Information Systems 35 (2010) 186–203

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Finding and ranking compact connected trees for effective keyword proximity search in XML documents

Jianhua Feng, Guoliang Li *, Jianyong Wang, Lizhu Zhou

Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 10084, China

ARTICLE INFO

Article history:
Received 3 April 2008
Received in revised form
20 April 2009
Accepted 31 May 2009
Recommended by: D. Suciu

Keywords:
Lowest common ancestor (LCA)
Compact LCA (CLCA)
Maximal CLCA (MCLCA)
Compact connected trees (CCTrees)
Maximal CCTrees (MCCTrees)

ABSTRACT

In this paper, we study the problem of keyword proximity search in XML documents. We take the disjunctive semantics among the keywords into consideration and find *top-k* relevant compact connected trees (CCTrees) as the answers of keyword proximity queries. We first introduce the notions of *compact lowest common ancestor* (CLCA) and *maximal CLCA* (MCLCA), and then propose *compact connected trees* and *maximal CCTrees* (MCCTrees) to efficiently and effectively answer keyword proximity queries. We give the theoretical upper bounds of the numbers of CLCAs, MCLCAs, CCTrees and MCCTrees, respectively. We devise an efficient algorithm to generate all MCCTrees, and propose a ranking mechanism to rank MCCTrees. Our extensive experimental study shows that our method achieves both high efficiency and effectiveness, and outperforms existing state-of-the-art approaches significantly.

1. Introduction

Traditional query processing approaches in XML documents are constrained by the query constructs imposed by query languages such as XPath and XQuery. Firstly, these query languages are hard to comprehend for non-database users. For example, the XQuery is fairly complicated to grasp. Secondly, these languages require the queries to be posed against the underlying, sometimes complex, database schemas. These traditional querying methods are powerful but unfriendly to non-expert users. Fortunately, keyword search is proposed as an alternative means for querying XML databases, which is simple and yet familiar to most internet users as it only requires the input of keywords. Keyword search is a proven and widely accepted mechanism for querying in textual document systems and World Wide Web. Database research community has recently recognized the benefits of keyword search and has been introducing keyword search capability into relational databases [1,4,7,12,14,17,27,30,21], XML databases [2,3,6,9,13,15,18,19,24–26,28,31,32,37,38], graph databases [11,16], and heterogenous data sources [20,22].

Given an XML document and a keyword query, although the whole XML document could be taken as the answer of the keyword query if it contains all input keywords, it may be too large and frustrate users. For example, to search for interested references from the DBLP dataset, users are usually interested in a portion of the XML document that is relevant to the query, as opposed to the whole document which is more than 470 MB.

To address this problem, the notion of *lowest common ancestor* (LCA) has been introduced to answer keyword queries in XML documents [9]. Existing methods [6,9,37] first compute LCAs (or their variants) of *nodes* that contain input keywords and then construct the subtrees rooted at LCAs to answer keyword queries. Most of existing approaches consider the conjunctive semantics (AND) among the input keywords. When there are few answers that contain all input keywords, they lead to ineffectiveness. It is

* Corresponding author.
E-mail address: liguoliang@tsinghua.edu.cn (G. Li).

0306-4379/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2009.05.004

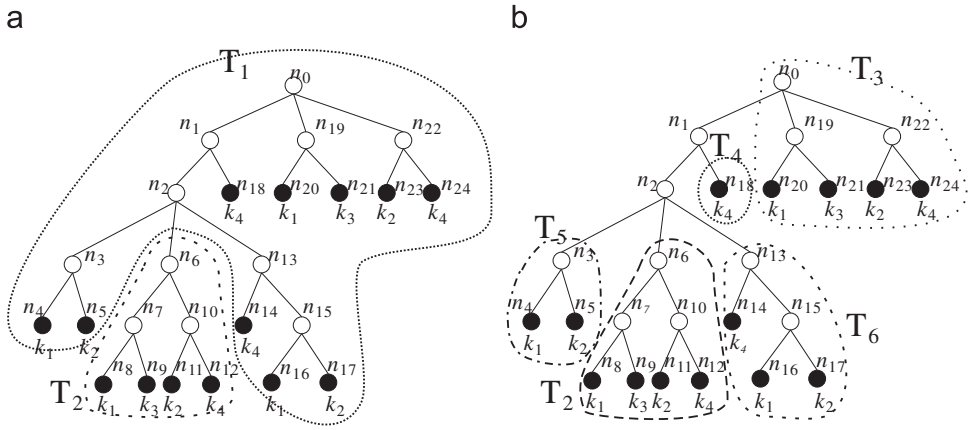

Fig. 1. An example of keyword search in XML documents. (a) Existing method. (b) Our method.

not straightforward to extend existing semantics to support disjunctive semantics (OR). Moreover, existing search semantics cannot find the most relevant subtrees to answer keyword queries. For example, consider the XML document in Fig. 1. Suppose a user types in a query "k_1, k_2, k_3, k_4". Existing methods [9] identify the two subtrees rooted at n_0 and n_6 as the answers (Fig. 1(a)). Note that T_1 contains large numbers of nodes, and most of nodes in the subtree are irrelevant as they have large distances. To address this problem, we propose a new search semantics of *compact lowest common ancestor* (CLCA) to effectively answer keyword proximity queries. We identify the most relevant *compact connected trees* (CCTrees) rooted at CLCAs as answers. In this example, we identify five *compact* connected trees as the answers (Fig. 1(b)).

In this paper, we propose a new search semantics to answer keyword queries in XML documents. We identify compact connected trees as answers. We devise a new ranking mechanism by considering the structural compactness of tree-structure answers for returning the most relevant ones with the highest ranks. To summarize, we make the following contributions:

- We introduce a new search semantics of *compact lowest common ancestor* to answers keyword queries in XML documents. We find compact connected trees rooted at CLCAs as the answers.
- We devise an efficient algorithm to adaptively generate CCTrees. We give the theoretical minimum upper bounds of the numbers of CLCAs and CCTrees, respectively.
- We propose an effective ranking mechanism to rank CCTrees for returning the most relevant ones with the highest ranks.
- We have conducted an extensive performance study using real datasets and various queries with different characteristics. The results show that our method achieves both high efficiency and effectiveness, and outperforms existing state-of-the-art approaches significantly.

The rest of this paper is organized as follows. We introduce a new semantics to answer keyword queries in Section 2. Section 3 proposes an efficient algorithm to generate CCTrees and Section 4 gives a new ranking mechanism. Extensive experimental evaluations are provided in Section 5. We review the existing studies in Section 6 and conclude the paper with Section 7.

2. Compact connected trees

In this section, we propose a new keyword search method for XML documents in terms of the disjunctive semantics. We first introduce a new search semantics to answer keyword proximity queries (Section 2.2) and give the theoretical upper bounds of the number of answers (Section 2.3). Then, we introduce how to compute relevant answers based on our semantics (Section 2.4).

2.1. Notations

An XML document can be modeled as a rooted, ordered and labeled tree. Nodes in this rooted tree correspond to elements in the XML document. For any two nodes u and v, $u \prec v$ ($u \succ v$) denotes that node u is an ancestor (descendant) of node v. $u \preceq v$ denotes that $u \prec v$ or $u = v$. Given a keyword query $\mathcal{K} = \{k_1, k_2, \ldots, k_m\}$ and an XML document $\mathcal{D}$, we use $\mathcal{I}_i$ to denote the list of nodes that contain k_i. We call each node in $\mathcal{I}_i$ a *content node*. $\mathcal{I}_i$ can be retrieved by using the well-known inverted indices. For example, consider the XML document in Fig. 1, we have $\mathcal{I}_{k_1} = \{n_4, n_8, n_{16}, n_{20}\}$.

2.2. Compact lowest common ancestor

Existing methods [6] employed LCA semantics to answer XML keyword queries. We first review the concept of *lowest common ancestor*.

Definition 2.1 (*Lowest common ancestor*). Given m nodes, $n_1, n_2, \ldots, n_m$, `ca` is a common ancestor of these m nodes, if `ca` is an ancestor of each node n_i for $1 \le i \le m$. `lca` is the lowest common ancestor of these m nodes, denoted as $\texttt{lca} = LCA(n_1, n_2, \ldots, n_m)$, if `lca` is a common ancestor of

the m nodes and there does not exist a common ancestor of these m nodes, u, and $\texttt{lca} \prec u$.

LCA based methods have two drawbacks: (1) LCA is not meaningful enough to answer keyword queries. For example, in Fig. 1, although n_2 is the LCA of n_4 and n_{17}, it is not as meaningful as the subtree of nodes $\{n_3, n_4, n_5\}$ and the subtree of nodes $\{n_{15}, n_{16}, n_{17}\}$ to answer the query. As the nodes in the latter two subtrees are much more related than nodes $\{n_2, n_4, n_{17}\}$; (2) The subtree rooted at LCA is not compact enough to answer a query. For example, consider the XML document in Fig. 1. Suppose a user types in a query "k_1, k_2, k_3, k_4". Existing methods [9] identify the two subtrees rooted at n_0 and n_6 as the answers (Fig. 1(a)). Note that T_1 contains large numbers of nodes, and most of nodes in the subtree are irrelevant as they have large distances. To address these problems, we introduce a more meaningful concept.

Definition 2.2 (*Compact lowest common ancestor*). Given q content nodes, $v_1 \in \mathscr{I}_1$, $v_2 \in \mathscr{I}_2, \ldots, v_q \in \mathscr{I}_q$. $w = LCA(v_1, v_2, \ldots, v_q)$ is said to dominate v_i w.r.t. $\{k_1, k_2, \ldots, k_q\}$, if $w \geq LCA(v'_1, \ldots, v'_{i-1}, v_i, v'_{i+1}, \ldots, v'_q)$,$\forall$ $v'_1 \in \mathscr{I}_1$, $v'_2 \in \mathscr{I}_2, \ldots, v'_{i-1} \in \mathscr{I}_{i-1}$, $v'_{i+1} \in \mathscr{I}_{i+1}, \ldots, v'_q \in \mathscr{I}_q$. w is a CLCA w.r.t. $\{k_1, k_2, \ldots, k_q\}$, if w dominates each v_i for $1 \leq i \leq q$.

A CLCA is a non-trivial LCA, which is the lowest common ancestor of more relevant nodes, while the irrelevant nodes cannot share a CLCA. For example, recall the above example. For query $\{k_1, k_2\}$, n_3 dominates n_4 and n_5, and n_{15} dominates n_{16} and n_{17}. Thus n_3 and n_{15} are CLCAs. However, n_2 does not dominate n_4 and n_{17}, as n_5 is much more related to n_4 than n_{17}, and n_{16} is much more related to n_{17} than n_4. In the other words, the most related nodes are dominated by the same CLCA, while the irrelevant nodes are not.

CLCA is different from SLCA [37]. A LCA is a SLCA if it has no LCA descendants. For example, in Fig. 1, n_0 and n_6 are both CLCAs w.r.t. $\{k_1, k_2, k_3, k_4\}$, and they dominate $\{n_{20}, n_{21}, n_{23}, n_{24}\}$ and $\{n_8, n_9, n_{11}, n_{12}\}$, respectively. n_0 is not a SLCA as n_0 has a LCA descendant n_6, hence n_0 is a false negative for SLCA. Moreover, the false negative problem is not an *ad hoc* problem but ubiquitous over the XML documents with nested structures. CLCA can avoid those false negatives and thus is a more meaningful methodology to answer keyword queries. Furthermore, we give the minimum upper bound of the number of CLCAs in terms of the conjunctive semantics, i.e., $\min(|\mathscr{I}_1|, |\mathscr{I}_2|, \ldots, |\mathscr{I}_m|)$, as formalized in Lemma 2.1, which is much smaller than the number of LCAs.

Lemma 2.1. *There are at most* $\min(|\mathscr{I}_1|, |\mathscr{I}_2|, \ldots, |\mathscr{I}_m|)$ *CLCAs w.r.t. a keyword query* $\mathscr{K} = (k_1, k_2, \ldots, k_m)$ *for the conjunctive semantics.*

Proof. For each content node $n \in \mathscr{I}_i$ $(1 \leq i \leq m)$, there is at most one $\texttt{clca}$ that dominates n in terms of conjunctive semantics. Accordingly, the number of CLCAs is no more than the size of the minimal keyword list, which contains the fewest content nodes. □

Most of existing studies assume the conjunctive semantics (AND) among input keywords, however, it will lead to low effectiveness if there are only a few results that contain all keywords. Especially, when considering keyword proximity search which returns the *top-k* answers, we had better rank all relevant answers, even if some answers do not contain all keywords, and then return the *top-k* answers with the highest ranks. In this paper, we investigate keyword search in XML documents with disjunctive semantics (OR). In the rest of this paper, we refer to keyword search as keyword proximity search with disjunctive semantics. Here we give the upper bound of the number of CLCAs.

