Title	Superimposed Code-Based Indexing Method for Extracting MCTs from XML Documents
Author	Wenxin Liang, Takeshi Miki, Haruo Yokota
Journal/Book name	Database and Expert Systems Applications of LNCS (Proc. of DEXA2008), Vol. 5181, , pp. 508-522
発行日 / Issue date	2008, 9
DOI	http://dx.doi.org/10.1007/978-3-540-85654-2_44
権利情報 / Copyright	The original publication is available at www.springerlink.com.
Note	このファイルは著者（最終）版です。 This file is author (final) version.

Superimposed Code-based Indexing Method for Extracting MCTs from XML Documents

Wenxin Liang[1,4], Takeshi Miki[2], and Haruo Yokota[3,4]

[1] CREST, Japan Science and Technology Agency (JST)
[2] Nomura Research Institute
[3] Department of Computer Science, Tokyo Institute of Technology
[4] Global Scientific Information and Computing Center, Tokyo Institute of Technology
{wxliang, takeshi}@de.cs.titech.ac.jp, yokota@cs.titech.ac.jp

Abstract. With the exponential increase in the amount of XML data on the Internet, information retrieval techniques on tree-structured XML documents such as keyword search become important. The search results for this retrieval technique are often represented by minimum connecting trees (MCTs) rooted at the lowest common ancestors (LCAs) of the nodes containing all the search keywords. Recently, effective methods such as the stack-based algorithm for generating the lowest grouped distance MCTs (GDMCTs), which derive a more compact representation of the query results, have been proposed. However, when the XML documents and the number of search keywords become large, these methods are still expensive. To achieve more efficient algorithms for extracting MCTs, especially lowest GDMCTs, we first consider two straightforward LCA detection methods: keyword B$^+$trees with Dewey-order labels and superimposed code-based indexing methods. Then, we propose a method for efficiently detecting the LCAs, which combines the two straightforward indexing methods for LCA detection. We also present an effective solution for the false drop problem caused by the superimposed code. Finally, the proposed LCA detection methods are applied to generate the lowest GDMCTs. We conduct detailed experiments to evaluate the benefits of our proposed algorithms and show that the proposed combined method can completely solve the false drop problem and outperforms the stack-based algorithm in extracting the lowest GDMCTs.

1 Introduction

Recently, there has been an exponential increase in the amount of data, such as life science data [20,22], bibliography data [24] and online encyclopedia data [23], that are disseminated and shared over the Internet in the form of XML documents. These are often modeled as ordered labeled trees. Information retrieval techniques on tree-structured XML documents such as keyword search are therefore important. Keyword search allows users to find relevant information without any prior knowledge of the schema of the underlying data or any need to learn complex queries [1, 2, 4, 11, 14, 25]. For example, assume an XML document consists of Shakespeare's plays in Figure 1. Users might be interested in finding the

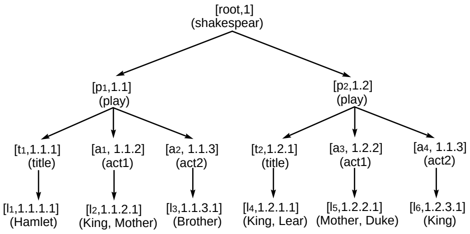


Fig. 1. An example XML document tree (labeled by Dewey number)

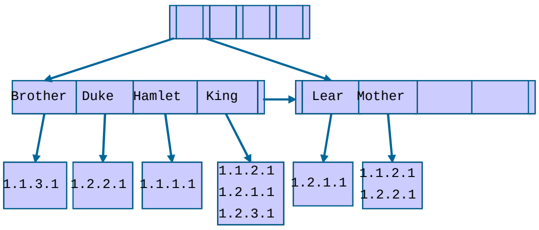


Fig. 2. An example keyword B+tree

relationship between the query keywords *king* and *mother*. The search system returns the relevant answers corresponding to the query keywords that might all appear within the same act or in different acts but within the same play, and so on.

In keyword searches over XML documents, the search results are often represented by *minimum connecting trees* (MCTs) rooted at the *lowest common ancestors* (LCAs) of the nodes containing all the query keywords. Therefore, keyword searching over XML document trees resolves the problem of detecting the LCAs of the nodes that contain all the query keywords. For example, the answer for the query by keywords *king* and *mother* over the XML tree of Figure 1 might be the subtrees rooted at $[a_1]$ and $[p_2]$.

Recently, many studies have been conducted on detecting the LCAs of the nodes containing query keywords over XML documents [8,12,14,18,25]. However, these methods only focus on finding LCAs and do not consider techniques for extracting XML subtrees rooted at the LCAs. To provide effective query answers to the users, the query system must efficiently extract the MCTs that are the subtrees rooted at the LCAs containing all the keywords. Hristidis et al. [10] propose an efficient stack-based algorithm for computing the MCTs and the lowest grouped distance MCTs (GDMCTs) that derive more compact representation of the query results. However, Hristidis's algorithm still results in expensive time complexity when handling large XML documents with complex keyword queries.

Therefore, a more efficient algorithm for extracting MCTs, especially the lowest GDMCTs containing all the query keywords, is necessary.

