Effective Keyword Search for Valuable LCAs over XML Documents

Guoliang Li Jianhua Feng Jianyong Wang Lizhu Zhou
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
{*liguoliang,fengjh,jianyong,dcszlz*}*@tsinghua.edu.cn*

ABSTRACT

In this paper, we study the problem of effective keyword search over XML documents. We begin by introducing the notion of Valuable Lowest Common Ancestor (VLCA) to accurately and effectively answer keyword queries over XML documents. We then propose the concept of Compact VLCA (CVLCA) and compute the meaningful compact connected trees rooted as CVLCAs as the answers of keyword queries. To efficiently compute CVLCAs, we devise an effective optimization strategy for speeding up the computation, and exploit the key properties of CVLCA in the design of the stack-based algorithm for answering keyword queries. We have conducted an extensive experimental study and the experimental results show that our proposed approach achieves both high efficiency and effectiveness when compared with existing proposals.

Categories and Subject Descriptors

H.2 [**Database Management**]: Miscellaneous; H.3.3 [**Information Search and Retrieval**]:

General Terms

Algorithms

Keywords

Information Retrieval;XML Keyword Search; LCA; VLCA; CVLCA

1. INTRODUCTION

Keyword search is a proven and widely accepted mechanism for querying in document systems and World Wide Web. The database research community has recently recognized the benefits of keyword search and has been introducing keyword search capability into relational databases [5, 9, 15, 16, 20, 22, 28, 31, 33, 35], XML databases [7, 8, 10, 13, 14, 17, 21, 23, 25, 26, 27, 29, 32, 34, 38] and graphs [19, 24].

Traditional query processing approaches on relational and XML databases are constrained by the query constructs imposed by the language such as SQL and XQuery. Firstly, the query language themselves are hard to comprehend for non-database users. For example, the XQuery is fairly complicated to grasp. Secondly, these query languages require the queries to be posed against the underlying, sometimes complex, database schemas. These traditional querying methods are powerful but unfriendly for day-to-day users. Keyword search is proposed as an alternative means of querying the database, which is simple and yet familiar to most internet users as it only requires the input of some keywords. Although keyword search has been proven to be effective for text documents (e.g. HTML documents), the problem of keyword search on the structured data (e.g. relational databases) and the semi-structured data (e.g. XML databases) is not straightforward nor well studied.

Keyword search in text documents take the documents that are more relevant with the input keywords as the answers, while that on relational databases will take the correlative records of database that contain all the keywords as the answers. However, it still remains an open problem that, for XML database, what should be the answer for keyword search? The notion of Lowest Common Ancestor (LCA) has been introduced to answer keyword queries on XML databases [17]. More recently, XRank, Meaningful LCA (MLCA), Smallest LCA (SLCA), Grouped Distance Minimum Connecting Tree (GDMCT) and XSeek have been proposed to improve the efficiency and effectiveness of keyword search against LCA in [17, 27, 38, 21, 29], respectively.

However, existing proposals on keyword search over XML databases suffer from two problems: meaningfulness and completeness of answers, and the scope of the search. First, the existing approaches, such as SLCA and XRank, return some irrelevant results or false positives as answers and may also miss some results from answers (We will give an example to describe the details in Section 2). The false positive problem renders the results less meaningful while the false negative problem causes the incompleteness of answers. Second, the answer of keyword search should not be limited to just the LCAs of the keywords, and taking only LCAs as the answer of keyword search is inaccurate. Although the subtrees proposed in existing methods [21, 29], which are composed of LCAs and their relevant keywords, may be more meaningful as the answer of keyword search on XML databases, these subtrees are not compact and meaningful enough to answer keyword search and cannot accurately reflect the essence of keyword queries.

In addition, XML documents involve fairly complicated structures, therefore it is fairly difficult to find the meaningful desired data, which still preserves the structure relationship and also conforms to the documents, for users through limited input keywords. Existing studies mainly focus on efficiency of keyword search on XML databases and usually leads to low effectiveness, and accordingly, how to discover the structure clue from the input keywords so as to improve the effectiveness is urgent to investigate. We emphasize the effectiveness of keyword search on XML databases in this paper, which is at least as important as efficiency.

To achieve our goal, we first introduce the notion of elementary type and propose Valuable LCA (VLCA) to effectively and accurately answer keyword queries, which not only improves the accuracy of LCAs by eliminating redundant LCAs that should not contribute to the answer, but also retrieves the false negatives filtered out wrongly by SLCA. We then introduce a novel concept, called compact VLCA (CVLCA), and take compact connected trees as answers of keyword queries. Finally, we propose an effective optimization strategy for computing the CVLCAs and devise an efficient stack-based algorithm as a total solution. To the best of our knowledge, this is the first paper that improves the effectiveness of keyword search over XML documents in considering XML schemas. To summarize, we make the following contributions:

- We introduce the notion of Valuable LCA (VLCA) to answer keyword queries over XML documents. VLCA not only eliminates redundant LCAs but also retrieve relevant answers filtered out wrongly, and thus improves both the accuracy and completeness of keyword search. To further improve the performance of keyword search, we introduce the concepts of compact VLCA (CVLCA) and compact connected trees to efficiently and effectively answer keyword queries.
- We propose an effective optimization strategy to improve the efficiency of computing CVLCAs, and devise an efficient stack-based algorithm, which incurs only one scan of the content nodes that directly contain some input keywords.
- We conducted an extensive performance study using both real and synthetic datasets with various characteristics. The results show that our algorithm achieves high efficiency and effectiveness, and outperforms the existing proposals in terms of elapsed time as well as precision and recall.

The remainder of this paper is organized as follows. We discuss the background and our motivation in Section 2. Section 3 introduces the notion of VLCA. In Section 4, a brute-force algorithm to compute VLCAs is proposed. We introduce an optimization strategy for computing CVLCAs and design an effective stack-based algorithm in Section 5. Extensive experimental evaluations are provided in Section 6. Finally we conclude the paper in Section 7.

2. BACKGROUND AND MOTIVATION

In this section, we review existing related proposals about keyword search, and discuss their weaknesses.

2.1 Notations

We begin first by introducing the XML data model and some 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 node v, $\lambda(v)$ denotes the label/tag of v, $u \prec v$ ($u \succ v$) denotes that node u is an ancestor (descendant) of node v, while $u \preceq v$ denotes that $u \prec v$ or u=v. $u<v$ ($u>v$) denotes that u precedes (follows) v in the XML document, that is, u is before (after) v in document order, but not an ancestor (descendant) of v. For example, in Figure 1(a), `paper`(5)$\preceq$ `author`(7), `paper`(12)$\succ$`conf`(2), `chair`(19) <`name`(21), and `author`(26)>`year`(22).

Given a keyword query $\mathcal{K}$=$\{k_1,k_2,\cdots,k_m\}$ and an input XML document $\mathcal{D}$, we use $\mathcal{I}_i$ to denote the keyword list of k_i, i.e., the list of nodes which directly contain k_i, and we call each node in $\mathcal{I}_i$ the content node w.r.t. k_i. $\mathcal{I}_i$ can be retrieved by using the well-known inverted indices.

2.2 Motivation

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 [18, 36]. As an extension of LCA, XRank, MLCA, SLCA, GDMCT and XSeek have recently been proposed to answer keyword queries over XML documents respectively in [17], [27], [38], [21] and [29]. We begin by formally introducing the concept of Lowest Common Ancestor (LCA).

DEFINITION 2.1. *(LCA) Given m nodes $n_1,n_2,\cdots,n_m$, v is called LCA of these m nodes, iff, $\forall 1\leq i\leq m$, v is an ancestor of node n_i, and $\nexists u, v\prec u$, u is also an ancestor of each n_i, denoted as v=LCA$(n_1,n_2,\cdots,n_m)$.*

DEFINITION 2.2. *(`LCASet`) Given a query $\mathcal{K}$=$\{k_1,k_2,\cdots,k_m\}$ and an XML document $\mathcal{D}$. The set of LCAs of $\mathcal{K}$ on $\mathcal{D}$ is, `LCASet`= LCA$(\mathcal{I}_1, \mathcal{I}_2,\cdots,\mathcal{I}_m)$=$\{v|v$=LCA$(v_1, v_2,\cdots,v_m)$, $v_i\in\mathcal{I}_i(1\leq i\leq m)\}$.*

The basic idea of LCA is that, given a keyword query $\mathcal{K}$=$\{k_1, k_2,\cdots,k_m\}$ and for $1\leq i\leq m$, node n_i is a content node w.r.t. k_i, if v=LCA$(n_1,n_2,\cdots,n_m)$, then v contains all the input keywords and should be an answer of $\mathcal{K}$. For example, a user wants to search for the *paper* with the *title* containing "IR" and one *author* being "John" in the XML document in Figure 1(a), and he/she can input query {"IR","John"}. Node *paper*(15) is a LCA of the two input keywords, and it should be an answer [*] of this keyword query.

However, there is a problem of LCA. For example, consider another keyword query {"IR","Tom"}, the nodes, *conf*(2),*paper*(12) and *paper*(15), circled by the rectangle in Figure 1(a), are the LCAs (especially, *conf*(2) is the LCA of *title*(13) and *author*(17), which directly contain "IR" and "Tom" respectively). However, it is easy to figure out that *conf*(2) should not be an answer of this keyword query, as *title*(13) and *author*(17) do not belong to the same paper. On the contrary, *paper*(12) and *paper*(15) should be the two results.