Lemma 2.2. *Suppose* $|\mathscr{I}_1| \leq |\mathscr{I}_2| \leq \cdots \leq |\mathscr{I}_m|$. *There are at most* $\sum_{i=1}^{m} 2^{m-i}|\mathscr{I}_i|$ *CLCAs for* $\mathscr{K} = (k_1, k_2, \ldots, k_m)$ *in considering the disjunctive semantics.*

Proof. Let $\texttt{CLCASet}^i_{dis}$ denote the set of all the CLCAs, which dominate exactly i content nodes in i different keyword lists, we have,

$$\begin{aligned}|\texttt{CLCASet}^i_{dis}| &\leq C^{i-1}_{m-1}|\mathscr{I}_1| + C^{i-1}_{m-2}|\mathscr{I}_2| \\ &\quad + \cdots + C^{i-1}_{m-(m-i+1)}|\mathscr{I}_{m-i+1}| \\ &= \sum_{j=1}^{m-i+1} C^{i-1}_{m-j}|\mathscr{I}_j|,\end{aligned}$$

where C^{i-1}_{m-j} is the number of combinations that we do not select keywords $k_1, k_2, \ldots, k_{j-1}$, but select k_j and other $i-1$ keywords from $\{k_{j+1}, k_{j+1}, \ldots, k_m\}$. In this case, there are at most $|\mathscr{I}_j|$ CLCAs for each of these combinations. We give the theoretical upper bound of the number of CLCAs,

$$\begin{aligned}|\texttt{CLCASet}_{dis}| &\leq \sum_{i=1}^{m} |\texttt{CLCASet}^i_{dis}| \leq \sum_{i=1}^{m}\sum_{j=1}^{m-i+1} C^{i-1}_{m-j}|\mathscr{I}_j| \\ &= \sum_{i=1}^{m}\sum_{j=1}^{m} C^{j}_{m-i}|\mathscr{I}_i| = \sum_{i=1}^{m} 2^{m-i}|\mathscr{I}_i|. \qquad \square\end{aligned}$$

As the number of keywords in $\mathscr{K}$ (m) is usually very small, the number of CLCAs is proportional to the number of content nodes.

2.3. Compact global tree

To derive the minimum upper bound of the number of CLCAs, we introduce another notion, *compact global tree*, which will also be used to generate compact connect trees to answer keyword queries in Section 2.4.

As discussed in Section 2.1, we model XML documents as tree structures. Given two subtrees of an XML document, $T_1\langle r, V_1, E_1\rangle$ and $T_2\langle r, V_2, E_2\rangle$, where r is the root, V_1 and V_2 are the node sets, and E_1 and E_2 are the edge sets of T_1 and T_2, respectively. $T\langle r, V, E\rangle$ is the joined tree of T_1 and T_2, where $V = V_1 \cup V_2$ and $E = E_1 \cup E_2$. Given a keyword query $\mathscr{K} = \{k_1, k_2, \ldots, k_m\}$ and an XML document. Let r be the LCA of all content nodes in each $\mathscr{I}_i$. The tree rooted at r and containing the nodes and edges in the path from r to $v_{i_j} \in \mathscr{I}_i$ for $1 \leq i \leq m$ is called a global tree. Nodes in the global tree except the content nodes are called structural nodes. The structural nodes that have

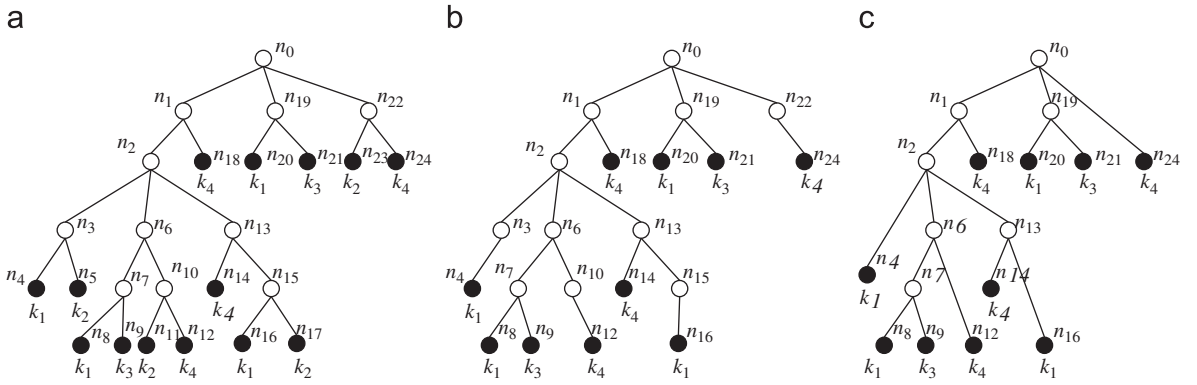

Fig. 2. An XML document and its global tree and CGTree. (a) is an XML document, (b) is the global tree for $\mathscr{K} = \{k_1, k_3, k_4\}$, and (c) is the CGTree.

more than one child are called branch nodes, and those having only one child are called linked nodes. Fig. 2 illustrates an example.

Lemma 2.3. *Given a query $\mathscr{K}$ and an XML document $\mathscr{D}$, $\mathscr{G}\langle r, V, E\rangle$ is the global tree w.r.t. $\mathscr{K}$ and $\mathscr{D}$, where $V = B \cup C \cup L$, B is the branch node set, C is the content node set, and L is the linked node set. Each CLCA* `clca` *is either a branch node or a content node in $\mathscr{G}$, i.e.,* `clca` $\in B \cup C$.

Proof. Firstly, we prove that `clca` $\in V$. Since all LCAs of content nodes must be in V and `clca` is also a non-trivial LCA, and thus `clca` $\in V$. Secondly, we prove that `clca` $\neq L$ by contradiction. Suppose `clca` $\in L$. As each node in L has only one child based on the definition of linked nodes, `clca` has only one child. Without loss of generality, suppose c is the child of `clca`, c will dominate those nodes which are also dominated by `clca`, hence `clca` cannot be a CLCA. This contradicts that `clca` is a CLCA. Hence, we have `clca` $\notin L$ and `clca` $\in B \cup C$. □

The global tree is a subtree of the XML document, which contains all the content nodes and their ancestors but without other irrelevant nodes. More importantly, the global tree is easier to be manipulated than the whole XML document. Furthermore, CLCAs must be the branch nodes or content nodes in the global tree, and Lemma 2.3 guarantees the correctness. As CLCAs in the global tree cannot be linked nodes, here we introduce a more compact form of the global tree by eliminating the linked nodes.

Definition 2.3 (*Compact global tree*). Given a global tree, $\mathscr{G}\langle r, V, E\rangle$, $\mathscr{CG}\langle r', V', E'\rangle = \varphi(\mathscr{G})$ is called a compact global tree, if mapping function φ satisfies the following conditions:

(i) $r' = \varphi(r) = r$;
(ii) $V' = \varphi(V) = B \cup C$, where $V = B \cup C \cup L$, B is the branch node set, C is the content node set, and L is the linked node set in $\mathscr{G}$; and
(iii) $E' = \varphi(E) = \{(a, d) | a, d \in B \cup C,\ a \prec d$ and $\nexists v \in B \cup C,\ a \prec v \prec d\}$.

As the compact global tree is constructed by eliminating the linked nodes from the global tree, it is more compact and meaningful than the global tree. We will introduce how to answer keyword queries on this tree later. Here, based on the CGTree, we give the minimum upper bound of the number of all CLCAs as formalized in Lemma 2.4.

Lemma 2.4. *There are at most $2 * \sum_{i=1}^{m} |\mathscr{I}_i| - 1$ CLCAs for a keyword query $\mathscr{K} = (k_1, k_2, \ldots, k_m)$ and an XML document $\mathscr{D}$.*

Proof. We prove it by constructing the CGTree. Suppose $\mathscr{CG}$ is the CGTree w.r.t. $\mathscr{K}$ and $\mathscr{D}$. Let n_{CLCA} denote the number of CLCAs, n_l denote the number of leaf nodes, n_b denote the number of branch nodes, n_c denote the number of content nodes and n_{nlc} denote the number of the non-leaf content nodes in $\mathscr{CG}$, that is, the content nodes which are not leaf nodes in $\mathscr{CG}$.[1] It is obvious that $n_c = n_l + n_{nlc}$ and $n_c \leq \sum_{i=1}^{m} |\mathscr{I}_i|$.[2] In addition, we have proved that all the CLCAs must be in $\mathscr{CG}$ (Lemma 2.3), thus $n_{CLCA} \leq n_c + n_b$.

As the out-degree of each branch node in $\mathscr{CG}$ is no smaller than two and that of each non-leaf content node is at least one, $n_o \geq 2 * n_b + n_{nlc}$, where n_o is the total out-degree of all the nodes in $\mathscr{CG}$. On the other hand, the in-degree of each node is one (except the root), thus $n_i = n_b + n_l + n_{nlc} - 1$, where n_i is the total in-degree of all the nodes. It is obvious that the total in-degree is the same as the total out-degree, i.e., $n_i = n_o$. Hence, $n_b + n_l + n_{nlc} - 1 \geq 2 * n_b + n_{nlc}$, and $n_b \leq n_l - 1$. Thus, $n_{CLCA} \leq n_c + n_b \leq n_c + n_l - 1 \leq 2 * n_c - 1 \leq 2\sum_{i=1}^{m} |\mathscr{I}_i| - 1$.

We can give a query and an XML document, and the number of CLCAs w.r.t. the keyword query and the XML document is exactly $2 * \sum_{i=1}^{m} |\mathscr{I}_i| - 1$. As illustrated in Fig. 3, there are $2 * m - 2$ content nodes and $2 * m - 3$

[1] If the keywords in a keyword query are labels/tags of some nodes in $\mathscr{D}$, these nodes may be non-leaf content nodes in $\mathscr{CG}$.

[2] Some content nodes may appear in different keyword lists. For example, a node, which directly contains both k_i and k_j, must appear in both $\mathscr{I}_i$ and $\mathscr{I}_j$.

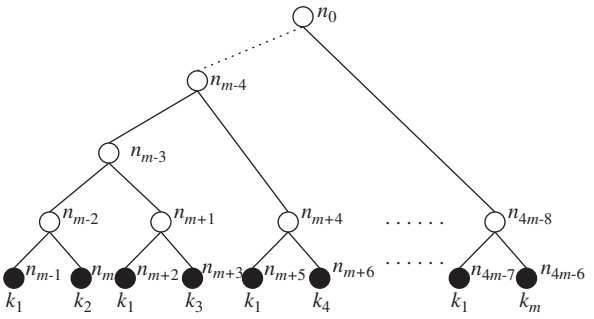

Fig. 3. An example with the minimum upper bound of CLCAs.

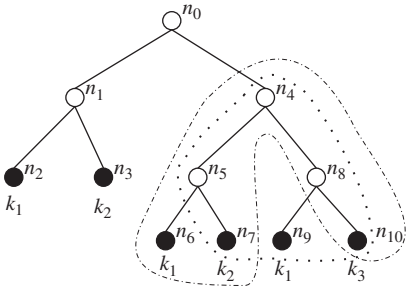

Fig. 4. A CCTree rooted at n_4 by joining two cover trees.

branch nodes. It is easy to figure out that all of these $4 * m - 5$ nodes are CLCAs. □

To summarize, we can merge the content nodes to construct the global tree, and then generate the CGTree by eliminating the linked nodes. More importantly, we present Lemma 2.5 to check whether each node in the CGTree is a CLCA. We can retrieve all CLCAs through one traversal of the CGTree.

Lemma 2.5. *For any node u in the CGTree $\mathscr{CG}$, let $CK(u)$ denote the set of all the distinct keywords contained in the subtree rooted at u. u must be a CLCA, if it satisfies*

(i) *u is a content node*; *or*
(ii) $\exists c_i, c_j \in children(u)$, $CK(c_i) \not\subseteq CK(c_j)$ *and* $CK(c_j) \not\subseteq CK(c_i)$.

Proof. (i) is obvious as u dominates itself. We here mainly prove (ii). If u is not a content node, it must be a branch node and has at least two children. Without loss of generality, suppose $k_i \in CK(c_i)$ and $k_i \notin CK(c_j)$, while $k_j \in CK(c_j)$ and $k_j \notin CK(c_i)$. Let n_i and n_j denote the nodes that contain k_i and k_j in the subtrees rooted at c_i and c_j, respectively. Neither c_i nor c_j contains both k_i and k_j, and u is the LCA of n_i and n_j. Thus u dominates both n_i and n_j w.r.t. $\{k_i, k_j\}$. Hence, u must be a CLCA. □

We propose Lemma 2.6 to guarantee that CLCA always exists, thus we can always find answers using CLCA semantics.