We make the following main technical contributions in this paper: 1) To effectively detect the LCAs of the nodes containing all the query keywords, we first consider two straightforward indexing methods: keyword B$^+$trees with Dewey-order labels and superimposed code-based methods. In the first method, the nodes of the XML tree are labeled by Dewey numbers and stored in a B$^+$tree index. Then, the LCAs of the nodes containing the query keywords can be found by comparing the Dewey-order label of each node containing the corresponding keyword. In the superimposed code-based method, fixed-length superimposed codes (signatures) are first assigned to the nodes of the XML tree. Second, the query signature is determined by the logical OR of the signatures of all the query keywords. Finally, the LCAs of nodes containing all the query keywords are determined by the logical AND of the query signature and the signature of each node of the XML tree. However, false drop problems may occur in the superimposed code-based method; 2) Keyword B$^+$trees with Dewey-order labels and superimposed code-based methods are effective in finding the LCAs but expensive in query cost. To reduce the query cost, we propose an efficient LCA detection method that combines the keyword B$^+$tree with the Dewey-order label and the superimposed code-based indexing methods. In this method, both the superimposed codes and the Dewey-order labels of the nodes are first stored in a B$^+$tree index. Second, it searches for the leaf nodes in the B$^+$tree by the signature of any one query keyword. After finding the leaf nodes, the LCAs are determined by a logical AND operation between the query signature and those of the corresponding nodes. The combined method reduces the comparisons of many internal nodes and the query can be completed using only one query keyword. Therefore, it is superior in performance compared with the two original methods. However, the false drop problem still cannot be avoided; 3) We also present an effective method to solve the false drop problem. In this method, the signatures of all the keywords are used to find the corresponding leaf nodes in the B$^+$tree index. Then, the LCAs can be determined by detecting the common Dewey-order labels of the corresponding nodes; 4) We apply the proposed LCA detection methods to generate the lowest GDMCTs. We perform experiments to evaluate the performance of detecting LCAs comparing the proposed combined method with the original ones. The experimental results indicate that our proposed LCA detection method is cost-efficient and can completely solve the false drop problem. We also conduct experiments to compare the query time using our proposed methods with that using the stack-based method. The experimental results show that the proposed combined method can completely solve the false drop problem and outperforms the previously known stack-based method in extracting the lowest GDMCTs.

The remainder of the paper is organized as follows. In Section 2, we briefly introduce related work. Section 3 describes the notation and definitions of LCAs and MCTs. In Section 4, we discuss two straightforward methods for finding LCAs and propose an efficient method combining the two methods. In Section 5, we describe the lowest GDMCTs extraction algorithms based on the proposed LCA detection methods and compare the query costs of the proposed methods and the SA algorithm. Section 6 describes the experimental evaluation, shows the benefits of our approach and compares it with the stack-based algorithm. Finally, Section 7 concludes this paper and outlines future work.

2 Related Work

The first research area relevant to this work is LCA computation to detect the LCA of two or more nodes over tree-structured data such as XML documents. Computation of the LCA of two nodes has been intensively studied over the past 30 years. References [3] and [21] first introduced the problem of finding LCAs in trees. In [9], Harel and Tarjan introduced upper and lower bounds for the problem of LCAs. In [17], Schieber et al. presented a simpler algorithm with optimal asymptotic bounds for finding the LCAs in trees. References [9] and [15] showed that the LCA query can be computed in constant time after linear-time preprocessing in arbitrarily directed trees. However, these algorithms are still too complicated to implement effectively. In [25], Xu et al. proposed efficient algorithms for computing the smallest LCAs in XML databases using Dewey-order labels. Computing the LCA of nodes utilizing these labels does not require any disk access that would degrade the performance of the query. The concept of a binary superimposed code was first introduced by Kautz and Singleton [13]. Since then, it has been extensively studied and applied to many areas such as data security and cryptology [7, 19], broadcasting in radio networks [5] and so on. In our proposed method, we use Dewey-order labels and superimposed codes to find the LCAs.

The second area of research relevant to this paper is the work on keyword search over XML documents. XRANK [8] considered the problem of producing ranked results for keyword search queries in hierarchical and hyperlinked XML documents. A specific document fragment, namely the subtree, is returned as the keyword search results in XRANK. In [6], Cohen et al. proposed a semantic search engine over XML documents that employed more techniques of information retrieval than XRANK. However, these studies did not consider the problem of extracting the MCTs containing the query keywords. In [10], Hristidis et al. proposed an effective stack-based algorithm (SA) for computing the lowest GDMCTs rooted at the LCAs of nodes containing the query keywords. However, as XML documents and the number of query keywords become larger because of more complex queries, Hristidis's algorithm still causes high query costs.

3 Notation and Definitions

In this section, we introduce notation and definitions of the LCAs and MCTs used in this paper[5]. An XML document is represented by the conventional order

[5] Some notation and definitions described in this section refer to the reference [10].

labeled tree T. Each node v of the XML tree T corresponding to an XML element or leaf is labeled with a tag or string value $\lambda(v)$. Each node is assigned a unique id $id(v)$ and a Dewey-order label $lab(v)$ as a 2-tuple column $[id(v), lab(v)]$, as shown in Figure 1.