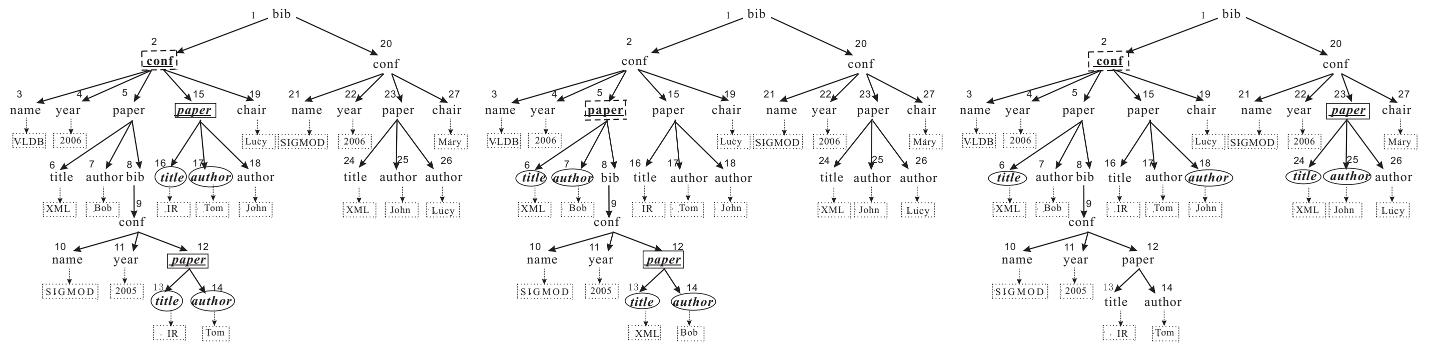
To address the false positive problem of LCA, Meaningful LCA (MLCA)[27], Smallest LCA (SLCA) [38] and XRank [17] have been proposed. There is a difference between MLCA and other methods. MLCA assumes that the users have some knowledge of XML structures and incorporates a new function `mlcas` to compute all the MLCAs into XQuery, and it is not a pure keyword search method. XSeek [29] studies the problem of how to infer the *return clause* for keyword search, however XSeek is orthogonal to our method as we mainly study the meaningfulness and completeness of keyword search in this paper. XRank and SLCA are much related to our work, we here only briefly introduce SLCA, which is defined as follows.

DEFINITION 2.3. *(`SLCASet`) Given a keyword query $\mathcal{K}$=$\{k_1, k_2,\cdots,k_m\}$ and an XML document $\mathcal{D}$, the set of SLCAs of $\mathcal{K}$ on $\mathcal{D}$ is, `SLCASet`=SLCA$(\mathcal{I}_1, \mathcal{I}_2,\cdots,\mathcal{I}_m)$=$\{v|v\in$LCA$(\mathcal{I}_1,\mathcal{I}_2,\cdots,\mathcal{I}_m)$, and $\nexists u,v\prec u$, $u\in$LCA$(\mathcal{I}_1, \mathcal{I}_2,\cdots, \mathcal{I}_m)$.$\}$.*

The basic idea behind SLCA is that, if node v contains all the input keywords, its ancestors will be less meaningful than v. Hence, SLCA introduces the concept of the smallest tree, which is a tree that contains all the keywords but contains no subtrees which also contain all the keywords. For example, although *conf*(2)$\in$ `LCASet` of query {"IR", "Tom"} on the XML document in Figure 1(a), it is not in `SLCASet`, as *paper*(15)$\in$`LCASet` and *conf*(2)$\prec$*paper*(15). Hence, `SLCASet`={*paper*(12), *paper*(15)} of this keyword query.

However, there are still two problems, false positive (i.e., taking some irrelevant nodes as answers) and false negative (i.e., missing some correct results from answers), of SLCA. For example, in

[*] *paper*(15) itself cannot be taken as an answer and the compact connected subtree rooted at this *paper* should be the answer, which will be introduced in details later.



(a) False positives of LCA (b) False negatives of SLCA (c) False positives of LCA and SLCA

Figure 1: False positive and false negative problems introduced by LCA and SLCA

Figure 1(b), consider query {"XML","Bob"} issued on the XML document, *paper*(12) should be an answer, while, *paper*(5), which also contains the two keywords, should be another answer. However, *paper*(5) is an ancestor of *paper*(12) (*paper*(5)$\prec$ *paper*(12)), therefore *paper*(5) will not be a valid SLCA according to Definition 2.3. Hence, *paper*(5) will not be in `SLCASet` and will be absent from the final answers of SLCA. Therefore, SLCA causes a serious problem, false negative. For another example, suppose a keyword query, {"XML", "John"} is issued on the XML document in Figure 1(c), *conf*(2) and *paper*(23), the nodes enclosed by rectangles, are both their SLCAs. It is easy to figure out that *conf*(2) should not be the answer of this keyword query, because *title*(6) and *author*(18) do not belong to the same paper. Therefore, SLCA causes another problem, false positive. We mainly address these two problems in this paper as accurate answers to keyword queries are vital to the meaningfulness of keyword search.

2.3 Other Related Work

DISCOVER [22], BANKS [9] and DBXplorer [5] 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 the query, while BANKS identifies connected trees in a labeled graph by using an approximation to the Steiner tree problem, which is an NP-hard problem. Liu et al [28] proposed a novel ranking strategy to solve the effectiveness problem for relational database, which employs phrase-based and concept-based models to improve search effectiveness. Luo et al [31] presented a more effective ranking method on relational databases by adopting the concept of virtual document. More recently, Sayyadian et al [35] introduced schema mapping into keyword search and proposed a method to answer keyword search across heterogenous databases. Ding et al [15] employed dynamical programming to improve the efficiency of identifying the Steiner trees, while Guo et al [16] proposed data topology search to retrieve meaningful structures from much richer structural data – biological databases. [24] and [19] studied the problem of keyword search over graphs by employing the techniques of bidirectional expansion and graph partition respectively.

XRank [17] and XSEarch [14] are systems facilitating keyword search for XML databases, which return subtrees as answers for the keyword queries. However, XRank does not return connected trees to explain how the keywords are connected to each other. Furthermore, only the most specific result is output. They also present a ranking method which, given a tree $\mathcal{T}$ containing all the keywords, assigns a score to $\mathcal{T}$ using an adaptation of PageRank for XML databases. Their ranking techniques are orthogonal to the retrieval and, hence, can easily be incorporated in other works. 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, which are very inefficient for large XML documents. Hristidis et al [21] proposed GDMCT, which generated the grouped subtrees to answer keyword queries.

In addition, various XML full-text query languages have also been proposed, such as, [10, 25, 26], and a workshop INEX [2], INitiative for the Evaluation of XML Retrieval, aiming at evaluating XML retrieval effectiveness, has also been organized. More recently, two algebras for keyword search over XML documents have been proposed in [7, 34]. [7] presented the XFT algebra that accounts for element nesting in XML document structure to evaluate queries with complex full-text predicates, while [34] demonstrated several optimization techniques that guarantee better efficiency for keyword search over tree-structured documents.

3. VLCA AND MDC

In this section, we introduce the notion of Valuable Lowest Common Ancestor (VLCA), and illustrate how to employ this semantics in effectively answering keyword queries. Moreover, we propose a novel numbering scheme of Meaningful Dewey Code (MDC) to accelerate computing VLCAs.

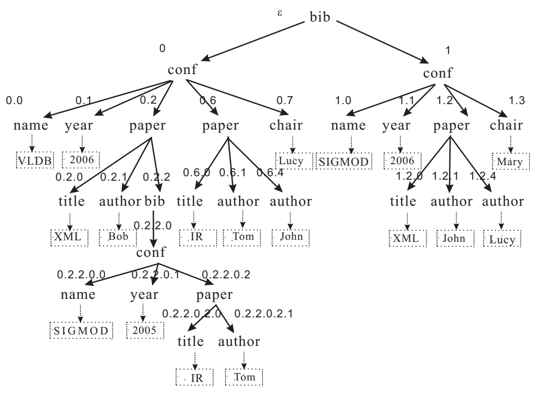
3.1 VLCA

The problem of SLCA is that, it will take SLCA of content nodes w.r.t. given keywords, even if certain content nodes may be irrelevant between each other, as answers, and this will violate the constraint of the DTD (or schema) w.r.t. an XML document and lead to some errors. For example, in Figure 2(c), although *conf*(2) is SLCA (LCA) of *title*(6) and *author*(18), *conf*(2) should not be an answer of the keyword query {"XML", "John"}, because they are descendants of different *papers*. Based on above observations, we propose the concept of Valuable LCA (VLCA) to address this problem and we begin by introducing two notions as follows.