Lemma 2.6. *If node* `lca` *is a LCA, there must exist a corresponding CLCA in the subtree rooted at* `lca`.

Proof. First, if any child of `lca` cannot contain all keywords, `lca` must satisfy condition (ii) of Lemma 2.5 and thus `lca` is a CLCA. Second, after removing the children of `lca` which contain all keywords, if `lca` still contains all keywords, `lca` must be a CLCA as it satisfies condition (ii) of Lemma 2.5. On the contrary, if `lca` does not satisfy the above two conditions, there must exist a descendant of `lca`, u, which contains all keywords and any child of u cannot contain all keywords. Thus u must be a CLCA according to Lemma 2.5. □

Example 2.1. In Fig. 2, suppose answering query $\mathscr{K} = \{k_1, k_3, k_4\}$ in the XML document in (a), we can construct the global tree as illustrated in (b). However, the four linked nodes in (b), n_3, n_{10}, n_{15}, and n_{22}, are less meaningful than other nodes. We construct the corresponding compact global tree by eliminating these four linked nodes as illustrated in (c). It is obvious that all CLCAs w.r.t. $\mathscr{K}$ and the document in (a) are in (c). More importantly, we can check whether each node in the CGTree in (c) is a CLCA as follows. n_0 has three children, n_1, n_{19} and n_{24}. $CK(n_1) = \{k_1, k_3, k_4\}$, $CK(n_{19}) = \{k_1, k_3\}$, $CK(n_{24}) = \{k_4\}$. As $CK(n_{19}) \not\subseteq CK(n_{24})$, $CK(n_{24}) \not\subseteq CK(n_{19})$, n_0 is a CLCA based on Lemma 2.5. n_1 has two children, n_2 and n_{18}, and $CK(n_{18}) = \{k_4\} \subseteq CK(n_2) = \{k_1, k_3, k_4\}$, thus n_1 is not a CLCA. Similarly, we deduce that all nodes except n_1 and n_2 in (c) are CLCAs based on Lemma 2.5.

2.4. Compact connected trees

We present the notion of *compact connected tree* to answer keyword queries in this section. We first introduce the concept of *cover tree*, which is rooted at a CLCA and contains the content nodes dominated by this CLCA. Given q nodes, $v_1 \in \mathscr{I}_1, v_2 \in \mathscr{I}_2, \ldots, v_q \in \mathscr{I}_q$, and suppose $r = CLCA(v_1, v_2, \ldots, v_q)$. The tree rooted at r and containing all the nodes and edges on the path from r to v_i is called a cover tree. Cover trees contain content nodes and their corresponding CLCAs and can describe how the content nodes are connected. Most of existing proposals [13] take cover trees as the answers. However, cover trees are not compact and meaningful enough to answer keyword queries. For example, in Fig. 4, as n_4 dominates n_6, n_7, and n_{10} for $\{k_1,k_2,k_3\}$, n_4 is the CLCA of these three nodes. The subtree composed of n_4, n_5, n_6, n_7, n_8, and n_{10} is a cover tree. The subtree composed of n_4, n_5, n_7, n_8, n_9, and n_{10} is also a cover tree. Note that the two cover trees contain duplicates n_4, n_5, n_7, n_8, and n_{10}. It is evident that the tree rooted at n_4 should be more meaningful to answer $\{k_1,k_2,k_3\}$. Hence, we introduce the concept of *compact connected tree* as follows.

Definition 2.4 (*Compact connected tree*). Givena CLCA `clca` and its cover trees $CT_1, CT_2, \ldots, CT_q$ rooted at `clca`. The tree by joining the cover trees is called the compact connected tree w.r.t. `clca`.

A CCTree is the subtree of the joined result of all cover trees with the same root. As cover trees sharing the same root contain duplicate nodes and the CCTree eliminates such duplicate nodes, CCTree is more meaningful and

compact in answering keyword queries. It is evident that the result set of a single CCTree is more compact than a result set of multiple overlapped cover trees. For example, in Fig. 4, the tree rooted at n_4 is a CCTree.

Given a CLCA, we propose Lemma 2.7 to construct the CCTree rooted at it from the CGTree. More importantly, we can enumerate all CCTrees through one traversal of a given CGTree based on Lemmas 2.5 and 2.7. We take these CCTrees as the answers of keyword proximity queries. For example, in Fig. 4, for n_0, $CK(n_1) \subseteq CK(n_4)$, thus n_0 is not a CLCA. For n_4, $|CK(n_4)| = 3$, $|CK(n_5)| = |CK(n_8)| = 2$, thus n_4 dominates the content nodes that are dominated by n_5 and n_8. Hence, the CCTree rooted at n_4 is exactly the subtree rooted at n_4.

Lemma 2.7. *Given a CGTree $\mathscr{G}$ and a CLCA* `clca` *with q children, $c_1, c_2, \ldots, c_q$, sorted by $|CK(c_i)|$ in descending order. Let T denote the tree rooted at* `clca`*, t_i denote the tree rooted at c_i, and $\forall 1 \le i \le q, T_i = T_{i-1} - t_i (T_0 = T)$. T_{max} is exactly the CCTree rooted at* `clca`*, if $\exists \max (0 \le \max \le q)$, which satisfies*

(i) $\forall j, 0 < j \le \max, |CK(t_j)| = |CK(T_{j-1})|$; *and*
(ii) $|CK(T_{max})| \ne |CK(t_{max+1})| (t_{q+1} = \phi)$,

where $CK(T_j)(CK(t_j))$ denote the sets of keywords contained in $T_j(t_j)$.

Proof. If $|CK(t_j)| = |CK(T_{j-1})|$, `clca` will not dominate any content node in subtree t_j, thus t_j can be eliminated from T_{j-1}. On the contrary, if $|CK(T_{max})| \ne |CK(t_{max+1})|$, it is easy to figure out that `clca` must dominate all the content nodes in T_{max}. Consequently, T_{max} is exactly the CCTree rooted at `clca`. □

We observe that some CCTrees may be subtrees of other CCTrees. Given two CLCAs, u and v, suppose u is a descendant of v, the CCTree rooted at u must be a subtree of the CCTree rooted at v. For example, in Fig. 4, n_5 and n_4 are two CLCAs and n_5 is a descendant of n_4. It is obvious that the CCTree rooted at n_5, CCTree_{n_5}, is a subtree of CCTree_{n_4} (the CCTree rooted at n_4). Moreover, CCTree_{n_4} contains more keywords (e.g., k_3) than CCTree_{n_5}. Hence, CCTree_{n_4} is more meaningful than CCTree_{n_5} to answer query $\{k_1, k_2, k_3\}$. Generally, if there are many CCTrees, it is better to retrieve CCTrees that are not subtrees of other CCTrees as the answers. We will introduce the notion of *maximal CCTree* (MCCTree) to address this issue in Section 2.5.

2.5. Maximal CLCA and maximal CCTree

Definition 2.5 (*Maximal CCTree*). Given a query $\mathscr{K} = \{k_1, k_2, \ldots, k_m\}$ and $\mathscr{K}_i = \{k_{i_1}, k_{i_2}, \ldots, k_{i_q}\} \subseteq \mathscr{K}$. Suppose $w = CLCA(v_{i_1}, v_{i_2}, \ldots, v_{i_q})$, where $v_{i_j} \in \mathscr{I}_{k_{i_j}}$ $(1 \le j \le q)$. w is called a maximal CLCA, if $\forall k' \in (\mathscr{K} - \mathscr{K}_i)$ and $v'_k \in \mathscr{I}_{k'}$, $\nexists w'$, which dominates both v'_k and each v_{i_j} for $1 \le j \le q$. The CCTree rooted at a maximal CLCA (MCLCA) is called a maximal CCTree.

A maximal CLCA is a non-trivial CLCA, which has no ancestors that still dominate some other content nodes besides the content nodes dominated by the MCLCA. Therefore, an MCLCA dominates a maximal set of content nodes and is more meaningful than a CLCA. In addition, a maximal CCTree is the CCTree rooted at an MCLCA and contains more keywords than any CCTree which is one of its subtrees. For example, in Fig. 4, the subtrees rooted at n_1 and n_4 are two MCCTrees w.r.t. the keyword query $\{k_1,k_2,k_3\}$. While the subtrees rooted at n_5 and n_8 are not MCCTrees but only CCTrees. More importantly, all the CLCAs must be descendants of certain MCLCAs, while all the CCTrees must be subtrees of certain MCCTrees, and Lemma 2.8 guarantees the correctness. Therefore, once all the MCLCAs and MCCTrees have been obtained, we can retrieve other CLCAs by checking whether the descendants of those MCLCAs are CLCAs, and other CCTrees by checking whether the roots of the subtrees of those MCCTrees are CLCAs. Note that the number of CLCAs is the same as that of CCTrees while the number of MCLCAs is the same as that of MCCTrees, and they are proportional to the sum of the sizes of keyword lists.

Lemma 2.8. *Given a keyword query $\mathscr{K}$ and an XML document $\mathscr{D}$. Suppose CLCASet is the set of all the CLCAs, MCLCASet is the set of all the MCLCAs, MCCTreeSet is the set of all the MCCTrees, and CCTreeSet is the set of all the CCTrees w.r.t. $\mathscr{K}$ and $\mathscr{D}$. We have the following properties*:

(i) *MCLCASet $\subseteq$ CLCASet and MCCTreeSet $\subseteq$ CCTreeSet.*
(ii) *$\forall u \in$ CLCASet, $\exists v \in$ MCLCASet, $v \preceq u$ (v is an ancestor (or self) of u).*
(iii) *$\forall T_c \in$ CCTreeSet, $\exists T_m \in$ MCCTreeSet, T_c is a subtree of T_m.*

Proof. (i), (ii), and (iii) can be derived according to the definitions of maximal CLCA and maximal CCTree. □

Moreover the number of MCLCAs is no more than the number of all content nodes. Lemma 2.9 gives a more accurate bound of the number of MCLCAs.

Lemma 2.9. *There are at most $\sum_{i=1}^{m} |\mathscr{I}_i|$ MCLCAs (or MCCTrees) w.r.t. a keyword query $\mathscr{K} = (k_1, k_2, \ldots, k_m)$.*

Proof. It is obvious that each content node is at most dominated by one MCLCA, thus the number of MCLCAs is no more than the number of all the content nodes, i.e., $\sum_{i=1}^{m} |\mathscr{I}_i|$. □

To generate the MCCTrees, we need to check whether each node in the CGTree is an MCLCA, and if the node is an MCLCA, we retrieve the MCCTree rooted at it according to Lemma 2.7. Moreover, we introduce Lemma 2.10 to check whether a CLCA is an MCLCA. For any CLCA, `clca`, if it is the root of the CGTree, it must be an MCLCA; otherwise, we check if there is an ancestor of `clca`, which still dominates some other content nodes besides the content nodes in the CCTree rooted at `clca`. If so, `clca` cannot be an MCLCA. On the contrary, `clca` must be an MCLCA.

Lemma 2.10. *A CLCA u is an MCLCA if it satisfies* (1) *u is the root*; *or* (2) *the following conditions hold*:

(i) *$\forall$* `a` $\prec u$, $CK($`a`$) = \bigcup_{c_i \in children(\mathtt{a})} CK(c_i)$; *and*
(ii) *$\forall$* `a` $\preceq u$, *$\nexists$* `s`, *a sibling of* `a`, $CK(\mathtt{s}) \nsubseteq CK(\mathtt{a})$ *and* $CK(\mathtt{a}) \nsubseteq CK(\mathtt{s})$.

Proof. (1) is obvious. We here mainly prove (2). (i) guarantees that, for any ancestor of u, a, even if a is a content node, the keywords associated with a must be contained by one of its children; while (ii) guarantees that, for any ancestor (or self) of u, its parent cannot dominate both the content nodes dominated by itself and the content nodes dominated by its siblings, thus u dominates a maximal set of content nodes. Hence u must be an MCLCA. □

Example 2.2. In Fig. 1, as $CK(n_{19}) = \{k_1, k_3\} \nsubseteq CK(n_{22}) = \{k_2, k_4\}$ and $CK(n_{22}) \nsubseteq CK(n_{19})$, n_0 is a CLCA. As n_0 is the root, n_0 is an MCLCA. For n_6, it is a CLCA and its ancestors n_0, n_1 and n_2 cannot dominate the content nodes that are dominated by n_6, thus n_6 is an MCLCA. For n_7, $CK(n_7) \nsubseteq CK(n_{10})$ and $CK(n_{10}) \nsubseteq CK(n_7)$, thus n_7 is not an MCLCA as its parent n_6 contains more keywords than n_7 and n_{10}. As $|CK(T_{n_0})| = |CK(T_{n_1})|$, the subtree rooted at n_1 does not belong to the MCCTree rooted at n_0 (denoted as MCCTree_{n_0}). As $|CK(T_{n_0} - T_{n_1})| \neq |CK(T_{n_{19}})|$, MCCTree_{n_0} is exactly $T_{n_0} - T_{n_1}$. Accordingly, we can generate all the MCCTrees, i.e., the five circled subtrees as illustrated in Fig. 1.