Definition 1 (LCA). *Given a set of n nodes $v_1, ..., v_n$ and an input XML tree T, the LCA of the set of nodes $v_1, ..., v_n$, $lca(v_1, ..., v_n)$ is defined as the node v in T that is ancestor to all the nodes $v_1, ..., v_n$, and is farthest from the root.*

Definition 2 (MCT). *Given a set of n nodes $v_1, ..., v_n$ and an input XML tree T, the MCT of nodes $v_1, ..., v_n$ is the minimum subtree T_m that connects $v_1, ..., v_n$. Conversely, the MCT of nodes $v_1, ..., v_n$ is the subtree rooted at the LCA of $v_1, ..., v_n$.*

Given a list of m keywords, $k_1, ..., k_m$, the MCT of keywords $k_1, ..., k_m$ is the MCT of nodes $v_1, ..., v_n$ that contain $k_1, ..., k_m$. For example, given two query keywords (*King*, *Mother*) for the XML tree in Figure 1, the MCTs containing both keywords are shown in $(1-6)$ as follows, where each root of $(1-6)$ is an LCA that contains the two keywords.

$$\begin{tikzcd} root \arrow[r] \arrow[dr] & p_1 \arrow[r] & a_1 \arrow[r] & l_2 \\ & p_2 \arrow[r] & t_2 \arrow[r] & l_4 \end{tikzcd} \tag{1}$$

$$\begin{tikzcd} p_2 \arrow[r] \arrow[dr] & t_2 \arrow[r] & l_4 \\ & a_3 \arrow[r] & l_5 \end{tikzcd} \tag{2}$$

$$\begin{tikzcd} root \arrow[r] \arrow[dr] & p_1 \arrow[r] & a_1 \arrow[r] & l_2 \\ & p_2 \arrow[r] & a_1 \arrow[r] & l_5 \end{tikzcd} \tag{3}$$

$$a_1 \rightarrow l_2 \tag{4}$$

$$\begin{tikzcd} root \arrow[r] \arrow[dr] & p_1 \arrow[r] & a_1 \arrow[r] & l_2 \\ & p_2 \arrow[r] & a_4 \arrow[r] & l_6 \end{tikzcd} \tag{5}$$

$$\begin{tikzcd} p_2 \arrow[r] \arrow[dr] & a_3 \arrow[r] & l_5 \\ & a_4 \arrow[r] & l_6 \end{tikzcd} \tag{6}$$

Assume a list $l_i (1 \leq i \leq m)$ of nodes that contain keywords k_i. The number N of MCTs for keywords $k_1, ..., k_m$ can be determined by the length of the list $|l_i|$ as follows:

$$N = |l_1| \times |l_2| \times ... \times |l_k|. \tag{7}$$

Therefore, larger numbers of query keywords generate more MCTs. To reduce the redundancy of MCTs, a set of MCTs can be combined into *grouped distance trees*. We first define the distance MCT (DMCT) as follows.

Definition 3 (DMCT). *Given a set of nodes $v_1, ..., v_n$ and an input XML tree T, the DMCT T_d of the MCT T_m of nodes $v_1, ..., v_n$ is the tree such that:*

1. *T_d contains $v_1, ..., v_n$;*
2. *T_d contains $lac(v_i, v_j)$, where $v_i, v_j \in [v_1, ..., v_n], i \neq j$; and*
3. *there is an edge labeled with the distance d between each node $v_1, ..., v_n$ and $lac(v_i, v_j)$.*

The DMCTs corresponding to $(1-6)$ are shown in $(8-13)$ as follows:

$$\begin{tikzcd} root \arrow[r, "3"] \arrow[dr, "3"] & l_2 \\ & l_4 \end{tikzcd} \tag{8}$$

$$\begin{tikzcd} p_2 \arrow[r, "2"] \arrow[dr, "2"] & l_4 \\ & l_5 \end{tikzcd} \tag{9}$$

$$\begin{tikzcd} root \arrow[r, "3"] \arrow[dr, "3"] & l_2 \\ & l_5 \end{tikzcd} \tag{10}$$

$$a_1 \xrightarrow{1} l_2 \qquad (11)$$

$$\begin{tikzcd} \mathit{root} \arrow[r, "3"] \arrow[dr, "3"] & l_2 \\ & l_6 \end{tikzcd} \qquad (12)$$

$$\begin{tikzcd} p_2 \arrow[r, "2"] \arrow[dr, "2"] & l_5 \\ & l_6 \end{tikzcd} \qquad (13)$$

However, because the number of DMCTs is the same as the number of MCTs, the problem of exponential explosion in the number of subtrees still cannot be resolved. Next, we define grouped DMCTs as follows.