DEFINITION 3.4. *(Elementary Type) An Elementary Type of a node in an XML document is its label/tag in the DTD (or schema).*

DEFINITION 3.5. *(Homogenous/Heterogenous) Given two nodes u,v, and w=LCA(u,v). `uSet` and `vSet` are two sets of nodes in the paths of $w \longrightarrow u$ and $w \longrightarrow v$ respectively. Let `wSet`=`uSet`$\cup$`vSet`. u and v are heterogenous (denoted as $u \not\sim v$), iff, $\exists u' \in$`wSet`, $v' \in$`wSet`, u' and v' are of the same elementary type, i.e., $\lambda(u')=\lambda(v')$, except u and v maybe having the same elementary type. On the contrary, u and v are homogenous (denoted as $u \sim v$), iff, there are no two nodes of the same elementary type, except u and v.*

EXAMPLE 3.1. *Node `paper(5)` and node `paper(15)`, in Figure 1 (c), are of the same elementary type as they have the same*



(a) An example XML document

```
<!ELEMENT bib (conf)*>
<!ELEMENT conf (name,year,paper*,chair)>
<!ELEMENT paper (title,author+,bib?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT chair (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
```

(b) DTD of the XML document in (a)

Figure 2: An example XML document and its corresponding DTD

label in DTD. LCA of `title`*(6) and* `author`*(18) is* `conf`*(2). As* `paper`*(5) and* `paper` *(15) are in the paths,* `conf`*(2)→*`title`*(6) and* `conf`*(2)→*`author`*(18) respectively, so* `title`*(6)≁*`author`*(18). On the contrary,* `author`*(17)∼*`author`*(18), because there are no other two nodes in {*`paper`*(15),* `author`*(17),* `author`*(18)} with the same elementary type, except themselves.*

Based on the concept of homogenous/heterogenous, we introduce a novel semantics, Valuable LCA, to answer keyword search.

DEFINITION 3.6. *(Valuable LCA) Given m nodes n_1,n_2,$\cdots$, n_m, v=LCA(n_1,n_2,$\cdots$, n_m). VLCA(n_1,n_2,$\cdots$,n_m)=v, iff, these m nodes are homogenous, that is, $\forall 1 \leq i < j \leq m$, $n_i \sim n_j$.*

DEFINITION 3.7. *(*`VLCASet`*) Given query $\mathcal{K}$=$\{k_1, k_2,\cdots,k_m\}$ and an input XML document $\mathcal{D}$. The set of VLCAs w.r.t. $\mathcal{K}$ and $\mathcal{D}$, is,* `VLCASet`*=VLCA($\mathcal{I}_1$,$\cdots$,$\mathcal{I}_m$)=$\{v | v$=VLCA$(v_1,\cdots,v_m), v_i \in \mathcal{I}_i\}$.*

`LCASet` is the set of the nodes, which contain all of the input keywords. `SLCASet` is a subset of `LCASet`, which eliminates the LCAs that have LCA descendants. `VLCASet` can avoid the false positives and false negatives introduced by SLCA, and is a more accurate subset of `LCASet` to answer keyword queries. To describe how each result matches a keyword query, we introduce a meaningful concept as defined in Definition 3.8. Each result is represented as a connected subtree, which is rooted at a VLCA and contains the corresponding content nodes so that each result is self-explained and can explain how it matches the keyword query.

DEFINITION 3.8. *(Answer of keyword search) Given a query $\mathcal{K}$=$\{k_1,k_2,\cdots,k_m\}$ and an XML document $\mathcal{D}$. $\mathcal{K}_{(\mathcal{D},(k_1,k_2,\cdots,k_m))}$ is the answer of $\mathcal{K}$ on $\mathcal{D}$, where $\mathcal{K}_{(\mathcal{D},(k_1,k_2,\cdots,k_m))}$=$\{(r;\lambda(v_1)$:$c_1$, $\lambda(v_2)$:c_2,$\cdots$, $\lambda(v_m)$:$c_m)$ $|$ $\forall v_i \in \mathcal{I}_i$, r=VLCA(v_1, v_2,$\cdots$, v_m), where c_i denotes the text content that v_i contains.$\}$.*

Existing proposals on keyword search usually focus on effectively computing `LCASet` or `SLCASet`, which cannot describe how each result matches a keyword query. In contrast, the tuple (r; $\lambda(v_1)$:c_1, $\lambda(v_2)$:c_2, $\cdots$, $\lambda(v_m)$:c_m) in our method, not only preserves the structure information (r and v_i), but also reflects the user desired data (c_i). However, to compute VLCAs, we need to determine whether two nodes are homogeneous, which is cost-efficient through navigating the XML document, while the all-pairs interconnection index proposed in [14] is very expensive for large XML documents. Therefore, Meaningful Dewey Code (MDC) is introduced to address this issue in next sections.

$$\mathcal{O}_n = \begin{cases} k & \textit{if n is the first child of parent}(n) \\ \mathcal{O}_{presib(n)}+k-\mathcal{O}_{presib(n)}\%m & \textit{else if } \mathcal{O}_{presib(n)}\%m < k \\ \mathcal{O}_{presib(n)}+m+k-\mathcal{O}_{presib(n)}\%m & \textit{otherwise} \end{cases} \tag{3-1}$$

$$\mathcal{C}_n = \begin{cases} \epsilon & \textit{if n is the root node} \\ \mathcal{C}_{parent(n)} \circ \mathcal{O}_n{}^{\dagger} & \textit{otherwise} \end{cases} \tag{3-2}$$

3.2 MDC

To effectively compute VLCAs, we introduce a novel numbering scheme, MDC, which is inspired from Dewey code in [30], [37]. Generally, there is a corresponding DTD (or schema) associated with an XML document, which describes the document type definition. A DTD is typically much smaller than its corresponding XML document, therefore it is easier to be manipulated. Even if there does not exist a DTD, we can extract one from the XML document. For example, in Figure 2, (a) is an XML document and (b) is the DTD extracted from this XML document.

Given an XML document, we number/encode its nodes based on the corresponding DTD. Let $parent(n)$ denote the parent of n and $presib(n)$ denote the preceding sibling (neighboring) of n. Suppose the MDC of the root is ϵ and $\mathcal{C}_n$ denotes the MDC of node n, we can encode each node from the root to the leaf as follows: for any node n, its MDC is its parent's MDC, $\mathcal{C}_{parent(n)}$, concatenated with an assigned and ordered number $\mathcal{O}_n$, i.e., $\mathcal{C}_n$=$\mathcal{C}_{parent(n)} \circ \mathcal{O}_n$. Without loss of generality, let l_0,l_1,...,l_{m-1} denote m distinct labels of the children of $parent(n)$ in DTD and $\lambda(n)$ is the k-th label, we can compute $\mathcal{O}_n$ and $\mathcal{C}_n$ through Equations (3-1) and (3-2) respectively. It is obvious that $\mathcal{O}_n\%m$=k, that is, the siblings with the same label will get the same remainder when divided by the total number of distinct labels among their siblings. More importantly, MDC captures the following properties:

i) Node a is an ancestor of node d, iff, $\mathcal{C}_a$ is a prefix of $\mathcal{C}_d$. a is the parent of d, iff, $\mathcal{C}_a$ is a prefix of $\mathcal{C}_d$ and $|\mathcal{C}_a|$=$|\mathcal{C}_d|$-1, where $|\mathcal{C}_v|$ denotes the length of $\mathcal{C}_v$, i.e., the depth of node v in the XML document tree.

ii) Node a follows (or precedes) node b iff $\mathcal{C}_a$ is greater (or smaller) than $\mathcal{C}_d$ in lexicographical order.

iii) Given the MDC of a node, we can deduce its ancestors' MDCs and elementary types based on the numbering scheme.

[†] a ∘ b denotes another code constructed by concatenating a and b with a delimiter (e.g. a dot) between them.

LEMMA 3.1. *Given two nodes, u and v, w=LCA(u,v), iff,* $\mathcal{C}_w$ *=LCP(*$\mathcal{C}_u$,$\mathcal{C}_v$*), where LCP(*$\mathcal{C}_u$,$\mathcal{C}_v$*) denotes the longest common prefix of* $\mathcal{C}_u$ *and* $\mathcal{C}_v$.

LEMMA 3.2. *Given m nodes,* $v_1,\cdots,v_m$, *w=LCA(*$v_1,v_2,\cdots,v_m$*), iff,* $\mathcal{C}_w$*=LCP(*$\mathcal{C}_{v_1}$, $\mathcal{C}_{v_2},\cdots,\mathcal{C}_{v_m}$*), where LCP(*$\mathcal{C}_{v_1},\mathcal{C}_{v_2},\cdots,\mathcal{C}_{v_m}$*) denotes the longest common prefix of* $\mathcal{C}_{v_1},\mathcal{C}_{v_2},\cdots,\mathcal{C}_{v_m}$.

i) and ii) are obvious according to the encoding strategy. Based on them, we introduce Lemma 3.1 and Lemma 3.2 to compute the LCA of two or more nodes. iii) is the key property of MDC different from the general Dewey code. Given the MDC of node n, we demonstrate how to infer the labels of the ancestors of n as follows. Since MDC of the root is always ϵ, we deduce the labels iteratively form the root, and here we only introduce how to infer the label of a given node according to its MDC and its parent's label. Consider the MDC of node n is $\mathcal{C}_{parent(n)} \circ \mathcal{O}_n$, and its parent's label, $\lambda(parent(n))$, has been obtained through iteration. Suppose the distinct labels of $parent(n)$'s children are, orderly, l_0,l_1,...,l_{m-1}, which can be gotten from the corresponding DTD. We can compute the order of $\lambda(n)$ among these labels, i.e., $\mathcal{O}_n\%m$, and accordingly get its label, i.e., $l_{\mathcal{O}_n\%m}$. Hence, given the MDC of a node, we can deduce the labels of its ancestors from the root to itself iteratively.