In summary, when computing the *top-k* answers, we first generate the MCCTrees as follows. We check whether each node in a given CGTree is a CLCA or MCLCA according to Lemmas 2.5 and 2.10. If the node is an MCLCA, we retrieve the MCCTree rooted at it as formalized in Lemma 2.7. If the number of MCCTrees is smaller than k, we retrieve other CCTrees that are subtrees of certain MCCTrees as stated in Lemma 2.8. Moreover, the CCTrees can be generated through one traversal of MCCTrees. Finally, we rank MCCTrees (or CCTrees) to return the *top-k* answers with the highest ranks.

3. Algorithms

We propose two algorithms `CGTreeGenerator` and `MCCTreesGenerator` to construct the CGTree and generate the corresponding MCCTrees, respectively.

3.1. CGTreeGenerator

To construct the CGTree, there is a straightforward way: we first compute the LCA of all content nodes in each $\mathscr{I}_i$, denoted as `lca`, and then for each node $n_{i_j} \in \mathscr{I}_i$, enumerate each path tree T_{i_j}, which is rooted at `lca` and contains the nodes and edges from `lca` to n_{i_j}. Subsequently, we join each T_{i_j} to generate the global tree. Finally, we construct the CGTree by eliminating the linked nodes from the global tree. However, this straightforward method is inefficient, because it involves two steps: one is to generate the global tree and the other is a postprocessing to eliminate the linked nodes.

There is an alternative way to construct the CGTree iteratively: for any structural node, u, if u has only one subtree that only contains content nodes and branch nodes, u must be a linked node. This subtree is taken as a subtree of the parent of u; otherwise, u must be a branch node. A super-tree rooted at u and containing all of its subtrees is constructed. This super-tree is taken as a subtree of the parent of u. Iteratively, we can construct the CGTree. We propose an efficient algorithm to construct the CGTree without postprocessing.

To efficiently construct the CGTree, we maintain a set of nodes in a stack $\mathscr{S}$. Each node in $\mathscr{S}$ is the child of the node directly below it. Each node in the stack keeps all of its subtrees rooted at its descendants and having been already constructed. Moreover, when merging a content node into a subtree, we visit content nodes in document order. That is, we always select the minimal node n_{min}, which precedes all other nodes. While the topmost node of $\mathscr{S}$ is not an ancestor of n_{min}, we pop it from $\mathscr{S}$; otherwise, we push n_{min} and its ancestors into the stack. When popping a node n from $\mathscr{S}$, we check if n contains only one subtree and is not a content node. If so, n must be a linked node and we transfer the subtree associated with n to the parent of n (i.e., the node below n); otherwise, we construct the super-tree that is rooted at n and contains all subtrees associated with n, and transfer this super-tree to the parent of n. We repeat these steps until all content nodes are merged into a subtree. The final super-tree is exactly the CGTree. Note that our algorithm is different from PathStack algorithm [5]. PathStack identifies all subtrees for an XPath query while our algorithm finds a CGTree that contains all input keywords.

Dewey numbers [29,36] provide a straightforward solution to locate the LCAs, and prior work [37] has shown that employing Dewey numbers is a good choice for keyword search in XML documents. We also employ Dewey numbers to encode XML elements for improving the efficiency of constructing the CGTree. We present an algorithm `CGTreeGenerator` (Fig. 5). `CGTreeGenerator` first retrieves the keyword lists (line 3), and then while either $\mathscr{S}$ or $\mathscr{I}_i$ is not empty, selects the minimal node n_{min} that has the minimal Dewey number among those of the first nodes in each current $\mathscr{I}_i$ (lines 5–6), pops it from $\mathscr{I}_{min}$ (line 7), and subsequently merges it into a subtree. While the top node of $\mathscr{S}$ (denoted as n) is not an ancestor of n_{min}, `CGTreeGenerator` pops n from $\mathscr{S}$ (line 9) and merges the subtrees associated with n to construct a super-tree (lines 10–15). Especially, if n is not a content node and has only one subtree, it must be a linked node, thus the subtree associated with it is exactly a subtree of its parent (lines 10–11); otherwise, `CGTreeGenerator` constructs a super-tree rooted at n and containing all of its subtrees (lines 12–15). Subsequently, `CGTreeGenerator` transfers the current subtree to the top node of $\mathscr{S}$ (line 16). On the other hand, if n_{min} is a descendant of n, `CGTreeGenerator` pushes n_{min} and its ancestors into $\mathscr{S}$, and takes n_{min} itself as the subtree associated with the top node of $\mathscr{S}$ (lines 17–18). Finally, `CGTreeGenerator` returns the compact global tree, `CGTree` (line 19). Theorem 1 guarantees the correctness of this algorithm and gives the complexity.

Theorem 1. `CGTreeGenerator` *constructs the CGTree correctly. Its complexity is* $O(d * \log m * \sum_{i=1}^{m} |\mathscr{I}_i|)$, *where* d *is the depth of the XML document and* m *is the number of keywords.*

Algorithm 1: CGTreeGenerator Algorithm

Input: A keyword query $\mathcal{K} = \{k_1, k_2, \ldots, k_m\}$ and an XML document $\mathcal{D}$
Output: CGTree, the compact global tree for $\mathcal{K}$ and $\mathcal{D}$.

```
1  begin
2      CGTree←φ;
3      getKeywordLists(); //I_i={v_i|v_i directly contains keyword k_i}
4      while S is not empty or I_i is not empty do
5          min=minarg_i(I_i.first());
6          n_min=I_min.first();
7          I_min.pop-front();
8          while n=S.top() is not an ancestor of n_min do
9              S.pop();
10             if n is not a content node and has only one subtree t_c then
11                 CGTree=t_c;
12             else
13                 CGTree={n};
14                 foreach t_c∈n.Subtrees do
15                     CGTree=CGTree→t_c;
16             S.top().Subtrees∪={CGTree};
17         S.push(n_min);   //push n_min and its ancestors into S
18         S.top().Subtrees={n_min};
19     return CGTree;
20 end
```

Fig. 5. CGTreeGenerator algorithm.

Proof. It is easy to figure out that CGTreeGenerator can construct the CGTree correctly, and we here mainly prove the complexity. Firstly, the algorithm needs to select the minimal node, and the complexity is $O(\log m)$ through constructing a selection tree. Secondly, for each node in the keyword lists, the algorithm needs to push it into the stack and pop the nodeswhich are not its ancestors from the stack, and the complexity is $O(d)$. Subsequently, CGTreeGenerator merges all the content nodes to construct the CGTree. Hence the total complexity is $O(d * \log m * \sum_{i=1}^{m} |\mathscr{I}_i|)$. □

To further illustrate how the algorithm works, we walk through our algorithm with a running example as shown in Example 3.1.

Example 3.1. In Fig. 6, suppose constructing the CGTree on query $\{k_1,k_3,k_4\}$, the keyword lists are $\mathscr{I}_{k_1} = \{n_4,n_8,n_{16},n_{20}\}$, $\mathscr{I}_{k_3} = \{n_9,n_{21}\}$, and $\mathscr{I}_{k_4} = \{n_{12},n_{14},n_{18},n_{24}\}$. Initially, as $n_{min} = n_4$, n_4 (associated with the subtree rooted at n_4) is pushed into the stack (Fig. 7(1)). Then, $n_{min} = n_8$. As the top node of $\mathscr{S}$ n_4 is not an ancestor of n_{min}, n_4 is popped from the stack. The subtree associated with n_4 is transferred to n_4's parent, n_3(0.0.0) (Fig. 7(2)), and then to n_2(0.0) (Fig. 7(3)). Subsequently, n_8 is pushed into the stack with a subtree rooted at n_8(0.0.1.0.0) (Fig. 7(4)). Similarly, we walk through our algorithm as illustrated in Fig. 7, and finally construct the CGTree as shown in Fig. 6(b). Especially,$n_4|n_6[n_7[n_8,n_9],n_{12}]$ in Fig. 7(9) denotes that there have been already two subtrees constructed for node n_2(0.0): one is the subtree rooted at n_4 and the other is the subtree rooted at n_6 of the CGTree in Fig. 6(b).

3.2. MCCTreesGenerator

We propose an effective algorithm MCCTreesGeneratorto generate the MCCTrees in this section. There is a straightforward way to generate all MCCTrees: we check whether each node in the CGTree is an MCLCA or not according to Lemma 2.10. If so, we retrieve the MCCTree rooted at it as stated in Lemma 2.7. However, this approach is inefficient since it needs to check whether each ancestor of a given node dominates more content nodes than this given node. This check involves redundant computations and leads to low efficiency. Therefore, we introduce an optimization technique as described in Lemma 3.1 to improve the efficiency of retrieving the MCCTrees.

Lemma 3.1. *Suppose T_r is any subtree rooted at r of a CGTree* CG, *and r has q children $c_1, c_2, \ldots, c_q$ sorted by $|CK(c_i)|$ in descending order. Let t_i denote the tree rooted at c_i, and $\forall 1 \le i \le q$, $T_i = T_{i-1} - t_i (T_0 = T_r)$. f(CG) is exactly the set of all MCCTrees for* CG, *if function f satisfies*

(i) $f(T_r) = \{T_r\}$, *if r has no child (i.e., r is a leaf content node).*
(ii) $f(T_r) = \bigcup_{i=1}^{max} f(t_i) \bigcup \{T_{max}\}$, *if $\exists max (0 \le max \le q)$, which satisfies*

  (1) $\forall j, 0 < j \le max$, $|CK(T_{j-1})| = |CK(t_j)|$; *and*
  (2) $|CK(T_{max})| \ne |CK(t_{max+1})| (t_{q+1} = \phi)$.

(iii) $f(T_r) = \bigcup_{i=1}^{q} f(t_i)$, *otherwise.*

Proof. This lemma is easy to be proved according to Lemmas 2.7 and 2.10, and the detailed proof is omitted due to limited space. □

According to Lemma 3.1, we can efficiently generate all MCCTrees from the root iteratively without involving redundant computations to check whether each node in the CGTree is an MCLCA as stated in Lemmas 2.5 and 2.10. Moreover, for any node n, $CK(n)$ can be computed during constructing the CGTree. We devise an efficient algorithm

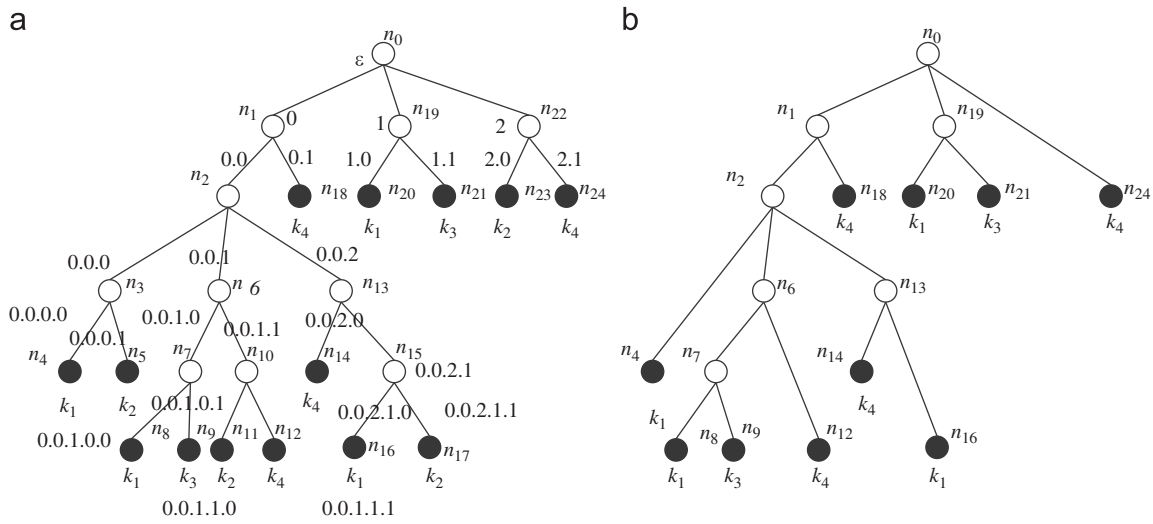

Fig. 6. An XML document with Dewey numbers and its for $\{k_1,k_3,k_4\}$.