Definition 4 (GDMCT). *A grouped DMCT (GDMCT) T_g contains the DMCT T_d if T_d and T_g are isomorphic. Assume $\mathcal{M}$ is the mapping of the nodes in T_d to those in T_g, and $\mathcal{M}'$ is a corresponding mapping of the edges of T_d to those of T_g, which must meet the following conditions.*

1. *If v_d and v_g are nodes of T_d and T_g, respectively, and $\mathcal{M}(v_d) = v_g$, then the label of v_g contains the ID of v_d.*
2. *If e_d and e_g are edges of T_d and T_g, respectively, and $\mathcal{M}'(v_d) = v_g$, then the labels of v_d and v_g are the same.*

For example, the following GDMCTs (14), (15) and (16) contain DMCTs $(8, 10, 11)$, $(9, 13)$ and (12), respectively.

$$\begin{tikzcd} \mathit{root} \arrow[r, "3"] \arrow[dr, "3"] & {[l_2, l_5]} \\ & {[l_2, l_4, l_6]} \end{tikzcd} \qquad (14)$$

$$\begin{tikzcd} p_2 \arrow[r, "2"] \arrow[dr, "2"] & {[l_5]} \\ & {[l_4, l_5]} \end{tikzcd} \qquad (15)$$

$$a_1 \xrightarrow{1} [l_2] \qquad (16)$$

Therefore, grouping the DMCTs into GDMCTs can effectively reduce the number of results. However, there might be some GDMCTs with roots that are ancestors of other GDMCTs. The lowest GDMCT rooted at the smallest LCA does not contain any roots of other GDMCTs. The following gives the definitions of the smallest LCA and lowest GDMCT.

Definition 5 (Smallest LCA). *Given a set of nodes $v_1, ..., v_n$ in an input tree T, the smallest LCA of $v_1, ..., v_n$ is the node v such that:*

1. *v is the LCA of $v_1, ..., v_n$; and*
2. *v is not an ancestor of any other LCAs of $v_1, ..., v_n$.*

Definition 6 (Lowest GDMCT). *Given a set of nodes $v_1, ..., v_n$ in an input tree T, a GDMCT is a lowest GDMCT if it is rooted at the smallest LCAs of $v_1, ..., v_n$.*

For the same example we used before, the lowest GDMCTs of GDMCT (14), (15) and (16) are as follows:

$$\begin{tikzcd} p_2 \arrow[r, "2"] \arrow[dr, "2"] & {[l_5]} \\ & {[l_4, l_5]} \end{tikzcd} \qquad (17)$$

$$a_1 \xrightarrow{1} [l_2] \qquad (18)$$

4 LCA Detection Method

In this section, we first consider two straightforward methods for detecting LCAs of nodes containing query keywords: the method using keyword B$^+$trees with Dewey-order labels and the method based on superimposed codes. However, the query cost is expensive using these methods because both require traversal of all the nodes of the XML tree. To reduce the query cost, we propose an efficient method combining both methods. We also present an effective solution for the false drop problem caused by the superimposed code.

4.1 Keyword B$^+$tree with the Dewey-order Label Method

Dewey order assigns a vector to each node representing the path from the root of the tree to the node. The containment relationship (parent-child and ancestor-descendant relationships) between two nodes can be conveniently and simply detected by the path: the common ancestor of a set of nodes can be found by comparing the Dewey-order labels of the nodes. For example, assume a query by keywords (*Hamlet*, *King*) in the XML tree of Figure 1. The common ancestor of the nodes $[l_1, 1.1.1.1]$ and $[l_2, 1.1.2.1]$ that contain the query keywords can be found by their Dewey-order labels: their common ancestor is the node $[p_1, 1.1]$, which has the common label of $[l_1, 1.1.1.1]$ and $[l_2, 1.1.2.1]$.

In this method, each query keyword is assigned to a B$^+$tree index, and each entry of the B$^+$tree stores all the leaf nodes that contain the keyword, together with their Dewey-order labels. For example, the XML tree of Figure 1 can be transformed into the B$^+$tree shown in Figure 2. In the query phase, assume a query with keywords (*Hamlet*, *King*). We scan the keyword B$^+$tree for each query keyword to find the corresponding leaf nodes. In this example, the corresponding node for keyword *Hamlet* is $[a_1, 1.1.1.1]$, and those for keyword *King* are $[a_2, 1.1.2.1]$, $[a4, 1.2.1.1]$ and $[a6, 1.2.3.1]$. Therefore, the LCA for the two query keywords is the node $[s_1, 1.1]$, because of the common label between node a_1 and a_2. This method is effective for finding the LCAs, but it results in expensive query cost, as we will discuss in Section 5.2.

4.2 Superimposed Code-based Method

Signature file partitioning techniques based upon superimposed codes are widely applied in such research areas as information retrieval and data security. Assume the size of the signature file F is S; according to superimposed coding, each query keyword yields a word signature, i.e., a bit sequence of size S. These bit sequences are OR-ed together to form the signature file F. To create a word signature, each word is hashed to m bit positions in the range $1 - S$. The corresponding bits are set to "1", while all the other bits are set to "0". For example, consider the two files F_1 and F_2 of Table 1. The signature of each file can be generated by OR-ing the word signatures of all the keywords.