When checking whether two nodes u,v are homogenous or not, we first compute their LCA, w, based on LEMMA 3.1, and then deduce the labels of nodes on $w\rightarrow u$ and $w\rightarrow v$ based on the property iii) of MDC, finally check whether there are two distinct nodes with the same elementary type in the two paths. Moreover, we introduce an optimization technique to check whether two nodes are homogenous or not as stated in LEMMA 3.3.

LEMMA 3.3. *Given two nodes u,v, w=LCA(u,v),* `uSet`*,* `vSet` *are two sets of the nodes in the paths of w→u and w→v respectively. Let* `wSet`*=*`uSet`∪`vSet`*,* `lSet`*=*$\{\lambda(u)|u\in$`wSet`$\}$*. u and v are heterogenous, iff,* |`wSet`|-|$\{\lambda(u)\}\cap\{\lambda(v)\}$|>|`lSet`|*, while u and v are homogenous, iff,* |`wSet`|-|$\{\lambda(u)\}\cap\{\lambda(v)\}$|=|`lSet`|.

PROOF. If u and v have the same elementary type, $\lambda(v)$=$\lambda(u)$, thus |$\{\lambda(u)\}\cap\{\lambda(v)\}$|=1; otherwise, |$\{\lambda(u)\}\cap\{\lambda(v)\}$|=0. Since |`lSet`| is the number of the distinct labels in `wSet`, there are two same labels in `wSet` except u and v, iff, |`wSet`|-|$\{\lambda(u)\}\cap\{\lambda(v)\}$| -|`lSet`|>0. Similarly, u and v are heterogenous, iff, |`wSet`|-|$\{\lambda(u)\}\cap\{\lambda(v)\}$|>|`lSet`| according to Definition 3.5. □

LEMMA 3.4. *Given m nodes* $v_1,v_2,\cdots,v_m$*, w=LCA(*$v_1,\cdots,v_m$*),* `vSet`$_i$ *is the set of nodes in the path* $w\rightarrow v_i$*. Let* `wSet`=$\cup_{i=1}^{m}$`vSet`$_i$*,* `lSet`*=*$\{\lambda(u)|u\in$`wSet`$\}$*. The m nodes are heterogenous, iff,* |`wSet`|-(m-|$\cup_{i=1}^{m}\{\lambda(v_i)\}$|)>|`lSet`|*, while these m nodes are homogenous iff* |`wSet`|-(m-|$\cup_{i=1}^{m}\{\lambda(v_i)\}$|)=|`lSet`|.

PROOF. It is easy figure out that |$\cup_{i=1}^{m}\{\lambda(v_i)\}$| is the the number of distinct labels in $\{v_1,v_2,\cdots,v_m\}$, m-|$\cup_{i=1}^{m}\{\lambda(v_i)\}$| is the number of duplicate labels in $\{v_1,v_2,\cdots,v_m\}$, and |`lSet`| is the number of distinct labels in `wSet`. Hence, there are two same labels except the m nodes, iff, |`wSet`|-(m-|$\cup_{i=1}^{m}\{\lambda(v_i)\}$|)-|`lSet`|>0. Similarly, the m nodes are heterogenous, iff, |`wSet`|-(m-|$\cup_{i=1}^{m}\{\lambda(v_i)\}$|) >|`lSet`|, according to Definition 3.5. □

We can adopt LEMMA 3.3 to check whether two nodes are homogenous or not. Consider d is the depth of an XML document, the complexities of computing `uSet` and `vSet` are both O(d), while the complexity of computing `wSet` through merging them is also O(d). To compute `lSet`, we need to eliminate the duplicates and the complexity is O($d$log(d)). To sum up, the complexity of checking whether two nodes are homogenous or not is O($d$log(d)) as formalized in LEMMA 3.3. Consider checking whether m nodes are homogenous, we need to check whether any two nodes among them are homogenous. As there are $\binom{m}{2}$ combinations, the complexity is O($\binom{m}{2}d$log(d))=O(m^2dlog(d)). We can reduce it to O(mdlog(md)) as formalized in LEMMA 3.4, where the complexities of computing `vSet`$_i$, `wSet` and `lSet` are O(d), O($m$log(m)d) and O(mdlog(md)) respectively. To further illustrate how to compute VLCAs, we give a running example as shown in Example 3.2.

Table 1: The inverted index (in part) of the XML document in Figure 2(a)

Keywords	MDC($\mathcal{I}_i$)
XML	0.2.0;1.2.0
John	0.6.4;1.2.1
IR	0.2.2.0.2.0;0.6.0
Bob	0.2.1

EXAMPLE 3.2. *Consider the XML document in Figure 2(a) and its DTD in Figure 2(b). MDC of* `conf`*, i.e., the first child of the root node(*`bib`*), is 0 according to Equation (3-2). Node* `conf(0)` *has four child labels,* `name`*,*`year`*,*`paper`*,*`chair`*, thus m=4. As* `name` *is the first child of* `conf(0)`*, so* $\mathcal{O}_{name}$*=0 according to Equation (3-1) and* $\mathcal{C}_{name}$*=0.0. As* `year` *is the second child label of* `conf(0)` *and* $\mathcal{O}_{name}\%m$=0<k(1)*, so* $\mathcal{O}_{year}$=$\mathcal{O}_{name}$+k(1)-$\mathcal{O}_{name}\%m$=1 *and* $\mathcal{C}_{year}$*=0.1. Similarly,* $\mathcal{O}_{paper^1}$=2 [‡], $\mathcal{C}_{paper^1}$*=0.2. As* $\mathcal{O}_{paper^1}\%m$*=k(2),* $\mathcal{O}_{paper^2}$=$\mathcal{O}_{paper^1}$+m+k(2)-$\mathcal{O}_{paper^1}\%m$*=6 based on Equation (3-1), hence* $\mathcal{C}_{paper^2}$*=0.6. Accordingly, we can encode the nodes in the document as illustrated in Figure 2(a).*

Given an MDC, 0.6.1, its ancestors' MDCs are ϵ, 0, 0.6. *As* |0.6|(2)=|0.6.1|(3)−1, 0.6 *is the parent of* 0.6.1. *More importantly, we can deduce the labels of* 0.6.1*'s ancestors based on the DTD. Since the root is* `bib`*, the label of* 0.6.1*'s ancestor at level 0 is* `bib` *(the level of the root is 0).* `bib` *has only one child label, i.e.,*`conf`*, in the DTD, and as the assigned and ordered number of* 0.6.1*'s ancestor at level* 1 *(*$\mathcal{O}_1$*) is 0, and* $\mathcal{O}_1\%1$*=0, so the ancestor of* 0.6.1 *at level* 1 *is* `conf`*. Node* `conf(0)` *has four (m=4) distinct child labels, orderly* `name`*,* `year`*,* `paper`*,* `chair` *according to the DTD. As the assigned and ordered number of* 0.6.1*'s ancestor at level 2 (*$\mathcal{O}_2$*) is 6, and* $\mathcal{O}_2\%m$*=6%4=2, so the ancestor of* 0.6.1 *at level 2 is* `paper`*. In the same way, as* `paper` *has three distinct child labels, orderly* `title`*,* `author`*,* `bib`*, and* $\mathcal{O}_3\%m$*=1%3=1, so the label (at level 3) of* 0.6.1 *is* `author`*. Therefore, the labels of* 0.6.1*'s ancestors form the root to itself, are* `bib`*,* `conf`*,*`paper`*,* `author`*.*

When checking whether u(0.2.0) and v(0.6.4) are homogeneous or not, we first compute their LCA, w($\mathcal{C}_w$*=0), and then deduce the labels of the nodes on the paths w→u and w→v, which are* `conf`*,* `paper`*,* `title` *and* `conf`*,* `paper`*,* `author` *respectively. Accordingly we can get* `wSet`*,* {`conf`*(0),*`paper`*(0.2),*`title`*(0.2.0),* `paper`*(0.6),*`author`*(0.6.4)} and* `lSet`*=*{`conf`*,*`paper`*,* `author`*,* `title`}*. As* |`wSet`|*=5,* |`lSet`|*=4, and* |$\{\lambda(u)\}\cap\{\lambda(v)\}$|*=0, u and v are heterogenous based on* LEMMA *3.3.*

4. THE BRUTE-FORCE ALGORITHM

This section presents a brute-force algorithm to compute VLCAs and the answers of keyword queries. Constructing the inverted index for the keywords in the XML document is proved to be an efficient way [38]. In our approach, we also employ the inverted index. The content nodes w.r.t. a keyword are indexed together and sorted in ascending order by their MDCs, as shown in Table 1. Accordingly, given a keyword query $\mathcal{K}$=$\{k_1,k_2,\cdots k_m\}$, we first retrieve

[‡] $paper^1$ refers to the first `paper` of `conf`(0), while $paper^2$ refers to the second `paper` of `conf`(0).