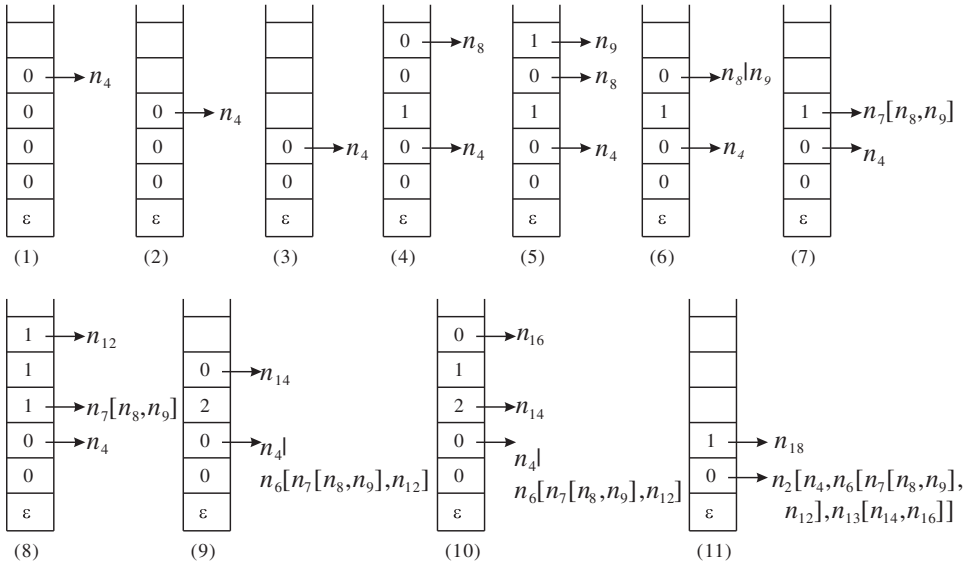

Fig. 7. An example to construct CGTree for $\{k_1,k_3,k_4\}$ and the XML document in Fig. 6(a).

(Fig. 8) to generate the MCCTrees. `MCCTreesGeneretor`-first initializes the candidate set of MCCTrees (denoted as *CMSet*) as {*CGTree*} (line 3). While *CMSet* is not empty, `MCCTreesGeneretor`selects any candidate MCCTree (denoted as *T*) from *CMSet*, and then eliminates it from *CMSet* (lines 5–6). For each child of the root of *T*, c_i, if the tree t_i rooted at c_i contains the same keywords as *T* (line 9), t_i will not belong to the MCCTree rooted at the root of *T*, and `MCCTreesGeneretor`updates *T* by eliminating t_i (line 10). Furthermore, t_i must contain candidate MCCTrees, therefore `MCCTreesGeneretor`adds t_i into *CMSet* (line 11). After eliminating the subtrees that contain candidate MCCTrees, if *T* still contains keywords, *T* must be an MCCTree and is added into `MCCTreeSet`(lines 14–15). `MCCTreesGeneretor`repeats these steps until *CMSet* is empty. Theorem 2 guarantees the correctness of this algorithm.

Theorem 2. `MCCTreesGeneretor`*correctly generates all the MCCTrees for a CGTree. Its complexity is* $O(m * \sum_{i=1}^{m} |\mathscr{I}_i|)$, *where m is the number of keywords.*

Proof. It is easy to figure out this algorithm is consistent with Lemma 3.1. We here mainly prove the complexity. For any node *n*, since $|CK(n)| \leq m$, the complexity of sorting its children is $O(m * CN)$ through the bucket sort, where *CN* is the number of the children of *n*. The total complexity to generate the MCCTrees is $O(m * N_I)$, where N_I is the total number of the nodes in the CGTree and

Algorithm 2: MCCTreesGeneretor Algorithm

Input: A compact global tree, $CGTree$
Output: MCCTreeSet, a set composed of all the MCCTrees.

```
1  begin
2      MCCTreeSet←φ;
3      CMSet={CGTree};
4      while CMSet is not empty do
5          r=root(T∈CMSet); //select any subtree, T, from CMSet
6          CMSet=CMSet-{T};
7          foreach c_i∈children(r)(in descending order by |CK(c_i)|) do
8              t_i=subtree(c_i);
9              if |CK(T)|==|CK(t_i)| then
10                 T=T-t_i;
11                 CMSet=CMSet∪{t_i};
12             else
13                 break;
14         if |CK(T)|>0 then
15             MCCTreeSet=MCCTreeSet∪{T};
16     return MCCTreeSet;
17 end
```

Fig. 8. MCCTreesGeneretor algorithm.

$N_I \leq 2\sum_{i=1}^{m}|\mathscr{I}_i|$. Hence, the complexity of MCCTreesGeneretor is $O(m * \sum_{i=1}^{m}|\mathscr{I}_i|)$. □

To further illustrate how the algorithm works, we walk through our algorithm with a running example as shown in Example 3.2.

Example 3.2. In Fig. 1, $CK(T_{n_0})=\{k_1,k_2,k_3,k_4\}$. As $CK(T_{n_1})=\{k_1,k_2,k_3,k_4\}$, $CK(T_{n_{19}})=\{k_1,k_3\}$, and $CK(T_{n_{22}})=\{k_2,k_4\}$, the children of n_0, sorted by the number of contained keywords, are n_1, n_{19} and n_{22}. Subsequently, as $|CK(T_{n_1})|=|CK(T_{n_0})|$, $CK(T_{n_1})$ must contain some candidate MCCTrees, thus T_{n_1} is added into *CMSet*. In addition, $|CK(T_{n_0}-T_{n_1})|\neq|CK(T_{n_{19}})|$, thus, $T_{n_0}-T_{n_1}$ is an MCCTree. As T_{n_1} is in *CMSet* and $|CK(T_{n_1})|=|CK(T_{n_2})|$, T_{n_2} contains some candidate MCCTrees and is added into *CMSet*. Furthermore, $|CK(T_{n_{18}})|=|CK(T_{n_1}-T_{n_2})|$, thus $T_{n_{18}}$ is added into *CMSet*. Consequently, $T_{n_{18}}$ is an MCCTree while T_{n_2} is not. Subsequently, T_{n_6}, $T_{n_{13}}$ and T_{n_3} are orderly added into *CMSet* and they are all MCCTrees. To sum up, there are five MCCTrees in the CGTree as shown in the dashed circles.

4. Ranking

As there are many MCCTrees with different significance (importance), we propose a ranking mechanism to rank MCCTrees in order to return the *top-k* relevant answers with the highest scores. Without loss of generality, we only show how to rank MCCTrees and the same method can be applied to the ranking of CCTrees.

4.1. MCCTree ranking model

The $tf \cdot idf$ based methods for ranking relevant documents have been proven to be effective for keyword search in textual documents. However, traditional ranking techniques in IR literature may not be effective to rank those MCCTrees. As besides the term frequency (tf) and inverse document frequency (idf), these MCCTrees also contain rather rich structural information. For example, in Fig. 1, T_2 is much more relevant to the query than T_1, as T_2 is more compact. We incorporate the structural compactness of an MCCTree into the ranking function and propose Formula (4.1) to rank an MCCTree.

$$Sim(\mathscr{K},\mathscr{T}) = \delta * Sim_{str}(\mathscr{K},\mathscr{T}) + (1-\delta) * Sim_{text}(\mathscr{K},\mathscr{T}), \tag{4.1}$$

where $Sim_{str}(\mathscr{K},\mathscr{T})$ denotes the structural compactness of $\mathscr{T}$ and $Sim_{text}(\mathscr{K},\mathscr{T})$ denotes the text similarity between $\mathscr{K}$ and $\mathscr{T}$. Note that structural compactness reflects the compactness of an MCCTree from the DB point of view, while text similarity captures the text relevancy, similar to the document relevancy in IR literature. δ is a real number parameter to differentiate the importance of two scores. Especially, if δ is set to zero, it means that we only consider the text similarity like traditional ranking methods. While if δ is set to one, it means that we only consider the structural compactness. We will experimentally demonstrate how to set δ in Section 5. We will introduce how to compute $Sim_{str}(\mathscr{K},\mathscr{T})$ and $Sim_{text}(\mathscr{K},\mathscr{T})$ in Sections 4.2 and 4.3, respectively.

4.2. Structural compactness

We in this section introduce how to compute the structural compactness. Intuitively, when an MCCTree is more compact, it is more likely to be meaningful and relevant. Thus, the structural compactness score should be larger. As such, the compactness of an MCCTree should include the structural relevancy between content nodes and the relevancy between input keywords. We note that when the distance between two content nodes is larger, the relevancy between them is smaller. Further, there may be multiple contents between two keywords, and we should consider all of them. Based on the above rationale, we propose Eq. (4.2) to score the relevancy between any two keywords.

$$Rel(k_i,k_j) = \sum_{n_i\in C_{k_i};n_j\in C_{k_j}} \frac{1}{dist(n_i,n_j)+1}, \tag{4.2}$$

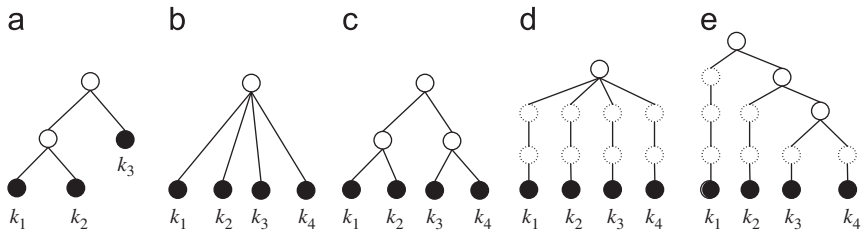

Fig. 9. Five MCCTrees of a keyword query $\mathscr{K} = \{k_1, k_2, k_3, k_4\}$. (a) $\mathscr{T}_a$; (b) $\mathscr{T}_b$; (c) $\mathscr{T}_c$; (d) $\mathscr{T}_d : \mathscr{T}_e$.

where C_{k_i} denotes the set of all content nodes that contain k_i in the MCCTree and $dist(n_i, n_j)$ denotes the distance between n_i and n_j. Moreover, it is obvious that the more succinct of an MCCTree, the higher score of the structural compactness. We employ the distance between each content node and the root of the MCCTree to capture the structural compactness. It is evident that the larger this distance, the less succinct the MCCTree, and thus the smaller the structural compactness score. The succinctness of the MCCTree should be reflected in the function. We present Formula (4.3) to compute the compactness of an MCCTree,

$$Compact(\mathscr{T}) = \sum_{c \in \mathscr{T}_c} \frac{1}{l_c - l_r + 1}, \tag{4.3}$$

where $\mathscr{T}_c$ denotes the content node set in $\mathscr{T}$, r denotes the root of $\mathscr{T}$, and $l_c(l_r)$ denotes the level of $c(r)$ in the XML document. We present Formula (4.4) to compute the overall compactness of an MCCTree:

$$Sim_{str}(\mathscr{K}, \mathscr{T}) = \sum_{1 \le i < j \le m} Rel(k_i, k_j) * Compact(\mathscr{T}). \tag{4.4}$$

Example 4.1. Given a keyword query $\mathscr{K} = \{k_1, k_2, k_3, k_4\}$, suppose that there are five MCCTrees as illustrated in Fig. 9, where the dashed nodes in the MCCTrees denote that they are linked nodes. For $\mathscr{T}_a$, we have $Rel(k_1, k_2) = \frac{1}{3}$, $Compact(\mathscr{T}_1) = \frac{1}{3} + \frac{1}{3} + \frac{1}{2}$, and $\mathscr{W}_{str}(\mathscr{K}, \mathscr{T}) = \frac{35}{36}$. We sort these five MCCTrees by their structural compactness in descending order, $\mathscr{T}_b$, $\mathscr{T}_c$, $\mathscr{T}_a$, $\mathscr{T}_d$ and $\mathscr{T}_e$.

4.3. Text similarity

In this section, we present how to compute textual similarity. We present Eq. (4.5) to compute the text similarity,

$$Sim_{text}(\mathscr{K}, \mathscr{T}) = \sum_{k_i \in \mathscr{K}} \mathscr{W}_{text}(k_i, \mathscr{T}), \tag{4.5}$$

where $\mathscr{W}_{text}(k_i, \mathscr{T})$ denotes the importance of k_i in $\mathscr{T}$. There may be different content nodes that contain k_i in $\mathscr{T}$. Therefore, we first compute the text similarity between k_i and each content node c that contains k_i in $\mathscr{T}$, denoted as $\mathscr{W}(k_i,c)$, and then take the sum of each $\mathscr{W}(k_i,c)$ as the text similarity of k_i in $\mathscr{T}$. We propose Formula (4.6) to compute $\mathscr{W}_{text}(k_i, \mathscr{T})$,

$$\mathscr{W}_{text}(k_i, \mathscr{T}) = \frac{1}{Size(\mathscr{T})} \sum_{c \in \mathscr{T}_c \cap \mathscr{I}_{k_i}} \mathscr{W}(k_i, c), \tag{4.6}$$

where $\mathscr{T}_c$ denotes the content node set in $\mathscr{T}$ and $\mathscr{I}_{k_i}$ denotes the input keyword list w.r.t. k_i. As an MCCTree contains more content nodes does not imply it is more meaningful, we employ the size of $\mathscr{T}$, $Size(\mathscr{T})$, to normalize the text similarity, which is the number of nodes in $\mathscr{T}$.