Given a query signature Q, for the signature S_i of file F_i, if $Q \wedge S_i = Q$, F_i is a candidate of the query result, which is called a *drop*. However, some drops

F_1		F_2	
Keyword	Signature	Keyword	Signature
Lear	1000001	Hamlet	0100001
King	0100010	King	0100010
Duke	0101000	Mother	0100100
Brother	1100000	Brother	1100000
File signature	1101011	File signature	1100111

Table 1. Examples of file signatures

Query keywords	Query signature	F_1	F_2
King, brother	1100010	actual drop	actual drop
King, mother	0100110	no match	actual drop
Lear, King	1100011	actual drop	false drop

Table 2. Examples of drops

actually do not correspond to all the query keywords. These drops are called *false drops*, while the drops that actually satisfy the query predicate are called *actual drops*. Table 2 shows example drops for different queries.

In this section, we propose an LCA detection method based on superimposed codes. Assume the set of keywords of an XML document is K and the keywords in the leaf node KN_i are $k_j \in K(1 \leq j \leqq |KN_i|)$. The superimposed code S_i for the leaf node KN_i can then be calculated by OR-ing the signatures of the keywords $S(k_j)$:

$$S_i = S(k_1) \vee S(k_2) \vee \ldots \vee S(k_{|KN_i|}). \tag{19}$$

Next, the superimposed code S_{pi} of the parent node PN_i can be computed by:

$$S_{pi} = S_{c1} \vee S_{c2} \vee \ldots \vee S_{cm}, \tag{20}$$

where $S_{c1}, S_{c2}, \ldots, S_{cm}$ denotes the signatures of the child nodes $CN_1, CN_2, \ldots, CN_m$ of PN_i.

In the query, the query signature Q is determined by OR-ing the signature of each query keyword. Then, we investigate whether the query signature and each node signature NS_i satisfy the condition:

$$Q \wedge NS_i = Q. \tag{21}$$

All the nodes satisfying the above condition are candidate LCAs that may contain all the query keywords. However, there may be some false drops in the candidate LCAs.

4.3 Combined Method

The B$^+$tree with the Dewey-order label and superimposed code-based methods are effective in finding the LCAs. However, they result in expensive query costs. To reduce the query cost, in this section we propose an efficient method that combines B$^+$trees with Dewey-order labels and superimposed codes.

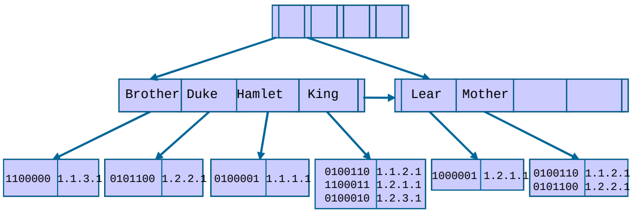
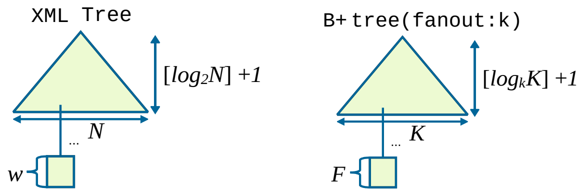


Fig. 3. An example keyword B+tree with superimposed codes



(a)XML tree (b) Keyword B$^+$tree

Fig. 4. Notations for query cost comparison

In the combined method, each node of the XML tree is assigned a superimposed code as in the original superimposed code-based method, and then both the superimposed code and the Dewey-order labels of nodes are stored in a keyword B$^+$tree, as shown in Figure 3. Assume a query with keywords $k_1, \ldots, k_m$. The query signature Q is determined by OR-ing the signature of each keyword $S_i, 1 \leq i \leq m$, namely, $Q = S_1 \vee S_2) \vee \ldots, \vee S_m$. The candidate LCAs can be determined by scanning the B$^+$tree with any keyword $k_i, 1 \leq i \leq m$ of the query, if the query signature Q and the node signature NS corresponding to the keyword satisfy the condition $Q \wedge NS = Q$. However, the candidate results may still have false drops.

We present an effective resolution for the false drop problem. For any keyword $k_i, 1 \leq i \leq m$ in the query, it is evident that the node with signature NS in the B$^+$tree satisfying $Q \wedge NS = Q$ must contain the keyword k_i itself. For $1 \leq i \leq m$, we can obtain the node sets $N_1, N_2, \ldots, N_m$ for each query keyword $k_i, 1 \leq i \leq m$ by using the condition $Q \wedge NS = Q$. Each node set $N_i, 1 \leq i \leq m$ must contain the corresponding keyword $k_i, 1 \leq i \leq m$. Assume the Dewey-order label sets of $N_i, 1 \leq i \leq m$ are $L_i, 1 \leq i \leq m$. The nodes that have the common label among $L_i, 1 \leq i \leq m$ are the LCAs that must contain all the query keywords. The algorithm for false drop resolution is shown in Figure 5.

```
getLCA($w_1, \dots w_k$){
    $Q = H(w_1) \vee \dots H(w_k)$;
    For ($i = 0$ to $k$){
        $NS_i$=getNodeSignature($w_i$);//Get the word signature of $w_i$ from the keyword B$^+$tree
        $L_i$=getResultSI($Q$, $NS_i$);
    }
     Return getTrueDropLCA($L_1, \dots L_k$);
}
getResultSI($Q$, $NS$){
    L=Null;
    For ($i = 0$ to $|NS|$){
        If ($Q \wedge NS(i) = Q$){
            L.add($NS(i)$);
        }
    }
    Return L;
}
getTrueDropLCA($L_1, \dots L_k$){
    N.addAll($L_1$);
    For ($i = 1$) to $|L_1|$}
        For ($j = 1$ to $k$){
            For ($s = 1$ to $|L_j|$){
                If($L_1(i) = L_j(s)$){
                    Break;
                }
                If($j = |L_i|$){
                    $N.remove(L_1(i))$;
                }
            }
        }
    }
    Return N;
}
```