Algorithm 1: Brute-force Algorithm

```
Input: $\mathcal{K}$={$k_1$,$k_2$,$\cdots$ $k_m$} and an XML document $\mathcal{D}$
Output: KwRst=$\mathcal{K}_{(\mathcal{D},(k_1,k_2,\cdots,k_m))}$={($r$; $\lambda(v_1)$:$c_1$,
        $\lambda(v_2)$:$c_2$,$\cdots$,$\lambda(v_m)$:$c_m$), $\cdots$}
1  begin
2      KwRst←$\phi$;
3      getNodeLists(); /*$\mathcal{I}_i$={$v_i$|$v_i$ directly contains keyword $k_i$}*/
4      for each combination ($v_1$,$v_2$,$\cdots$,$v_m$) [$v_i$∈$\mathcal{I}_i$] do
5          if $v_1$,$v_2$,$\cdots$,$v_m$ are heterogenous then
6              continue;
7          else
8              $r$=VLCA($v_1$,$v_2$,$\cdots$,$v_m$);
               /*$\mathcal{C}_r$ is the longest common prefix of $\mathcal{C}v_1$,$\mathcal{C}_{v_2}$,$\cdots$,$\mathcal{C}v_m$.*/
9              KwRst←($r$; $\lambda(v_1)$:$c_1$,$\lambda(v_2)$:$c_2$,$\cdots$, $\lambda(v_m)$:$c_m$);
               /*$c_i$ is the text content that $v_i$ contains.*/
10 end
```

Figure 3: The brute-force algorithm to answer keyword search

each content node list, $\mathcal{I}_i$ w.r.t. k_i, through our inverted index, and then enumerate all the combinations of the nodes in each $\mathcal{I}_i$, i.e., (v_1,v_2,$\cdots$,v_m), and subsequently compute VLCA of the nodes in each combination according to Lemma 3.4. Finally, we return the result set, $\mathcal{K}_{(\mathcal{D},(k_1,k_2,\cdots,k_m))}$ as described in Definition 3.8.

We devise the brute-force algorithm as shown in Figure 3. It first retrieves the inverted list $\mathcal{I}_i$ for each keyword k_i (line 3), then for each combination of nodes in $\mathcal{I}_i$ (line 4), if the nodes in the current combination are heterogenous, this combination will not constitute an answer and will be skipped (line 5), otherwise it will be an answer and added to the result set (lines 8-9). To further understand the algorithm, we walk through the algorithm with a running example as described in Example 3.3.

EXAMPLE 3.3. *Consider the keyword query {"XML", "John""} on the XML document in Figure 2(a). We first retrieve the inverted lists (as shown in Table 1), i.e., $\mathcal{I}_{XML}$={0.2.0;1.2.0}, $\mathcal{I}_{John}$={0.6.4; 1.2.1}. For combinations, (0.2.0;0.6.4), (0.2.0;1.2.1), (1.2.0;0.6.4) are not results of this keyword search because the two nodes in these tuples are heterogenous. Only node 1.2 is a VLCA of the combination (1.2.0;1.2.1), and the three nodes will constitute the answer of this keyword query.*

We analyze the complexity of the brute-force algorithm. There are $\prod_{i=1}^{m}(|\mathcal{I}_i|)$ combinations for the content nodes, where $|\mathcal{I}_i|$ is the number of content nodes w.r.t. k_i. For each combination, (v_1,v_2,$\cdots$,v_m), we need to compute their LCA and check whether these m nodes are homogenous or not, and the complexity of the former is O(md) according to Lemma 3.2, while that of the latter is $mdlog(md)$ according to Lemma 3.4. To sum up, the complexity of our algorithm is O($mdlog(md)\prod_{i=1}^{m}(|\mathcal{I}_i|)$). Since m and d are small integers, $mdlog(md)$ is dominated by $\prod_{i=1}^{m}(|\mathcal{I}_i|)$. When there are many content nodes w.r.t. the input keywords, the brute-force algorithm is inefficient, and we will introduce a more efficient stack-based algorithm to address this issue in Section 5.

5. THE STACK-BASED ALGORITHM

As above observation, if there are a large number of content nodes associated with the input keywords, the brute-force algorithm based on exhaustive enumeration is inefficient. To improve the search efficiency, in this section, we propose a stack-based algorithm `VLCAStack` to address this issue in this section. `VLCAStack` is inspired from the stack-based family of algorithms for structure join and twig join [6, 11, 12], however, our method is orthogonal to them in that they have to deal with the complicated structure relationships (ancestor-descendant or parent-child relationships). In addition, our method is different from the existing studies on keyword search algorithms, such as MLCA [27], SLCA [38], GDMCT [21] and XRank [17]. The difference is that, SLCA and XRank employ the general Dewey code and will involve false negatives and false positives as discussed in Section 2, MLCA requires some knowledge of XML structures and incorporates keyword search into XQuery, and GDMCT groups the candidate nodes to compute LCAs and ranks them through the distances of the connected trees rooted at their LCAs. MLCA and GDMCT employ the region-based code, and we will experimentally demonstrate that they are not so efficient as our algorithm based on MDC.

To speed up computing VLCAs, we introduce the notions of Compact LCA and Compact VLCA (CVLCA). CVLCA is more compact than VLCA, and the connected subtree rooted at CVLCA, called compact connected subtree, is more compact and meaningful to answer keyword queries.

DEFINITION 5.9. *(Compact LCA and Compact VLCA) Given m nodes, v_1∈$\mathcal{I}_1$,v_2∈$\mathcal{I}_2$,$\cdots$,v_m∈$\mathcal{I}_m$, w=LCA(v_1,v_2,$\cdots$,v_m). w is called to dominate v_i, if w⪰LCA(v'_1,v'_2,$\cdots$,v'_{i-1},v_i, v'_{i+1},$\cdots$,v'_m), ∀v'_1∈$\mathcal{I}_1$,$\cdots$,v'_{i-1}∈$\mathcal{I}_{i-1}$, v'_{i+1}∈$\mathcal{I}_{i+1}$,$\cdots$,v'_m∈$\mathcal{I}_m$. w is a Compact LCA w.r.t. these m nodes, if w dominates each v_i. w is a Compact VLCA (CVLCA), if w is a compact LCA and also a VLCA.*

DEFINITION 5.10. *(Compact Answer of keyword search) Given a keyword query $\mathcal{K}$={k_1,k_2, $\cdots$,k_m} and an input XML document $\mathcal{D}$. $\mathcal{K}_{(\mathcal{D},(k_1,k_2,\cdots,k_m))}$ is the compact answer of $\mathcal{K}$ on $\mathcal{D}$, where $\mathcal{K}_{(\mathcal{D},(k_1,k_2,\cdots,k_m))}$ ={(r; $\lambda(v_1)$:c_1, $\lambda(v_2)$:c_2,$\cdots$, $\lambda(v_m)$:c_m) | r is the CVLCA of v_1, v_2,$\cdots$, v_m, where c_i denotes all the text content that v_i contains. }.*

Definition 5.9 presents a more compact and accurate concept to answer keyword queries, and Definition 5.10 demonstrates that the compact answer composed of compact connected trees should be more meaningful to answer keyword queries, where each compact connected tree describes how each result matches the keyword query. The idea behind the compact connected tree is, since node v is in a compact connected tree, it will not be in another looser one, which contain some other irrelevant nodes. Furthermore, CVLCA is different from and more meaningful than SLCA. For example, consider the keyword query {"XML", "Bob"} on the XML document in Figure 1(b), *paper*(5) and *paper*(12) are both CVLCAs as *paper*(5) dominates *title*(6) and *author*(7) while *paper*(12) dominates *title*(13) and *author*(14); but *paper*(5) does not dominate *title*(6) and *author*(14). Moreover, only compact nodes (e.g., *title*(13) and *author*(14)) share a CVLCA while the loose ones (e.g., *title*(6) and *author*(14)) cannot. However, *paper*(5) is not a valid SLCA since it has a descendant *paper*(12) which is also a LCA of this keyword query, and *paper*(5) will be absent from the answer of SLCA. Therefore, SLCA causes the false negative problem as they wrongly discard LCAs which have LCA descendants, and CVLCA can avoid this problem.

More importantly, CVLCA has a key property that we can efficiently compute CVLCAs through only one scan of the input content nodes. Observed from Definition 5.9, when detecting that all the current elements in each input list are larger than the current LCA, we can assure that those elements in each input list and the elements before the current LCA (i.e., the elements that have been popped out from input lists) will not constitute a compact connected tree together, and Lemma 5.5 guarantees the correctness.

Algorithm 2: VLCAStack Algorithm

```
Input: $\mathcal{K}$={$k_1$,$k_2$,$\cdots k_m$} and an XML document $\mathcal{D}$
Output: KwRst=$\mathcal{K}_{(\mathcal{D},(k_1,k_2,\cdots,k_m))}$= {($r$; $\lambda(v_1)$:$c_1$,
        $\lambda(v_2)$:$c_2$,$\cdots$,$\lambda(v_m)$:$c_m$),$\cdots$ }
1  begin
2      KwRst←$\phi$;
3      getNodeLists(); /*$\mathcal{I}_i$={$v_i|v_i$ directly contains keyword $k_i$}*/
4      while $\mathcal{S}_{VLCA}$ is not empty or $\mathcal{I}_i$ is not empty do
5          $min$=$minarg_i(\mathcal{I}_i$.first());
6          $n_{min}$=$\mathcal{I}_{min}$.first();
7          $\mathcal{I}_{min}$.pop_front();
8          $\mathcal{S}_{min}$.push($n_{min}$);
9          while vlca=$\mathcal{S}_{VLCA}$.top() is not an ancestor of $n_{min}$
           do
10             $\mathcal{S}_{VLCA}$.pop();
11             if vlca contains all the keywords then
12                 if vlca is a CVLCA then
13                     KwRst←(vlca;$\lambda(v_1)$:$c_1$,$\cdots$,$\lambda(v_m)$:$c_m$);
                       /*$v_i$ is the node associated with the pointers of
                       vlca, and $c_i$ is its corresponding text.*/
14                 pop(vlca.Pointers); /*pop elements from stacks*/
15             else
16                 addPointer($\mathcal{S}_{VLCA}$.top,vlca.Pointers);
17         $\mathcal{S}_{VLCA}$.push($n_{min}$);
18         addPointer($\mathcal{S}_{VLCA}$.top(),$\mathcal{S}_{min}$.top());
19 end
```

Figure 4: VLCAStack algorithm to answer keyword search

Subsequently, we introduce an effective optimization technique for computing CVLCAs, and inspired from this optimization, we devise a novel stack-based algorithm, VLCAStack.