Furthermore, since k_i can be taken as either the tag/label of a content node, c, or a term in the content of c, and both of them will affect the text similarity between k_i and c, i.e., $\mathscr{W}(k_i, c)$. To accurately capture this similarity, we should take both of these two cases into consideration. Accordingly we propose Formula (4.7) to compute $\mathscr{W}(k_i, c)$:

$$\mathscr{W}(k_i, c) = \mathscr{W}_{label}(k_i, c) + \mathscr{W}_{text}(k_i, c). \tag{4.7}$$

On the one hand, if k_i is considered as the tag of c, its term frequency should be one. Therefore, we only consider its inverse document frequency and take this frequency as the text similarity between k_i and c, i.e., $\mathscr{W}_{label}(k_i, c)$, as described in Formula (4.8),

$$\mathscr{W}_{label}(k_i, c) = \ln \frac{N+1}{N_c + 1}, \tag{4.8}$$

where N is the number of the elements in the XML document and N_c is the number of such elements with tag c.

On the other hand, if k_i is taken as a term in the content of c, we should consider both its normalized term frequency (ntf) as shown in Formula (4.10) and inverse document frequency (idf) as described in Formula (4.11). In addition, the longer the content of a node, the more terms in it, thus we propose the normalized text length (ntl) to normalize the text length (tl) of a node as described in Formula (4.12). Hence, we compute $\mathscr{W}_{text}(k_i, c)$ by taking into consideration the three parameters as shown in Formula (4.19):

$$\mathscr{W}_{text}(k_i, c) = \frac{ntf * idf}{ntl}, \tag{4.9}$$

$$ntf = 1 + \ln(tf), \tag{4.10}$$

$$idf = \ln \frac{N_c + 1}{N_{(k_i,c)} + 1}, \tag{4.11}$$

$$ntl = (1 - s) + s * \frac{tl_c}{avg_{tl_c}}. \tag{4.12}$$

Formula (4.10) is used to compute ntf, where tf is the term frequency of k_i in c. Formula (4.11) is adopted to compute idf, where $N_{(k_i,c)}$ is the number of nodes, which

Table 1
Forty keyword queries on DBLP dataset used in the experiments.

Q_3^1	XML database system	Q_4^1	jim gray title year
Q_3^2	XML SIGMOD search	Q_4^2	author database index 2003
Q_3^3	keyword search author	Q_4^3	phdthesis title author school
Q_3^4	proximity keyword author	Q_4^4	schema data integration url
Q_3^5	jim gray david	Q_4^5	booktitle jim gray year
Q_3^6	gray book year	Q_4^6	XML ICDE author year
Q_3^7	VLDB keyword XML	Q_4^7	data mining author 2006
Q_3^8	keyword search journal	Q_4^8	twig join 2005 author
Q_3^9	XML keyword search	Q_4^9	IR XML author year
Q_3^{10}	proximity keyword search	Q_4^{10}	data stream 2006 author
Q_5^1	jim gray data year author	Q_6^1	keyword search XML author 2006 url
Q_5^2	data stream VLDB author year	Q_6^2	jim gray database ACM url year
Q_5^3	keyword search XML journal author	Q_6^3	XML cache view 2005 authorpages
Q_5^4	article data stream 2006 author	Q_6^4	jim gray Ullman ACM title url
Q_5^5	XML database privacy author pages	Q_6^5	data mining jiawei 2005 title url
Q_5^6	XML twig join 2005 url	Q_6^6	graph mining VLDB pdf year author
Q_5^7	keyword search database 2004 author	Q_6^7	data stream 2005 pdf author url
Q_5^8	nested XML year author SIGMOD	Q_6^8	jagadish XML ICDE conf year mdate
Q_5^9	keyword search XML 2006 author	Q_6^9	Alfred Ullman title STOC year pages
Q_5^{10}	IR VLDB conf pages url	Q_6^{10}	twig XML query 2005 author url

also contain k_i and have the same tag with c. We use Formula (4.12) to compute ntl, where tl_c is the number of terms in c and avg_{tl_c} is the average number of terms among all such nodes with tag c. s is a constant parameter borrowed from IR literature [27], which is usually set to 0.2.

5. Experimental study

We have designed and performed a comprehensive set of experiments to evaluate the performance of our approach. We used real datasets DBLP[3] and TreeBank[4] in our experiments. The DBLP dataset is highly structured. The Treebank dataset is also structured with deep recursion. It has a very complicated structure (without explicit schema). It is composed of English sentences, tagged with parts of speech. The text nodes have been encrypted because they are copywritten text from the Wall Street Journal. The deep recursive structure of this data makes it an interesting case for experiments. The raw files of DBLP and TreeBank were, respectively 470 and 82 MB. We compared with the state-of-the-art approaches, XSEarch [6], XRank [9], and MSLCA [35].

To better understand the performance of our method on different keyword queries with various characteristics, we selected 40 keyword queries on DBLP with the number of keywords varying from 3 to 6 as illustrated in Table 1 and 40 keyword queries on TreeBank. All the experiments were conducted on an Intel(R) Pentium(R) 2.4 GHz computer with 2 GB of RAM running Windows XP Professional. The algorithms were implemented in Java.

[3] http://dblp.uni-trier.de/xml/.
[4] http://www.cs.washington.edu/research/xmldatasets/.

5.1. Efficiency

This section evaluates search efficiency. We first compared `MCCTreesGenerator` with MSLCA [35], XSEarch [6], and XRank [9] in terms of conjunctive semantics, and then compared `CCTreesGenerator` with them in terms of disjunctive semantics. `CCTreesGenerator` is evolved from `MCCTreesGenerator` and used to generate CCTrees. Figs. 10 and 11 illustrate experimental results on the two datasets, respectively. In the figures, `MCCTree` and `CCTree`, respectively denote `MCCTreesGenerator` and `CCTreesGenerator`.

Fig. 10(a) describes the obtained results in considering the conjunctive semantics. We observe that `MCCTree` is better than XSEarch and XRank, and is comparable with MSLCA. This reflects that `MCCTree` is at least as good as MSLCA in terms of conjunctive semantics. More importantly, `CCTree` significantly outperforms the four existing methods when considering the disjunctive semantics as illustrated in Fig. 10(b). Furthermore, XRank only considers the conjunctive semantics, and we enumerated all the combinations of keywords to make XRank support the disjunctive semantics. Thus, XRank is not efficient to answer keyword proximity queries in terms of disjunctive semantics. However, our method is naturally efficient for both conjunctive semantics and disjunctive semantics. Especially, for Q_6^2 and Q_6^4, XSEarch, XRank and MSLCA take more than 10^7, 10^7, 10^6 ms, respectively, while CCTree only takes 10^4 ms. Moreover, we observe that the more keywords involved in a query, the more improvement of our method over the existing approaches. The main reason is that, XSEarch has to maintain an all-pairs interconnection index to check the connectivity between any two nodes, which is rather inefficient for large XML documents.

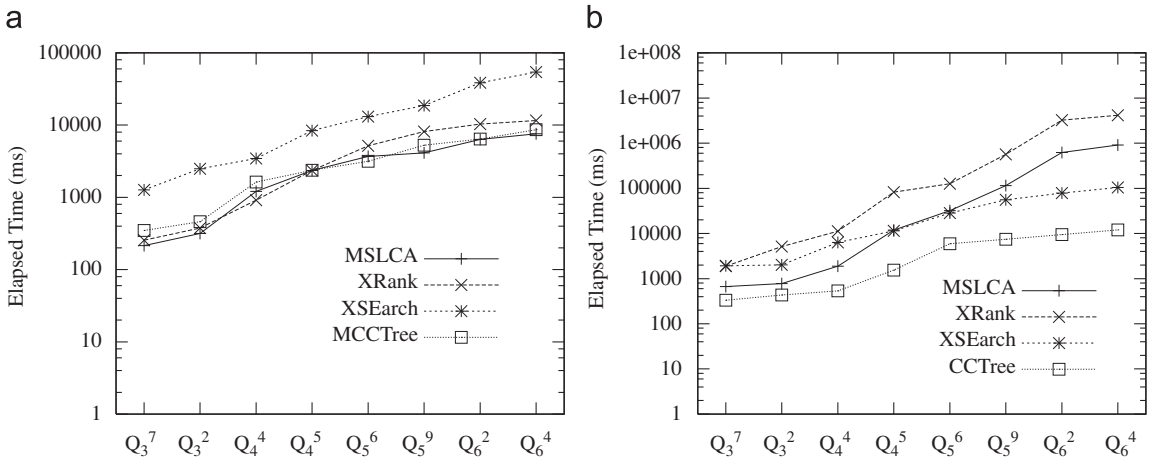

Fig. 10. Efficiency of various algorithms on DBLP. (a) Conjunctive Semantics; (b) Disjunctive Semantics.

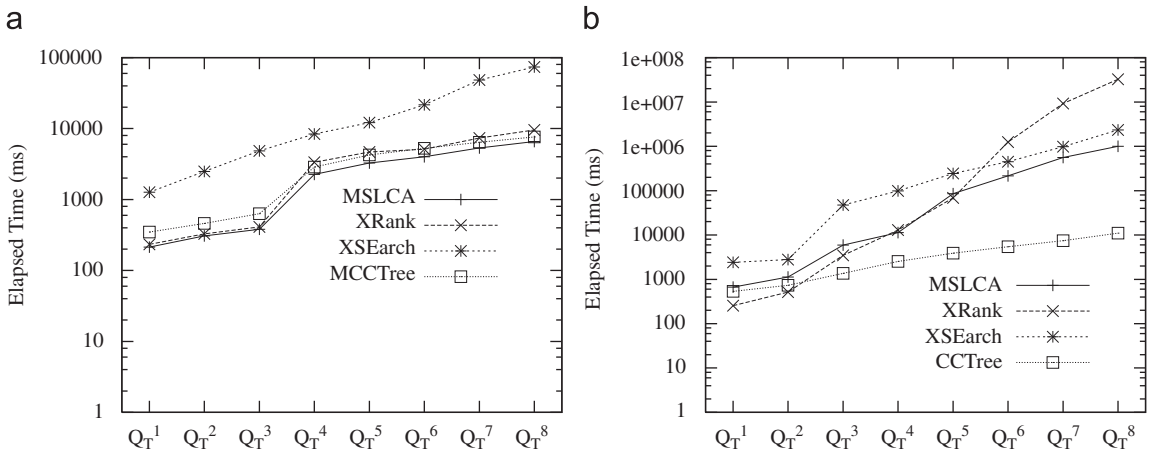

Fig. 11. Efficiency of various algorithms on TreeBank. (a) Conjunctive Semantics; (b) Disjunctive Semantics.

5.2. Effectiveness

This section evaluates the search effectiveness. We first evaluated the effect of δ, which differentiates the importance of the structural compactness and the text similarity, and then demonstrated how to select a best δ for our ranking mechanism. Finally, we evaluated the search quality of our method.

To evaluate the effectiveness of an algorithm, we employed the well-known metrics precision and recall. In order to compute the correct answers of a keyword query, we transformed the query into its corresponding schema-aware XQuery statement based on the structure of DBLP and took the answer of the transformed query as a baseline to compute precision and recall. For example, consider $Q_4^6 = \{$`XML ICDE author year`$\}$, we translated it to a XQuery,

```
For $p in $doc//paper
For $t in $p/title
For $b in $p/booktitlewherecontains
($t/text(), "XML") and contains($b/text(), "ICDE") return
⟨paper⟩ {$t, $b, $p/author, $p/year } ⟨/paper⟩
```

Without loss of generality, suppose *AR* is the correct result of a specified query $\mathcal{Q}$, i.e., the answer of the transformed XQuery, and *PR* is the approximate result of a given algorithm. We defined the precision of the given algorithm in terms of conjunctive semantics as $|AR \cap PR|/|PR|$ and recall is $|AR \cap PR|/|AR|$. Similarly, we can define the precision and recall in terms of disjunctive semantics, and we will compare the precision by considering both conjunctive semantics and disjunctive semantics in Section 5.2.2. Furthermore, to evaluate the search quality of an algorithm, we employed another good metric, *top-k* precision, which measures the ratio of the number of accurate answers among the first *k* returned results with highest scores of an algorithm to *k*.