Fig. 5. False drop resolution algorithm

```
getLowestGDMCT($l_1, \dots l_n, KL_1, \dots KL_k$){
    Assume L represents the list of ($l_1, \dots l_n$) and G represents the list of GDMCT
    L=getSmallestLCA($l_1, \dots l_n, KL_1, \dots KL_k$);
    For ($i = 0$ to $n$){
        G=getGDMCT($l_1, \dots l_n, KL_1, \dots KL_k, i$);
        Return (L, G);
    }
}
getSmallestLCA($l_1, \dots l_n, KL_1, \dots KL_k$){
    L=Null;
    For ($i = 0$ to $n$){
        For ($j = 0$ to $k$){
            If ($i! = j$){
                If ($l_j$.substring($l_i$)){
                    Break;
                }
                If ($j = n$){
                    L.add($l_i$);
                }
            }
        }
    }
    Return G;
}
getGDMCT($KL_1, \dots KL_k, n$){
    For ($i = 0$ to $k$){
        For ($j = 0$ to $|KL_i|$){
            If($KL_i(j).subtring(n)$){
                $G \leftarrow KL_(j)$;
            }
        }
    }
    Return G;
}
```

Fig. 6. Lowest GDMCT detection algorithm

5 Extracting Lowest GDMCTs

5.1 KBDLM and KBSIM

In this section, we present two methods for computing the lowest GDMCTs based on the keyword B$^+$tree with Dewey-order labels method and the combined method, which are called the keyword B$^+$tree with Dewey-order label method (KBDLM) and the keyword B$^+$tree with superimposed code method (KBSIM), respectively. In the KBDLM and KBSIM, the LCAs are first determined by the keyword B$^+$tree with Dewey-order labels or the superimposed codes. Then, the smallest LCAs, i.e., the root of the lowest GDMCTs, can be found by comparing the Dewey-order labels among the LCAs. The length of each edge can be computed by comparing the length between the label of the root and that of each leaf node. The lowest GDMCT detection algorithm is illustrated by Figure 6.

5.2 Query Cost Comparison

In this section, we evaluate the query cost for the proposed methods, KBDLM and KBSIM, and the SA algorithm. Firstly, we give some notation for query cost comparison shown in Table 3; some parameters are illustrated in Figure 4.

Query Cost of the SA Algorithm According to reference [10], the query cost of the SA algorithm is:

$$O(log_2 N * F^{2m}) = O(logN * N^{2m}). \tag{22}$$

notation	description	notation	description
m	number of query keywords	K	number of keyword
k	fanout (node entries) of B$^+$tree	N	number of leaf nodes in the XML tree
w	mean number of keywords in each node	F	mean number of nodes for each keyword
b	comparison cost for one bit	B	comparison cost for one string
r	mean number of strings of the labels	l	bit length of the signature
n	mean string length of the query keywords	d	mean length of the Dewey-order labels

Table 3. Notation for query cost comparison

Query Cost of the KBDLM In the KBDLM, the height of the keyword B$^+$tree is $[log_k K] + 1$, as shown in Figure 4 (b). In the worst case, with one query keyword, it is necessary to scan all the entries of the B$^+$tree. The number of required comparisons for finding the nodes containing the keyword is $m \times k \times ([log_k K] + 1)$. Therefore, the cost of finding the nodes containing all the m keywords is:

$$m \times k \times ([log_k K] + 1) \times n \times B. \tag{23}$$

The cost for finding the LCAs by comparing their Dewey-order labels is:

$$F^{m-1} \times d \times B. \tag{24}$$

In the worst case, the number of parents of the lowest GDMCTs is N, so the cost for computing the lowest GDMCT is $N \times m \times F$. Therefore, the total query cost of KBDLM is the sum of the above costs:

$$m \times k \times ([log_k K] + 1) \times n \times B + F^{m-1} \times d \times B + N \times m \times F. \tag{25}$$

F can be represented by N and K, because $F = \frac{wN}{K}$. Therefore, the query cost of the KBDLM can be expressed by the following equation in terms of the order of N:

$$O((\frac{wN}{K})^{m-1} + (\frac{wN^2}{K})) = O(N^{m-1} + N^2). \tag{26}$$

When $m \leq 3$, the cost is $O(N^2)$; when $m \geq 4$, it is $O(N^{m-1})$.

