098	6324	2145	11576
18775	5892	12410	4304	22606
27775	8723	18445	6324	33492
37001	11425	24484	8437	44346

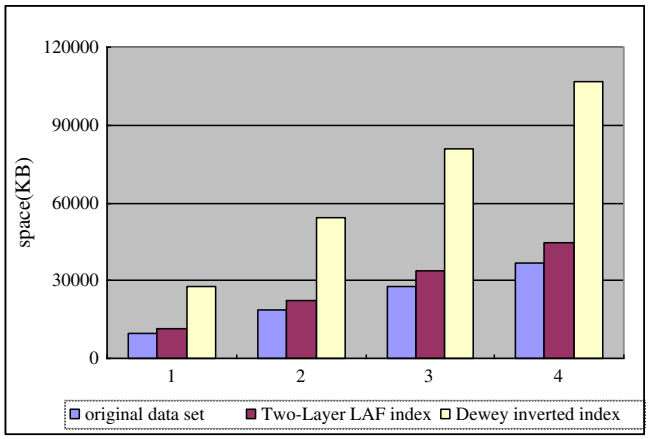

Figure 24. Space performance comparison.

index is about twice that of two-layer LAF index. There are two reasons for this difference. The first reason is that the length of Dewey ID increases as the element gets deeper. For example, in Figure 2, element '1.4' whose depth is 2 needs two integers to record its Dewey ID. On the other hand, the element '1.4.3.3.1' whose depth is 5 needs five integers to record its Dewey ID, whereas the length of LAF ID for any element is a constant value 3. The second reason is that Dewey inverted index contains only one layer index building on elements rather than documents, so that many document features (such as document ID, document length and document URL, and so on) are stored repeatedly. However, these features are stored only once in two-layer LAF inverted index.

6.3. Query performance

In this section, the performances of ABS and Stack algorithm are evaluated. We design our experiments in two different criterions: (i) various numbers of keywords on the same data set (ii) same amount of keywords on different data sets.

Figures 25–27 indicate the performances of Stack algorithm and ABS by querying two–four keywords on different data sets, respectively. As can be seen, ABS algorithm performs much better than Stack algorithm. There are two explanations for this phenomenon. Firstly, the Stack algorithm processes large quantity of meaningless elements included by those documents that do not contain all the keywords. On the contrary, ABS selects the documents containing all keywords at first, and then, the SLCAs are computed inner the documents, avoiding to processing the meaningless elements. Another cause is that the time complexity of comparing two Dewey IDs is $O(n)$ (n is the length of Dewey IDs); however, the time complexity of comparing two level order number is $O(1)$.

Figures 28–31 show the performances of Stack algorithm and ABS by querying two–four keywords on data sets 1(9505 KB), 2(18775 KB), 3(27775 KB), and 4(37001 KB), respectively. In these four figures, we can conclude that the time complexity of Stack algorithm on the same data set increases with the increment of keywords. However, the time complexity of ABS decreases as the number of

DOI: 10.1002/cpe

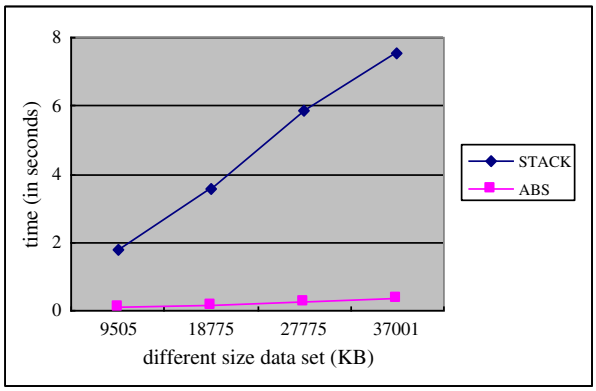

Figure 25. Two keywords on different data set.

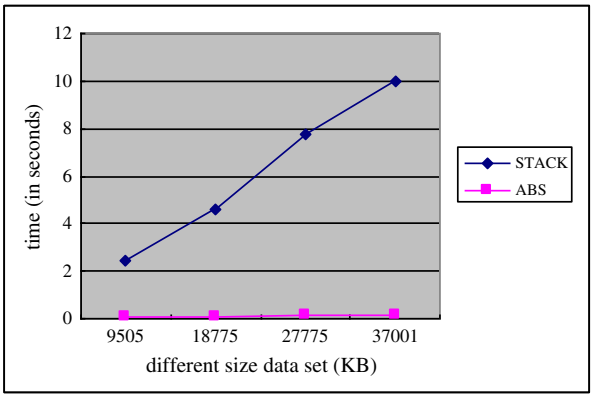

Figure 26. Three keywords on different data set.