LEMMA 5.5. *Given m nodes, $v_1 \in \mathcal{I}_1$, $v_2 \in \mathcal{I}_2$,$\cdots$,$v_m \in \mathcal{I}_m$, w= LCA(v_1,v_2,$\cdots v_m$). $\forall v_i' \in \mathcal{I}_i$, $v_i' < w$ or $v_i' > w$, there does not exist w', which dominates both v_i' and v_j ($j \neq i$).*

PROOF. Suppose node w' dominates v_j. As w=LCA(v_1,v_2,$\cdots$,v_m), $w' \succeq w$. If $v_i' > w$, then $w' < v_i'$; otherwise, if $v_i' < w$, then $w' > v_i'$, and thus w' can not dominate v_i'. According to Definition 5.9, there exists one and only one node, which dominates v_j. Hence, there is no w', which dominates both v_i' and v_j($j \neq i$). ☐

Optimization Technique: *Let r=LCA(v_1,v_2,$\cdots$,v_m). $\forall d$, $d \succeq r$, d will not share a common Compact LCA with any node n, $\mathcal{C}_n > \mathcal{C}_r$. Hence, when detecting a compact LCA, r, we can discard the descendants of r as they will not constitute other compact LCAs.*

Based on the above optimization technique, we can devise an effective stack-based algorithm. Stack-based algorithms require that the input elements $\mathcal{I}_i$ are sorted in order by their MDCs. In VLCAStack, MDCs of each inverted list are sorted in ascending order. Different from traditional stack-based algorithms, besides maintaining a stack $\mathcal{S}_i$ for each input list $\mathcal{I}_i$, VLCAStack still maintains another stack to preserve the current LCA, denoted as $\mathcal{S}_{VLCA}$. In addition, each node in $\mathcal{S}_{VLCA}$ is associated with some pointers to preserve the nodes that it contains in each current stack $\mathcal{S}_i$. If the node in $\mathcal{S}_{VLCA}$ contains all of the input keywords and is a VLCA, this node and the associated nodes with its pointers that contain input keywords will constitute a compact connected tree.

The basic idea of our algorithm is to merge the nodes in each $\mathcal{I}_i$ into compact connected trees rooted at their compact LCAs, and conceptually validate whether each compact LCA is a CVLCA. More importantly, all the nodes in each $\mathcal{I}_i$ will be scanned and pushed into its corresponding stack $\mathcal{S}_i$ and $\mathcal{S}_{VLCA}$ at most once.

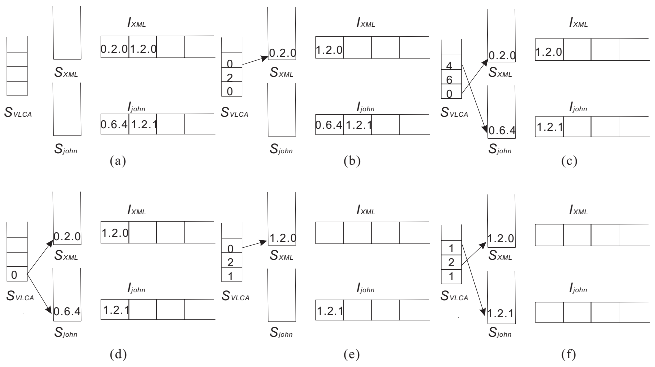


Figure 5: A running example of VLCAStack algorithm for the keyword query {"XML","John"} on the XML document in Figure 2(a).

In addition, once a node popped out from a stack, it will not contribute any answer in future. Accordingly, we demonstrate how to devise our algorithm based on the optimization technique.

Now, we introduce how to effectively identify all the compact connected tress through once scan of each $\mathcal{I}_i$. While $\mathcal{I}_i$ is not empty or $\mathcal{S}_{VLCA}$ is not empty, we select the node with minimal MDC among the first nodes of each current $\mathcal{I}_i$. Without loss of generality, we assume the minimal node is n_{min} and from $\mathcal{I}_{min}$. We pop n_{min} from $\mathcal{I}_{min}$ and push it into $\mathcal{S}_{min}$. If node vlca, the top element in $\mathcal{S}_{VLCA}$, is an ancestor of n_{min}, we push n_{min} into $\mathcal{S}_{VLCA}$ with a pointer to $\mathcal{S}_{min}$.top(); otherwise, for each element vlca in $\mathcal{S}_{VLCA}$, which is not an ancestor of n_{min}, if vlca contains all the keywords, it will be a compact LCA, and the nodes associated with its pointers and itself will constitute a compact connected tree. Subsequently, we need to check whether this compact LCA is a CVLCA. More importantly, if vlca is not an ancestor of n_{min}, n_{min}>vlca must hold, hence vlca and the nodes associated with its pointers can be popped out from corresponding stacks according to Lemma 5.5 and the proposed optimization technique; on the contrary, vlca does not contain all the input keywords and will be popped out from $\mathcal{S}_{VLCA}$, since its ancestor also contains the nodes associated with its pointers and may constitute a compact connected tree in future, and thus all the pointers associated with it will be transformed to its direct ancestor (i.e., the element directly below it in $\mathcal{S}_{VLCA}$). We repeat these steps until both each $\mathcal{I}_i$ and $\mathcal{S}_{VLCA}$ are empty, and Lemma 5.5 and the proposed optimization technique guarantee the correctness. Accordingly, we can devise our algorithm, VLCAStack, as shown in Figure 4.

VLCAStack first retrieves the input lists of the keywords (line 3), and then gets the node n_{min}, which has minimal MDC among the first nodes of each $\mathcal{I}_i$ (lines 5-6). n_{min} will be popped out from $\mathcal{I}_{min}$ and pushed into its corresponding stack $\mathcal{S}_{min}$ (lines 7-8). While the top element of $\mathcal{S}_{VLCA}$, vlca, is not an ancestor of n_{min}, VLCAStack pops vlca from $\mathcal{S}_{VLCA}$ (line 10). If vlca contains all the keywords and is a CVLCA, this vlca with the nodes associated with is pointers must constitute a compact connected tree and be added into the result set, KwRst (line 13), then VLCAStack pops them from corresponding stacks (line 14); otherwise, transforms the pointers of vlca to its ancestor, i.e., the current top node of $\mathcal{S}_{VLCA}$ (line 16). Finally, VLCAStack pushes n_{min} with its pointers into $\mathcal{S}_{VLCA}$ (lines 17-18).

To further digest the algorithm for readers, we walk through our algorithm with a running example as shown in Example 5.4 and Figure 5.