Query Cost of the KBSIM In the KBSIM, the cost for finding the nodes containing the keywords is the same as for the KBDLM. Then, the cost for detecting the LCAs of the nodes containing the keywords is $b \times l \times (F - 1) \times m$. Next, in the worst case, the cost of finding the false drops in the detected LCAs is $(m - 1)N^2$. Finally, the cost for computing the lowest GDMCTs is the same as the KBDLM, $N \times m \times F$. Therefore, the total query cost of the KBSIM can be calculated by:

$$\begin{aligned} m \times k \times ([log_k K] + 1) \times n \times B + b \times l \times (F - 1) \times m \\ +(m - 1)N^2 + N \times m \times F. \end{aligned} \tag{27}$$

CPU	AMD Opteron 2.2 GHz
Memory	6 GB
OS	Linux version 2.6.9
Database	PostgreSQL 8.1.3
Hard Disk	DRAILD
Java	1.5.0_07

Table 4. Experimental environment

	1 MB, 2 keywords	5 MB, 2 keywords
Original method	$\mathcal{R} = 7.6\%$	$\mathcal{R} = 10.3\%$
Combination method	$\mathcal{R} = 0\%$	$\mathcal{R} = 0\%$

Table 5. False drop rate of detected LCAs

The total cost of the KBSIM is $O(N^2)$ by substituting $F = \frac{wN}{K}$ in the above equation. Therefore, when $m > 2$, the proposed combined method KBSIM has lower costs than either the KBDLM or SA algorithms.

6 Experimental Evaluation

We conducted experiments to evaluate the benefits of the proposed combined method, KBSIM. Firstly, we show that the KBSIM can completely solve the false drop problem in contrast to the original superimposed code-based method. Then, we compare the execution time for keyword queries using the KBSIM with that using the KBDLM and SA algorithms with different numbers of query keywords and different sizes of XML documents.

The experiments were performed in the environment shown in Table 4. The XML data collections used in the experiments were generated by *xmlgen* of the XMark benchmark [16]. Three kinds of data were generated by using different scaling factors of 0.01, 0.05 and 0.1, respectively. The sizes of the generated XML data are about 1 MB, 5 MB and 10 MB, respectively. The experiments were performed using sets of keywords having different frequencies introduced in [10]. Namely, *low*, corresponding to keywords with frequencies between 1 and 10, *medium*, corresponding to keywords with frequencies 11–200, and *high*, corresponding to keywords with frequencies greater than 200.

6.1 Evaluating False Drop Resolution

We applied the original superimposed code-based method and the combined method with false drop resolution to detect the LCAs using the XMark data of 1 MB and 5 MB with two query keywords. Table 5 shows the false drop rates $\mathcal{R}$ of the detected LCAs. We can see that the proposed combined method can completely solve the false drop problem in LCA detection caused by the original superimposed code-based method.

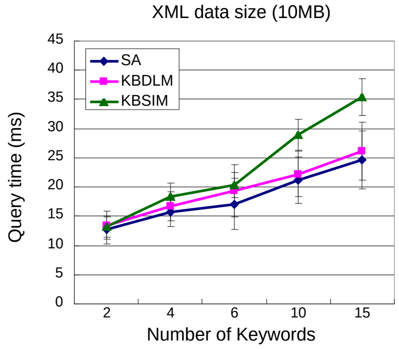


Fig. 7. Query time for keywords with low frequency

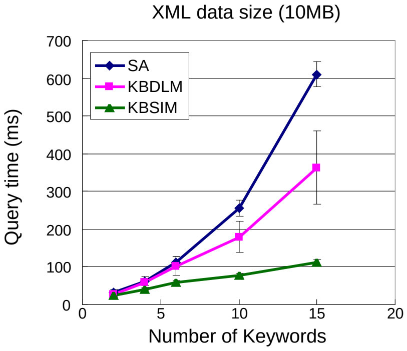


Fig. 8. Query time for keywords with medium frequency

6.2 Evaluating Query Performance

To evaluate the query performance of the proposed algorithms, we firstly performed experiments to observe and compare the query time with different numbers of query keywords using the KBSIM, KBDLM and SA algorithms. The size of the XMark data used in the experiments was 10 MB, and the keyword frequencies ranged from *low* to *high*. Figure 7 presents the performance of each method as the number of keywords increases for keywords at low frequencies. It shows that the SA performs slightly better than both proposed algorithms for the low-frequency keywords. Figure 8 shows the query time as the number of keywords with medium frequency increases. This figure indicates that both the proposed algorithms are superior to the SA algorithm, and the KBSIM performs better than the KBDLM at these keyword frequencies. Figure 9 presents the results of query time as the number of keywords at high frequencies increases. It is evident that KBSIM is overwhelmingly superior to the other two methods. However, the query time using the KBDLM increases extremely quickly when the number of high-frequency keywords is more than four[6].

We next conducted experiments to observe the performance of the KBSIM and SA algorithms with different sizes of XML documents. We measured the query times for XMark data of 1 MB, 5 MB and 10 MB with four high-frequency query keywords. Figure 10 represents the query time of the two methods for each XML document. This figure shows that as the XML document becomes larger, the proposed KBSIM performs more efficiently than the SA algorithm.

7 Conclusion and Future Work

[6] The query time using the KBDLM for more than four keywords is not shown in Figure 9.