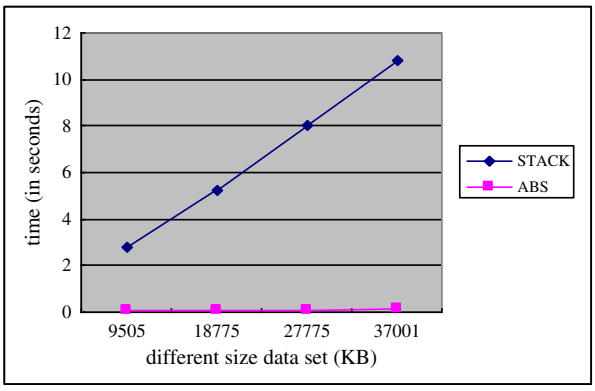

Figure 27. Four keywords on different data set.

keywords increases on the same data set. This is because that with the increasing of keywords, the number of elements containing these keywords increases, leading to more meaningless elements that should be processed by the Stack algorithm. On the contrary, with the addition of keywords, the number of documents containing all of these keywords decreases, which means that the meaningful elements contained by these documents decrease. As a results, the efficiency of ABS increases with the increment of keywords.

DOI: 10.1002/cpe

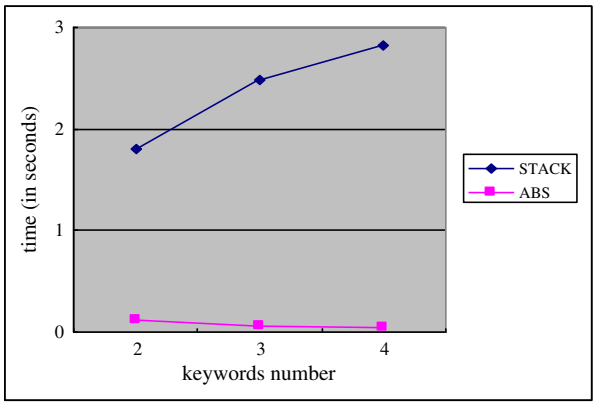

Figure 28. Different keywords on data set 1.

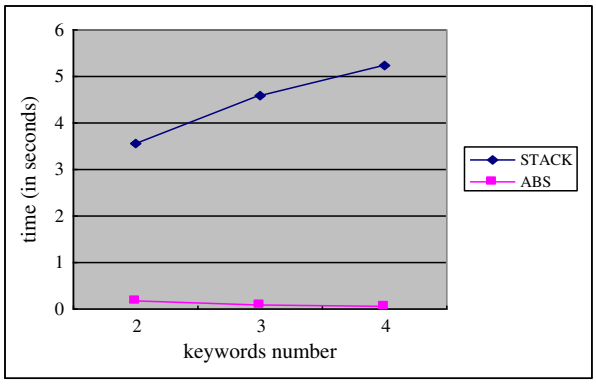

Figure 29. Different keyword on data set 2.

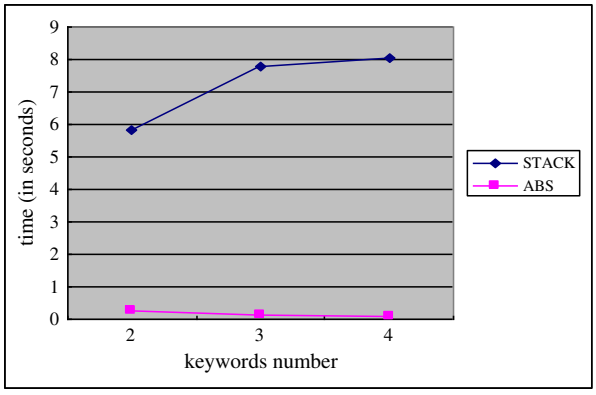

Figure 30. Different keywords on data set 3.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a new encoding schema called LAF for XML documents, which can overcome the shortcomings of Dewey encoding. Afterwards, we designed a new index structure

DOI: 10.1002/cpe

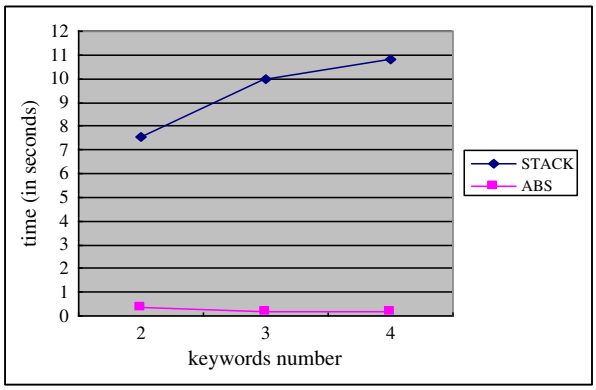

Figure 31. Different keywords on data set 4.

called two-layer LAF-based inverted index. With LAF, we devised a new algorithm called ABS for finding SLCA efficiently. The extensive experiment results showed that our new method demonstrates outstanding space performance and time performance.