5.2.1. Effect of δ

This section evaluates the effect of δ and introduces how to select a best δ for our ranking mechanism. We still

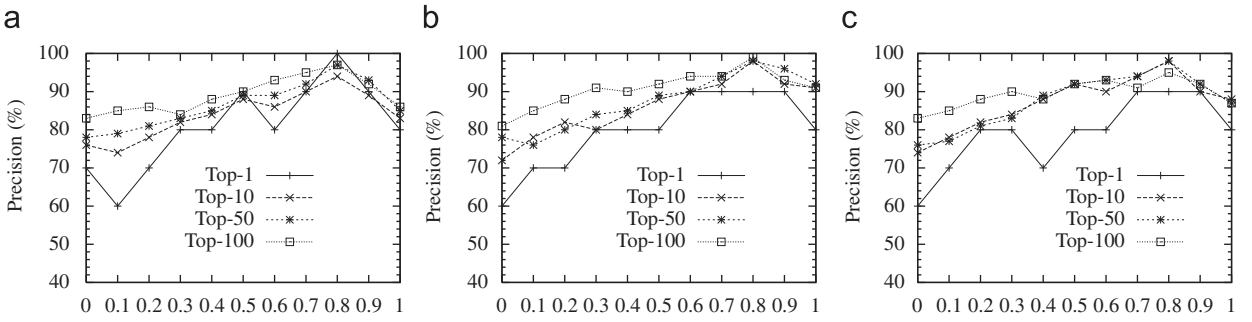

Fig. 12. *top-k* precision vs. different values of δ on DBLP. (a) $\delta(|Q| = 4)$; (b) $\delta(|Q| = 5)$; (c) $\delta(|Q| = 6)$.

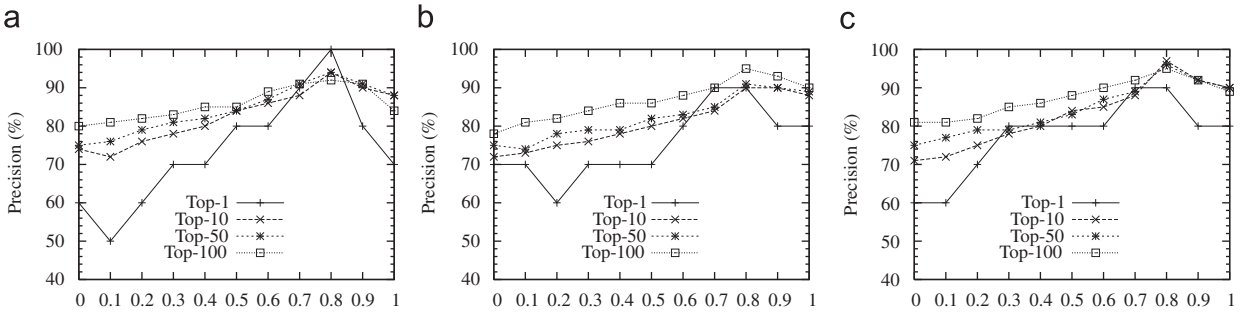

Fig. 13. *top-k* precision vs. different values of δ on TreeBank. (a) $\delta(|Q| = 4)$; (b) $\delta(|Q| = 5)$; (c) $\delta(|Q| = 6)$.

employed the 40 keyword queries with various keyword numbers in Table 1. We varied different values of δ and computed the corresponding *top-k* precision. The obtained results are illustrated in Figs. 12 and 13, where (a), (b) and (c) describe the average *top-k* precision of the 10 keyword queries with different keyword numbers of 4, 5, and 6, respectively.

We observe that our method achieves much better precision when δ in the range of 0.7 and 0.9. Our method has a perfect precision when δ is set to 0.8. As expected, our method always achieves high precision for various queries with different characteristics. Especially, the *top-k* precision of our method exceeds 90% when δ falls in the range of 0.7 and 0.9, and even reaches 100% for many queries. This further demonstrates that the structure compactness and the text similarity are both important to rank those tree-structured answers. The tree compactness will rank the answers with more succinct tree-structure higher, while the text similarity will rank the answers with more meaningful keywords higher.

Note that in this paper, we focus on data-centric XML documents, thus δ is set to a high value. For the document-centric XML data, XML content (text values) should be more important and δ should be set to a small value. In future work, we plan to study how to set δ effectively for document-centric XML data. In the remaining experiments, we set δ to 0.8 to further evaluate the search performance of our method.

5.2.2. Precision and recall

This section evaluates result quality in terms of precision and recall. Fig. 14 shows the 11-pt precision/recall graph. We observe that MCCTree outperforms the existing methods, and always achieves higher precision than them on whatever values of recall. Moreover, the precision of XSEarch falls sharply with the increase of recall, while that of MCCTree varies slightly.

To further evaluate the effectiveness, we compared the precision in terms of both the disjunctive semantics and conjunctive semantics. Given a keyword query, for the conjunctive semantics, we took the answers of XQuery translated from the query as the accurate answers. For the disjunctive semantics, we enumerated all the combinations of keywords in the query, transformed the query of each combination into its corresponding schema-aware XQuery statement, and took all the answers of XQuery queries as the accurate results. We computed the corresponding precision for different algorithms. Figs. 15 and 16 depict the precision.

We implemented the "optional nodes" for XSEarch to support the disjunctive semantics. We observe that MCCTree achieves better precision for both conjunctive and disjunctive semantics. XRank performs poorly on many queries. Especially, for disjunctive semantics, XRank is far worse than MCCTree. This is because XRank cannot support the disjunctive semantics. In terms of the disjunctive semantics, XSEarch with "optional nodes" achieve higher precision than XRank but is worse than MCCTree. The reason is that their answers are cover trees, which are not as compact as MCCTrees. MCCTree significantly outperforms existing methods on TreeBank in terms of effectiveness as they involve many false negatives. This reflects that our method outperforms the existing approaches on various datasets.

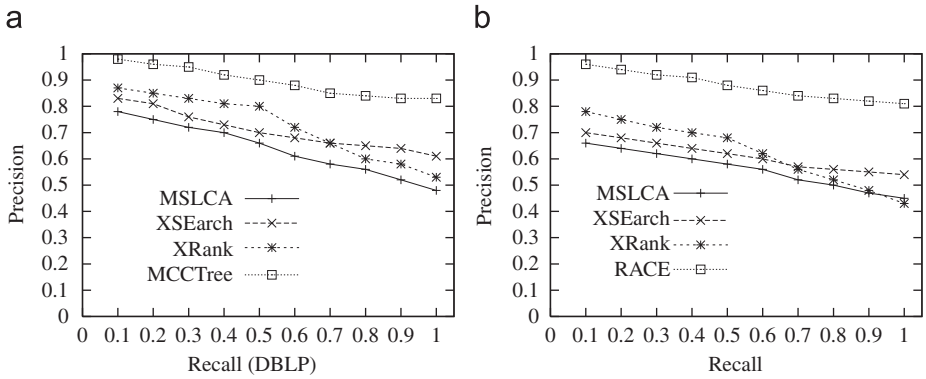

Fig. 14. Precision/recall curves of different algorithms. (a) Recall (DBLP); (b) Recall.

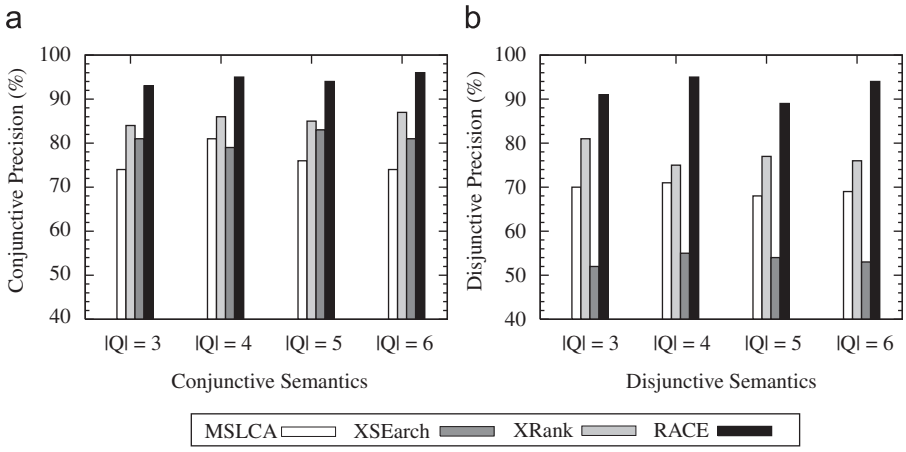

Fig. 15. Precision for conjunctive and disjunctive semantics on DBLP. (a) Conjunctive Semantics; (b) Disjunctive Semantics.

As users are usually interested in the *top-k* answers, we employed *top*-1, *top*-5, *top*-10, *top*-20, *top*-50 and *top*-100 precision to evaluate the selected algorithms. The obtained results of the average *top-k* precision for those 40 queries are illustrated in Table 2. As expected, MCCTree always achieves more than 90% top-*k* precision, which is about 10–35% higher than those of XSEarch and XRank. The reason is that, XSEarch only considers the traditional IR parameters which are ineffective for ranking the tree-structured answers; XRank does not normalize the overall ranking score, since a connected subtree containing more content nodes does not imply the subtree is more meaningful. We normalize the overall ranking score by dividing the number of content nodes and take both structural compactness and textual similarity into account.

5.3. User study

To evaluate the result quality, we conduct a user study. We took a group of 20 randomly selected people and asked them to evaluate the keyword queries in Table 3 against MCCTree, MSLCA, XRank and XSEarch. For each query, each user was asked to describe in natural language the semantics of the query. If there are more than one semantics, the truth is the one provided by the majority of the users. The semantics of all the queries described by natural language is shown in Table 3.

As users are usually interested in the *top-k* answers, we employed *top-k* answer relevancy (the ratio of the number of answers deemed to be relevant in the first *k* results to *k*) to compare those algorithms. The obtained result of the average *top-k* answer relevancy is illustrated in Fig. 17. As expected, MCCTree always achieves more than 80% answer relevancy, which is about 10–40% higher than those of other methods on various queries. This is because MCCTree identifies the compact MCCTrees as the answer and ranks the MCCTrees by taking both structural compactness and textual similarity.

6. Related work

The first area of research related to our work is the computation of the LCA of two or more nodes, which has been studied in [10,34]. As an extension of LCA, meaningful LCA (MLCA), smallest LCA (SLCA), multiway-SLCA

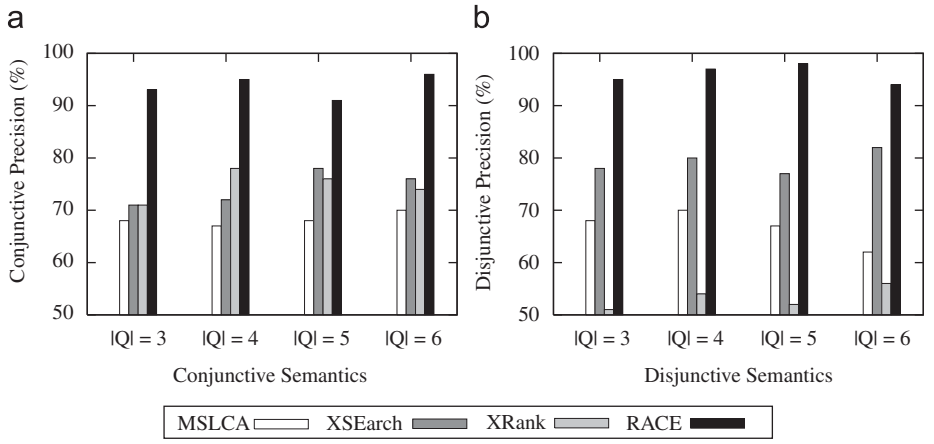

Fig. 16. Precision for conjunctive and disjunctive semantics on TreeBank. (a) Conjunctive Semantics; (b) Disjunctive Semantics.

Table 2
top-k precision.

DBLP	*top*-1 (%)	*top*-5 (%)	*top*-10 (%)	*top*-20 (%)	*top*-50 (%)	*top*-100 (%)
(a) DBLP						
MSLCA	66.5	70.0	71.0	68.0	62.5	66.0
XSEarch	77.5	84.5	81.5	78.0	74.0	70.5
XRank	82.5	83.0	84.5	80.5	78.0	75.0
MCCTree	92.5	94.5	96.0	95.5	96.0	94.5
TreeBank	*top*-1 (%)	*top*-5 (%)	*top*-10 (%)	*top*-20 (%)	*top*-50 (%)	*top*-100 (%)
(b) TreeBank						
MSLCA	61.5	62.0	64.0	58.5	60.5	59.0
XSEarch	71.5	76.0	77.5	77.0	76.0	70.5
XRank	78.5	81.5	80.5	81.5	79.0	75.0
MCCTree	90.0	92.5	94.0	93.5	95.5	94.5

Table 3
Queries for user study.