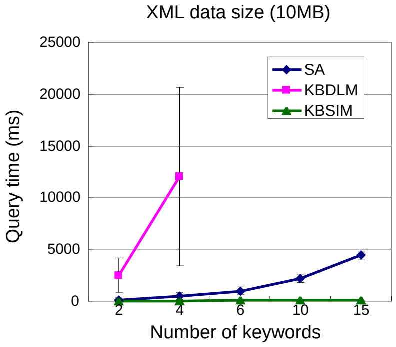


Fig. 9. Query time for keywords with high frequency

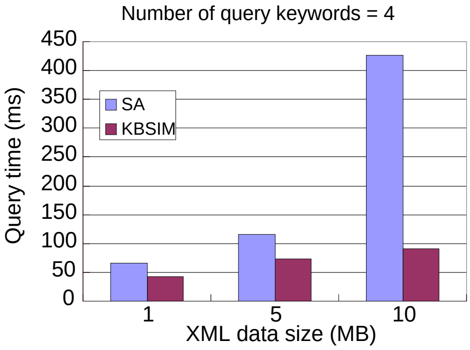


Fig. 10. Query time for different sizes of XML data

With the exponential increase in the amount of XML data on the Internet, information retrieval techniques on tree-structured XML documents such as keyword search become important. In keyword searches over XML documents, the search results are often represented by MCTs rooted at the LCAs of the nodes containing all the query keywords. In this paper, we first considered two straightforward LCA detection methods: keyword B^+trees with Dewey-order labels and superimposed code-based indexing methods. Then, we proposed a method for efficiently detecting the LCAs, which combines the two straightforward indexing methods. We also presented an effective resolution for the false drop problem caused by the superimposed codes. Finally, we applied the proposed LCA detection method to generate the lowest GDMCTs over XML documents. We also conducted detailed experiments to evaluate the benefits of our proposed algorithms and to show that the proposed combined method can completely solve the false drop problem and outperform the previously known stack-based algorithm in extracting the lowest GDMCTs.

In the future, we plan to consider more sophisticated keyword searches over XML documents such as the user context-based keyword search. We will also work on the issue of search result ranking so that we can provide more effective search results to the users.

Acknowledgment

This work was partially supported by the Grant-in-Aid for Scientific Research of MEXT Japan #16016232, by CREST of JST (Japan Science and Technology Agency) and by the TokyoTech 21COE Program "Framework for Systematization and Application of Large-Scale Knowledge Resources".

References

1. Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *ICDE*, pages 5–16, 2002.
2. Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. DBXplorer: Enabling Keyword Search over Relational Databases. In *SIGMOD*, page 627, 2002.
3. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. On Finding Lowest Common Ancestors in Trees. In *STOC*, pages 253–265, 1973.
4. Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *ICDE*, pages 431–440, 2002.
5. Andrea E. F. Clementi, Angelo Monti, and Riccardo Silvestri. Selective Families, Superimposed Codes, and Broadcasting on Unknown Radio Networks. In *SODA*, pages 709–718, 2001.
6. Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. XSEarch: A Semantic Search Engine for XML. In *VLDB*, pages 45–56, 2003.
7. M. Dyer, T. Fenner, A. Frieze, and A. Thomason. On Key Storage in Secure Networks. *J. of Cryptology*, 8(4):189–200, 1995.
8. Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *SIGMOD*, pages 16–27, 2003.
9. D. Harel and R. E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
10. Vagelis Hristidis and Nick Koudas. Keyword Proximity Search in XML Trees. *IEEE TKDE*, 18(4):525–539, April 2006.
11. Vagelis Hristidis and Yannis Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *VLDB*, pages 670–681, 2002.
12. Rasmus Kaae, Thanh-Duy Nguyen, Dennis Nørgaard, and Albrecht Schmidt. Kalchas: A Dynamic XML Search Engine. In *CIKM*, pages 541–548, 2005.
13. W. H. Kautz and R. C. Singleton. Nonrandom Binary Superimposed Codes. *IEICE Trans. Inform. Theory*, 10(4):363–377, 1964.
14. Yunyao Li, Cong Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, pages 72–83, 2004.
15. Matti Nykänen and Esko Ukkonen. Finding Lowest Common Ancestors in Arbitrarily Directed Trees. *Inf. Process. Lett.*, 50(6):307–310, 1994.
16. XML Benchmark Project. http://www.xml-benchmark.org.
17. B. Schieber and U. Vishkin. On Finding Lowest Common Ancestors: Simplification and Parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.
18. Albrecht Schmidt, Martin L. Kersten, and Menzo Windhouwer. Querying XML Documents Made Easy: Nearest Concept Queries. In *ICDE*, pages 321–329, 2001.
19. D. R. Stinson, Tran van Trung, and Wei R. Secure Frameproof Codes, Key Distribution Patterns, Group Testing Algorithms and Related Structures. *J. of Statistical Planning and Inference*, 86:595–617, 2000.
20. Swiss-Prot. http://www.ebi.ac.uk/swissprot/.
21. R. E. Tarjan. Applications of Path Compression on Balanced Trees. *J. ACM*, 26(4):690–715, 1979.
22. TrEMBL. http://www.ebi.ac.uk/trembl/.
23. The Free Encyclopedia: Wikipedia. http://www.wikipedia.org/.
24. XML Version of DBLP. Available at http://dblp.uni-trier.de/xml/.
25. Yu Xu and Yannis Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *SIGMOD*, pages 537–538, 2005.