For the future work, there are many interesting research issues. Firstly, we will integrate LAF with some state-of-the-art algorithms, such as IMS [23], JDewey [15], DIL [2], IL [6], Scan [6], IS [8], and HC [27], to establish a series of efficient algorithms for various query semantic models. Secondly, we will extend LAF into graph models because XML documents should be a graph when we consider the links between documents. Finally, to rank results efficiently, we will study how to adopt suitable ranking functions in the two-layer LAF inverted index.

ACKNOWLEDGMENT

This work is partially supported by Project 61170091 supported by the National Natural Science Foundation of China and Project 2009AA01Z136 supported by the National High Technology Research and Development Program of China (863 Program).

REFERENCES

1. Beyer K, Viglas SD, Tatarinov I, Shanmugasundaram J, Shekita E, Zhang C. Storing and querying ordered XML using a relational database system. In ACM SIGMOD'02, 2002.
2. Guo L, Shao F, Botev C, Shanmugasundaram J. XRANK: ranked keyword search over XML documents. In ACM SIGMOD'03, 2003.
3. Feng J, Li G, Wang J, Zhou L. Finding and ranking compact connected trees for effective keyword proximity search in XML documents. Information Systems, 2009.
4. Li G, Feng J, Wang J. Effective keyword search for valuable LCAs over XML documents. In ACM CIKM'07, 2007.
5. Li G, Li C, Feng J, Zhou L. SAIL: structure-aware indexing for effective and progressive top-k keyword search over XML documents. Information Sciences, 2009.
6. Xu Y, Papakonstantinou Y. Efficient keyword search for smallest LCAs in XML databases. In ACM SIGMOD'05, 2005.
7. Sun C, Chan CY, Goenka AK. Multiway SLCA-based keyword search in XML data. In WWW'07, 2007.
8. Xu Y, Papakonstantinou Y. Efficient LCA based keyword search in XML data. In ACM CIKM'07, 2007.
9. Huang Y, Liu Z, Chen Y. Query biased snippet generation in XML search. In ACM SIGMOD'08, 2008.
10. Kong L, Gilleron R, Lemay A. Retrieving meaningful relaxed tightest fragments for XML keyword search. In EDBT'09, 2009.
11. Liu Z, Chen Y. Identifying meaningful return information for XML keyword search. In ACM SIGMOD'07, 2007.
12. Liu Z, Chen Y. Reasoning and identifying relevant matches for XML keyword search. In VLDB'08, 2008.
13. Liu Z, Chen Y. Return specification inference and result clustering for keyword search on XML. *ACM TODS* 2010; **35**(2). DOI: 10.1145/1735886.1735889.
14. Liu Z, Huang Y, Chen Y. Improving XML search by generating and utilizing informative result snippets. *ACM TODS* 2010; **35**(3).
15. Chen L, Papakonstantinou Y. Supporting top-K keyword search in XML databases. In IEEE ICDE'10, 2010.
16. Yu H, Deng Z, Xiang Y, Gao N, Ming Z, Tang S. Adaptive top-k algorithm in SLCA-based XML keyword search. In APWeb'10, 2010.
17. Grimsmo N. Faster path indexes for search in XML data. In ADC'08, 2008.
18. Harder T, Haustein M, Mathis C, Wagner M. Node labeling schemes for dynamic XML documents reconsidered. *Data & Knowledge Engineering* 2007; **60**(1):126–149.

DOI: 10.1002/cpe

19. Luk RWP, Leong HV, Dillon TS, *et al*. A survey in indexing and searching XML documents. *Journal of the American Society for Information Science and Technology* 2002; **53**(6):415–437.
20. O'Neil PE, O'Neil EJ, Pal S, Cseri I, Schaller G, Westbury N. ORDPATHs: insert-friendly XML node labels. In ACM SIGMOD'04, 2004.
21. Xu L, Ling TW, Wu H, Bao Z. DDE: from Dewey to a fully dynamic XML labeling scheme. In ACM SIGMOD'09, 2009.
22. Bao Z, Ling TW, Chen B, Lu J. Effective XML keyword search with relevance oriented ranking. In IEEE ICDE'09, 2009.
23. Bao Z, Lu J, Ling TW, Chen B. Towards an effective XML keyword search. *IEEE Transactions on Knowledge and Data Engineering* 2010; **22**(8):1077–1092.
24. Huang Y, Liu Z, Chen Y. eXtract: generating query biased result snippet for XML search. Demo description. In VLDB'08, 2008.
25. Liu ZY, Walker J, Chen Y. XSeek: a semantic XML search engine using keywords. In VLDB'07, 2007.
26. Xiang Y, Deng Z, Yu H, Wang S, Gao N. A new indexing strategy for XML keyword search. In FSKD'10, 2010.
27. Zhou R, Liu C, Li J. Fast ELCA computation for keyword queries on XML data. In EDBT'10, 2010.

DOI: 10.1002/cpe