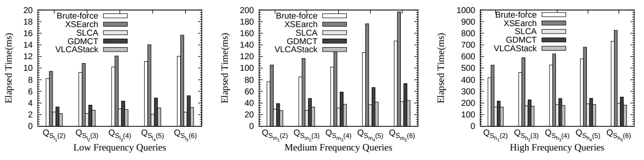


Figure 6: Evaluation of Elapsed Time on SIGMOD Record

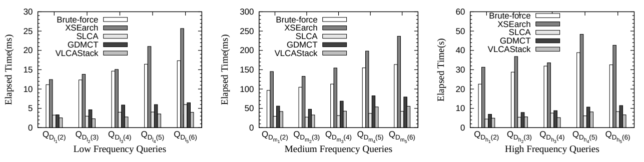


Figure 7: Evaluation of Elapsed Time on DBLP

EXAMPLE 5.4. *Consider the keyword query {"XML", "John"} on the XML document in Figure 2(a).* `VLCAStack` *first retrieves the input keyword lists, $\mathcal{I}_{XML}$={0.2.0;1.2.0}, $\mathcal{I}_{John}$={0.6.4;1.2.1} through the inverted index, and then computes the VLCAs and generates the compact connected trees rooted at these VLCAs as follows. In the first step, as $\mathcal{S}_{VLCA}$ is empty and n_{min}=0.2.0, whose MDC is minimal among the first nodes of each $\mathcal{I}_i$, we push it into $\mathcal{S}_{VLCA}$ with a pointer to the top element of $\mathcal{S}_{XML}$ (Figure 5(b)). In step 2, as n_{min} is 0.6.4 and the top element of $\mathcal{S}_{VLCA}$,* `vlca`*=0.2.0, is not an ancestor of 0.6.4, so* `VLCAStack` *pops* `vlca` *from $\mathcal{S}_{VLCA}$ and transforms its pointer to its ancestor node 0.2. In the same way,* `VLCAStack` *pops 0.2 and transforms the pointer of 0.2 to its ancestor 0, and then pushes node 0.6.4 with its pointer into $\mathcal{S}_{VLCA}$ (Figure 5(c)). In step 3, as n_{min} is 1.2.0 and* `vlca`*=0.6.4 is not an ancestor of 1.2.0, so we pop 0.6.4, 0.6 from $\mathcal{S}_{VLCA}$ and transform their pointers to their ancestors. As node* `conf(0)` *contains all the keywords (node 0.2.0 and node 0.6.4 are both associated with its pointers, i.e., they are descendants of* `conf(0)`*), we check whether* `conf(0)` *is a VLCA, and then pop it form $\mathcal{S}_{VLCA}$ and so do the two nodes, 0.2.0 and 0.6.4 from $\mathcal{S}_{XML}$ and $\mathcal{S}_{John}$ respectively (Figure 5(d)). Since node* `conf(0)` *is not a VLCA as discussed in Example 3.2, it will not be added into the result and skipped. In addition,* `VLCAStack` *pushes node 1.2.0 with its pointer into $\mathcal{S}_{VLCA}$ (Figure 5(e)). Accordingly, we can proceed to walk through the algorithm as shown in Figure 5. Finally, we get the answer of the keyword query, {(*`paper`*(1.2);*`title`*:XML(1.2.0),*`author`*:John(1.2.1))}.*

We analyze the complexity of `VLCAStack`. According to Lemma 5.5, there are at most $|\mathcal{I}_{minSize}|$ compact LCAs, where $\mathcal{I}_{minSize}$ is the input list that has minimal size among each $\mathcal{I}_i$. However, the number of the compacted connected trees may be larger than $|\mathcal{I}_{minSize}|$, and `VLCAStack` will identify all the compact connected trees as the answers. Therefore, `VLCAStack` needs to scan each $\mathcal{I}_i$ once and for each element, v_i in $\mathcal{I}_i$, pops the nodes which is not its ancestor from $\mathcal{S}_{VLCA}$ and pushes v_i into $\mathcal{S}_{VLCA}$, and then for each compact connected tree which contains all the input keywords, checks whether the root of this tree is a VLCA according to Lemma 3.4. The complexity of the former is O($d\sum_{i=1}^{m}|\mathcal{I}_i|$), while that of the latter is O($mdlog(md)|CCTrees|$), where m is the number of keywords involved in a keyword query, d is the depth of the XML document and $|CCTrees|$ is the number of compact connected trees. Thus, the total complexity of `VLCAStack` is O($d\sum_{i=1}^{m}|\mathcal{I}_i|$+ $mdlog(md)$*$|CCTrees|$). $|CCTrees|$ is usually proportional to $|\mathcal{I}_{minSize}|$, and is much less than $\prod_{i=1}^{m}|\mathcal{I}_i|$. Therefore, this further demonstrates that `VLCAStack` outperforms our brute-force algorithm based on exhaustive enumeration.

6. EXPERIMENTAL STUDY

We have designed and performed a comprehensive set of experiments to evaluate the performance of our proposed algorithms. We used both real and synthetic datasets. The synthetic dataset was generated using the XMark benchmark [4] with a factor of 1.0 and the raw file was about 115MB. We also used the real dataset DBLP [1] and SIGMOD Record, TreeBank datasets from Washington XML Data Repository [3] to explore the performance of our algorithms. The sizes of DBLP, SIGMOD Record and TreeBank were respectively about 350MB, 500KB and 82MB.

The experiments were conducted on an Intel(R) Pentium(R) 2.4GHz computer with 512MB of RAM running Windows XP Professional. The algorithms were implemented in Java and the parsing of the XML files was performed using the SAX API of the Xerces Java Parser. We compared our approach with the state-of-art proposals, XSEarch [14], SLCA [38][§], and GDMCT [21].

We employed four metrics, elapsed time, precision, recall and $\mathcal{F}$-*measure* to evaluate the efficiency and effectiveness of these algorithms. To compute precision and recall, we manually reformulated the keyword queries into schema-aware XQuery queries according to the schemas of datasets and took the results of these corresponding transformed queries as a baseline, and then computed precision and recall of given queries according to the baseline as follows. Given a keyword query $\mathcal{K}$ and its corresponding transformed XQuery $\mathcal{X}$, the accurate result set of $\mathcal{K}$, i.e., the result of $\mathcal{X}$, is denoted as `AR`, and the approximate result set, i.e., the result of a specified algorithm on $\mathcal{K}$, is denoted as `PR`. Accordingly, we can defined the precision and recall of this algorithm as follows. Precision of the specified algorithm is the ratio between |`AR`∩`PR`| and |`PR`|, while Recall is the ration between |`AR`∩`PR`| and |`AR`|. For $\mathcal{F}$-*measure*, $\mathcal{F}=\frac{2*\mathcal{P}*\mathcal{R}}{\mathcal{P}+\mathcal{R}}$, where $\mathcal{F}$,$\mathcal{P}$ and $\mathcal{R}$ are $\mathcal{F}$-*measure*, precision and recall of the specified algorithm respectively.

To better understand the performance of our algorithms for various keyword queries with different selectivities, we performed our experiments using various sets of keywords with different frequencies, namely, low, medium and high, respectively corresponding to keywords with frequency between 1-50, 51-500, and above 500.

6.1 Efficiency

We evaluated the efficiency of `VLCAStack`, our brute-force algorithm, XSEarch, SLCA and GDMCT on SIGMOD Record, DBLP, XMark and TreeBank datasets respectively in this section, and compared their elapsed time on various queries.

For each dataset, we selected fifteen keyword queries. Each one of the first five queries has at least one keyword with low frequency, each of the medium five queries has no keywords with low frequency and has at least one keyword with medium frequency, and for the last five queries, all the keywords of each query have high frequencies. In addition, every keyword query contains 2-6 keywords, for example, in Figure 6, $Q_{S_{l_2}}(3)$ means that $Q_{S_{l_2}}$ is a selected query on SIGMOD Record dataset, which contains 3 keywords and has at least one keyword with low frequency. Figure 6, 7, 8 and 9 describe the experiment results on SIGMOD Record, DBLP and XMark and TreeBank datasets respectively.

Besides inverted indices associated with keywords, XSEarch still has to maintain an all-pairs interconnection index, which is very expensive to compute whatever online or offline. The brute-force algorithm is also inefficient when some input keywords have high frequencies in that it has to exhaustively enumerate all the combinations of content nodes in each $\mathcal{I}_i$. Therefore, `VLCAStack` outperforms XSEarch and the brute-force algorithm. Especially, on

[§] In this paper, we adopted Indexed Lookup Eager algorithm of SLCA for comparison as it achieves better performance.

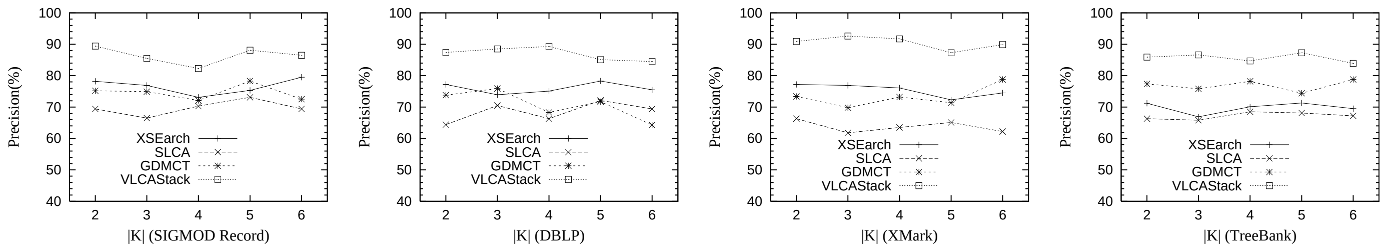


Figure 10: Comparison on Precision

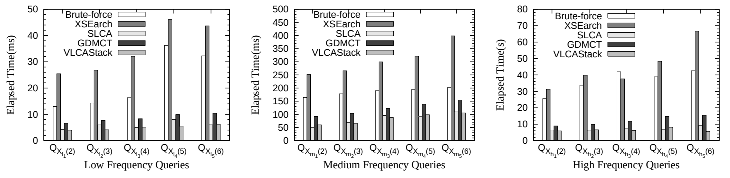


Figure 8: Evaluation of Elapsed Time on XMark

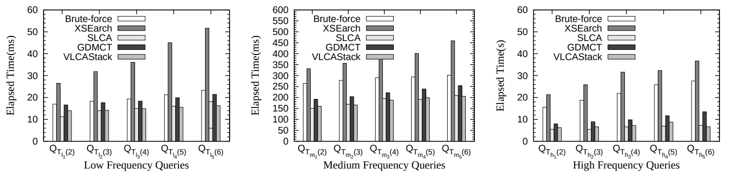


Figure 9: Evaluation of Elapsed Time on TreeBank

$Q_{S_{h_5}}$, VLCAStack costs less than 200ms, while the brute-force algorithm costs 700ms and XSEarch costs more than 800ms as shown in Figure 6. While, on $Q_{X_{h_5}}$, the speedup [¶] of VLCAStack over the brute-force algorithm and XSEarch are 8 and 12 respectively as illustrated in Figure 8.