Q_1	jim gray title	Find titles of papers written by "jim gray"
Q_2	DB IR XML author	Find authors who publish papers about "DB IR XML"
Q_3	jiawei han title year VLDB	Find papers written by "jiawei han" and published in "VLDB"
Q_4	jim gray david title	Find titles of papers co-authored by "jim gray" and "david"
Q_5	jagadish XML ICDE title	Find papers written by jagadish and published in "ICDE"
Q_6	VP DT NN	Find the sentences with "VP DT NN" structures
Q_7	NP VP VBD NNS	Find the sentences with "NP VP VBD NNS" structures
Q_8	NN SBAR WDT	Find the sentences with "NN SBAR WDT" structures
Q_9	IN VB VP	Find the sentences with "IN VB VP" structures
Q_{10}	VB NP NNP POS	Find the sentences with "VB NP NNP POS" structures

(MSLCA), grouped distance minimum connecting tree (GDMCT), valuable LCA (VLCA), and XSeek have been proposed to improve the efficiency and effectiveness of keyword search against LCA in [26,37,35,13,18,28], respectively.

MSLCA employs a strategy of multiway-SLCA to improve search efficiency. GDMCT returns cover trees as answers, which are not so meaningful as MCCTrees, since we integrate the most relevant cover trees together. XRank [9] and XSEarch [6] are two systems facilitating keyword search in XML documents, which return subtrees rooted at LCAs or their variants as answers. XRank presents a ranking method, which, given a tree containing all the keywords, assigns a score to it using an adaptation of PageRank for XML documents. XSEarch focuses on the semantics and the ranking of the results, and during execution it uses an all-pairs interconnection index to check the connectivity between the nodes. It is not efficient for large XML documents. XKeyword [15] is a system that offers keyword proximity search in XML documents that conform to an XML schema, however it has to compute candidate networks and is constrained by the underlying schemas. XSeek [28] generates return nodes, which can be explicitly inferred from keywords or dynamically constructed according to the entities in the data that are relevant to the search. XSeek is orthogonal to our work as it only considers the conjunctive semantics and identify the relevant answers by considering the structural information. Our prior work [18] proposes valuable LCAs to improve the meaningfulness and completeness of answers by considering the XML structures. This manuscript is an extended version of a poster from WWW 2008 [19].

Furthermore, various XML full-text query languages have been proposed [24,25], and a workshop INEX,[5] INitiative for the Evaluation of XML retrieval, aiming at evaluating XML retrieval effectiveness, has also been organized. Two algebras for keyword search over XML documents have been proposed in [2,32]. The XFT algebra

[5] http://inex.is.informatik.uni-duisburg.de/2006/index.html.

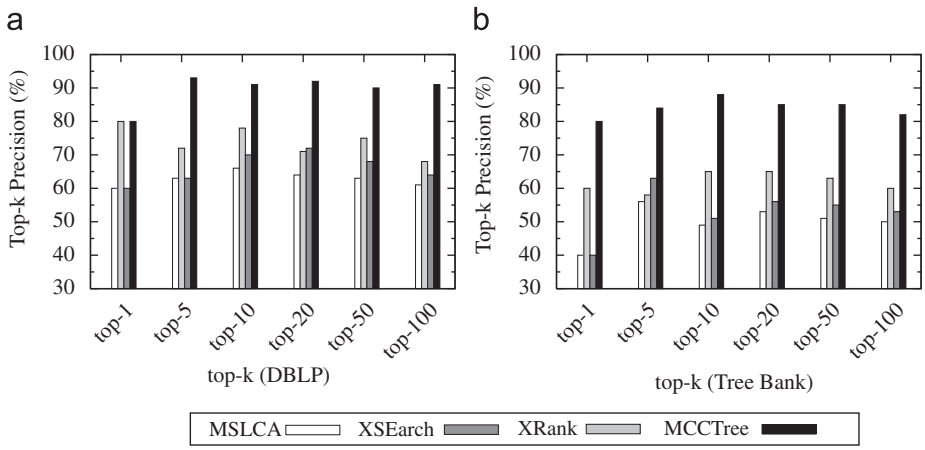

Fig. 17. top-*k* precision for user study. (a) *top-k* (DBLP); (b) *top-k* (TreeBank).

[2] that accounts for element nesting in XML document structure was proposed to evaluate queries with complex full-text predicates, while several optimization techniques that guarantee better efficiency for keyword search over tree-structured documents have been demonstrated in [32].

In addition, there have been many studies of keyword search in relational databases. DISCOVER [14], BANKS [4] and DBXplorer [1] are systems built on top of relational databases. DISCOVER and DBXplorer output trees of tuples connected through primary-foreign key relationships that contain all the keywords of a given keyword query, while BANKS identifies connected trees in a labeled graph by using an approximation to the Steiner tree problem. More recently, Liu et al. [27] proposed a novel ranking strategy to solve the effectiveness problem for relational databases, which employs phrase-based and concept-based models to improve search effectiveness. Kimelfeld et al. [17] demonstrated keyword proximity search in relational databases, which shows that the answer of keyword proximity search can be enumerated in ranked order with polynomial delay, under data complexity. Sayyadian et al. [33] introduced schema mapping into keyword search and proposed a method to answer keyword search across heterogenous databases. Ding et al. [7] employed dynamical programming to improve the efficiency of identifying the Steiner trees while Guo et al. [8] proposed data topology search to retrieve meaningful structures from much richer structural data–biological databases. Our previous work SAILER [20] proposes a unified keyword search framework for unstructured and semi-structured data. We proposed EASE [22] to adaptively and effectively answer keyword search over heterogenous data sources composed of unstructured, semi-structured, and structured data. We proposed compact Steiner tree [23] to approximate Steiner tree problem for efficiently answering keyword queries in relational databases.

7. Conclusion

In this paper, we have investigated the problem of effective keyword proximity search in XML documents, with the aim of identifying the relevant content nodes that contain input keywords, along with a meaningful compact connected trees to describe how each result matches a given keyword query. To improve the search performance of keyword search over XML documents, we first introduce the notions of CLCA and MCLCA to capture the focuses of keyword queries, and then propose two concepts of CCTree and MCCTree to effectively and efficiently answer keyword proximity queries. More importantly, we give the theoretical upper bounds of the numbers of CLCAs, MCLCAs, CCTrees and MCCTrees, respectively, and devise an efficient algorithm to generate the MCCTrees. Furthermore, we present a ranking mechanism to rank the compact connected trees, by taking into consideration both structural compactness from DB point of view and text similarity from IR perspective. Finally, we have conducted an extensive performance study to evaluate the search efficiency and result quality of our method. The experimental results show that our approach achieves both high search efficiency and accuracy for keyword proximity search, and outperforms existing approaches significantly.

In this paper, we focus on data-centric XML documents. For the document-centric XML data, XML content should be more important and we need propose more effective ranking functions. In future work, we will study effective keyword search in document-centric XML data.

Acknowledgments

This work is partly supported by the National Natural Science Foundation of China under Grant No. 60873065, the National High Technology Development 863 Programs of China under Grant No.2007AA01Z152 & 2009AA011906, and the National Grand Fundamental Research 973 Program of China under Grant No.2006CB303103.

References

[1] S. Agrawal, S. Chaudhuri, G. Das, Dbxplorer: a system for keyword-based search over relational databases, in: ICDE, 2002, pp. 5–16.

[2] S. Amer-Yahia, E. Curtmola, A. Deutsch, Flexible and efficient xml search with complex full-text predicates, in: SIGMOD, 2006, pp. 575–586.
[3] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, D. Toman, Structure and content scoring for xml, in: VLDB, 2005.
[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, S. Sudarshan, Keyword searching and browsing in databases using banks, in: ICDE, 2002, pp. 431–440.
[5] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: optimal XML pattern matching, in: SIGMOD, 2002, pp. 310–321.
[6] S. Cohen, J. Mamou, Y. Kanza, Y. Sagiv, Xsearch: a semantic search engine for xml, in: VLDB, 2003, pp. 45–56.
[7] B. Ding, J.X. Yu, S. Wang, L. Qin, X. Zhang, X. Lin, Finding top-k min-cost connected trees in databases, in: ICDE, 2007.
[8] L. Guo, J. Shanmugasundaram, G. Yona, Topology search over biological databases, in: ICDE, 2007.
[9] L. Guo, F. Shao, C. Botev, J. Shanmugasundaram, Xrank: ranked keyword search over xml documents, in: SIGMOD, 2003, pp. 16–27.
[10] D. Harel, R.E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 13 (2) (1984) 338–355.
[11] H. He, H. Wang, J. Yang, P. Yu, Blinks: ranked keyword searches on graphs, in: SIGMOD, 2007.
[12] V. Hristidis, L. Gravano, Y. Papakonstantinou, Efficient ir-style keyword search over relational databases, in: VLDB, 2003, pp. 850–861.
[13] V. Hristidis, N. Koudas, Y. Papakonstantinou, D. Srivastava, Keyword proximity search in xml trees, in: IEEE TKDE, vol. 18, no. 4, 2006, pp. 525–539.
[14] V. Hristidis, Y. Papakonstantinou, Discover: keyword search in relational databases, in: VLDB, 2002, pp. 670–681.
[15] V. Hristidis, Y. Papakonstantinou, A. Balmin, Keyword proximity search on xml graphs, in: ICDE, 2003, pp. 367–378.
[16] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, H. Karambelkar, Bidirectional expansion for keyword search on graph databases, in: VLDB, 2005, pp. 505–516.
[17] B. Kimelfeld, Y. Sagiv, Finding and approximating top-k answers in keyword proximity search, in: PODS, 2006, pp. 173–182.
[18] G. Li, J. Feng, J. Wang, L. Zhou, Efficient keyword search for valuable lcas over xml documents, in: CIKM, 2007.
[19] G. Li, J. Feng, J. Wang, L. Zhou, Race: Finding and ranking compact connected trees for keyword proximity search over xml documents, in: WWW, 2008.
[20] G. Li, J. Feng, J. Wang, L. Zhou, Sailer: an effective search engine for unified retrieval of heterogeneous xml and web documents, in: WWW, 2008.
[21] G. Li, J. Feng, L. Zhou, Progressive ranking for efficient keyword search over relational databases, in: BNCOD, 2008.
[22] G. Li, B.C. Ooi, J. Feng, J. Wang, L. Zhou, Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data, in: SIGMOD, 2008.
[23] G. Li, X. Zhou, J. Feng, L. Zhou, Progressive top-k keyword search in relational database, in: ICDE, 2009.
[24] Y. Li, H. Yang, H.V. Jagadish, Nalix: an interactive natural language interface for querying xml, in: SIGMOD, 2005, pp. 900–902.
[25] Y. Li, H. Yang, H.V. Jagadish, Constructing a generic natural language interface for an xml database, in: EDBT, 2006, pp. 737–754.
[26] Y. Li, C. Yu, H.V. Jagadish, Schema-free xquery, in: VLDB, 2004.
[27] F. Liu, C. Yu, W. Meng, A. Chowdhury, Effective keyword search in relational databases, in: SIGMOD, 2006, pp. 563–574.
[28] Z. Liu, Y. Chen, Identifying return information for xml keyword search, in: SIGMOD, 2007.
[29] J. Lu, T.W. Ling, C.-Y. Chan, T. Chen, From region encoding to extended Dewey: on efficient processing of xml twig pattern matching, in: VLDB, 2005, pp. 193–204.
[30] Y. Luo, X. Lin, W. Wang, X. Zhou, Spark: top-k keyword query in relational databases, in: SIGMOD, 2007.
[31] A. Marian, S. Amer-Yahia, N. Koudas, D. Srivastava, Adaptive processing of top-k queries in xml, in: ICDE, 2005, pp. 162–173.
[32] S. Pradhan, An algebraic query model for effective and efficient retrieval of xml fragments, in: VLDB, 2006, pp. 295–306.
[33] M. Sayyadian, H. LeKhac, A. Doan, L. Gravano, Efficient keyword search across heterogeneous relational databases, in: ICDE, 2007.
[34] B. Schieber, U. Vishkin, On finding lowest common ancestors: simplification and parallelization, SIAM J. Comput. 17 (6) (1988) 1253–1262.
[35] C. Sun, C.Y. Chan, A.K. Goenka, Multiway SLCA-based keyword search in XML data, in: WWW, 2007, pp. 1043–1052.
[36] I. Tatarinov, S. Viglas, K.S. Beyer, J. Shanmugasundaram, E.J. Shekita, C. Zhang, Storing and querying ordered xml using a relational database system, in: SIGMOD, 2002, pp. 204–215.
[37] Y. Xu, Y. Papakonstantinou, Efficient keyword search for smallest LCAs in xml databases, in: SIGMOD, 2005, pp. 527–538.
[38] Y. Xu, Y. Papakonstantinou, Efficient LCA based keyword search in XML data, in: EDBT, 2008.