VLCAStack is also superior to GDMCT, and the reason is that, GDMCT employs the region-based code, which is not as efficient as MDC to compute LCAs of various nodes. Especially, on $Q_{D_{m_4}}$, VLCAStack only costs 50ms, while GDMCT costs 80ms as shown in Figure 7. In addition, we can see SLCA is more efficient than the other ones when a keyword has a low frequency, e.g., $Q_{S_{l_5}}(6)$, $Q_{S_{l_2}}(3)$ and $Q_{T_{l_1}}(2)$. However when the frequencies of all keywords have no distinct differences, VLCAStack is as good as SLCA and even better than it. For example, on $Q_{S_{h_3}}(4)$, $Q_{D_{l_5}}(6)$ and $Q_{X_{l_4}}(5)$, the speedup of VLCAStack against SLCAs are about 1.2, 1.4 and 1.5 respectively. Furthermore, SLCA is efficient in that it only retrieves SLCASet instead of the compact connected tree as our method, so that it leads to low effectiveness when compared with other methods, which will be further discussed in Section 6.2.

6.2 Effectiveness

This section evaluates the effectiveness of those algorithms based on three good metrics borrowed from IR literature, precision, recall and $\mathcal{F}$-$measure$, and reports the experimental result. We selected six keyword queries for each dataset and performed the five algorithms on them, and computed precision, recall and $\mathcal{F}$-$measure$ of the five algorithms on the selected keyword queries.

SLCA will cause false positive and false negative problems, while XSEarch, GDMCT and the brute-force algorithm will cause the false positive problem as discussed in above sections. VLCAStack achieves higher precision than the other approaches, and the brute-force algorithm has the same precision as XSEarch, which in turn outperforms GDMCT and SLCA. In addition, VLCAStack is as good as the brute-force algorithm, XSEarch, and GDMCT in terms of completeness, and they achieve higher recall than SLCA, as SLCA will miss results from the answer in the case that their descendants are also LCAs. The experimental results are illustrated in Figure 10, Table 2 and Table 3.

[¶]The speedup of algorithm X over algorithm Y is the ratio between T_Y and T_X, where T_Y and T_X are the elapsed time of Y and X respectively.

Precision. We can see, in Figure 10, VLCAStack outperforms the other three algorithms in term of precision (we omit the brute-force algorithm since it has the same precision as XSEarch) on whatever datasets. More importantly, the precision of VLCAStack is nearly 90% on all the selected datasets, and thus it works well in practice. Although XSEarch also achieves high precision, it is inefficient since it is very expensive to compute an all-pairs interconnection index. For example, on Q_{T_5} (the keyword query with 5 keywords on TreeBank), the precision of VLCAStack nearly reaches 90%, while those of XSEarch, GDMCT and SLCA are only 73%, 70% and 68% respectively as shown in Figure 10. This comparison further reflects the effectiveness of our method. In addition, VLCAStack returns compact connected trees as answers, which are more compact and meaningful than the answers of SLCA, GDMCT and XSEarch.

Recall. Based on the discussion in above sections, VLCAStack, XSEarch, GDMCT and the brute-force algorithm should achieve higher recall than SLCA as only SLCA causes the false negative problem. However, there is no difference between the former four algorithms, hence we here mainly compare VLCAStack with SLCA as shown in Table 2. Since only if there are some nested labels/tags in the XML documents, SLCA involves false negatives, thus we compared them on TreeBank and our synthetic dataset generated according to the DTD in Figure 2(b) using IBM XML Generator. We can see VLCAStack achieves higher recall on all the keyword queries, and is superior to SLCA about 20 percent. This comparison reflects that our method outperforms SLCA significantly.

$\mathcal{F}$-$measure$. To further compare those algorithms, we employed another good metric $\mathcal{F}$-$measure$. We selected six queries for each dataset and compared the average of their $\mathcal{F}$-$measure$. We can see VLCAStack beats the other algorithms and achieves the best $\mathcal{F}$-$measure$ as shown in Table 3. For example, on TreeBank, $\mathcal{F}$-$measure$ of VLCAStack reaches 90.1%, while those of the other ones are less than 70%, and especially that of SLCA is only 62.1%.

In summary, in terms of efficiency, VLCAStack is always better than GDMCT, which in turn is superior to the brute-force algorithm and XSEarch on whatever datasets and keyword queries. When the input keywords have no distinct frequencies, VLCAStack is as good as SLCA and even better than SLCA. On the other hand, VLCAStack outperforms the brute-force algorithm and XSEarch, which in turn are better than GDMCT and SLCA in terms of effectiveness. Hence, VLCAStack achieves both higher efficiency and effectiveness when compared with other methods.

Table 2: Comparison on Recall

Recall(%)	The generated dataset				TreeBank			
	Q_{G_1}	Q_{G_2}	Q_{G_3}	Q_{G_4}	Q_{T_1}	Q_{T_2}	Q_{T_3}	Q_{T_4}
SLCA	74.4	71.5	75.3	66.1	75.4	70.2	73.1	66.5
`VLCAStack`	100	100	100	100	96.4	93.1	95.4	92.3

Table 3: Comparison on $\mathcal{F}$-$measure$

$\mathcal{F}$-$measure$(%)	SLCA	XSEarch	Brute-force	GDMCT	`VLCAStack`
SIGMOD Record	78.1	82.3	82.3	82.1	97.1
DBLP	77.4	81.5	81.5	81.2	96.3
XMark	76.1	79.2	79.2	77.6	93.9
TreeBank	62.1	68.5	68.5	66.5	90.1
Generated dataset	69.8	79.9	79.9	78.2	100

7. CONCLUSION

In this paper, we have investigated the problem of keyword search over XML documents, with the aim of identifying the most meaningful content elements that contain all the input keywords, along with a compact connected tree to describe how each result matches a given keyword query.

To obtain more meaningful results of keyword queries, we propose the notions of Valuable LCA and Compact VLCA to accurately and efficiently answer XML keyword queries. Based on the two concepts, we propose the compact connected trees rooted CVLCAs as the answers of keyword queries. Moreover, we present an optimization technique for accelerating the computation of CVLCAs and devise an efficient stack-based algorithm to identify the meaningful compact connected trees.

We have implemented the proposed method and the extensive experiment results showed that our method achieved high efficiency and effectiveness on both synthetic and real datasets.

Acknowledgement

This work is partly supported by the National Natural Science Foundation of China under Grant No. 60573094, the National High Technology Development 863 Program of China under Grant No. 2007AA01Z152, the National Grand Fundamental Research 973 Program of China under Grant No. 2006CB303103, Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology (TNList), and Tsinghua Basic Research Foundation under Grant No. JCqn2005022.

8. REFERENCES

[1] http://dblp.uni-trier.de/xml/.

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

[3] http://www.cs.washington.edu/research/xmldatasets/.

[4] http://www.xml-benchmark.org/.

[5] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

[6] S. Al-Khalifa, H. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE*, pages 141–152, 2002.

[7] S. Amer-Yahia, E. Curtmola, and A. Deutsch. Flexible and efficient xml search with complex full-text predicates. In *SIGMOD*, pages 575–586, 2006.

[8] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman. Structure and content scoring for xml. In *VLDB*, pages 361–372, 2005.

[9] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[10] C. Botev, S. Amer-Yahia, and J. Shanmugasundaram. Expressiveness and performance of full-text search languages. In *EDBT*, pages 349–367, 2006.

[11] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins:optimal xml pattern matching. In *SIGMOD*, pages 310–321, 2002.

[12] S. Chen, H. Li, et al. Twig2stack: Bottom-up processing of generalized-tree-pattern queries over xml documents. In *VLDB*, 2006.

[13] S. Cohen, Y. Kanza, B. Kimelfeld, and Y. Sagiv. Interconnection semantics for keyword search in xml. In *CIKM*, pages 389–396, 2005.

[14] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. Xsearch: A semantic search engine for xml. In *VLDB*, pages 45–56, 2003.

[15] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, 2007.

[16] L. Guo, J. Shanmugasundaram, and G. Yona. Topology search over biological databases. In *ICDE*, 2007.

[17] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *SIGMOD*, pages 16–27, 2003.

[18] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. In *SIAM J. Comput. 13(2)*, pages 338–355, 1984.

[19] H. He, H. Wang, J. Yang, and P. Yu. Blinks : Ranked keyword searches on graphs. In *SIDMOD*, 2007.

[20] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[21] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword proximity search in xml trees. In *IEEE Trans. Knowl. Data Eng. 18(4)*, pages 525–539, 2006.

[22] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[23] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on xml graphs. In *ICDE*, pages 367–378, 2003.

[24] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[25] Y. Li, H. Yang, and H. V. Jagadish. Nalix: an interactive natural language interface for querying xml. In *SIGMOD*, pages 900–902, 2005.

[26] Y. Li, H. Yang, and H. V. Jagadish. Constructing a generic natural language interface for an xml database. In *EDBT*, pages 737–754, 2006.

[27] Y. Li, C. Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, pages 72–84, 2004.

[28] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, pages 563–574, 2006.

[29] Z. Liu and Y. Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD*, 2007.

[30] J. Lu, T. W. Ling, C.-Y. Chan, and T. Chen. From region encoding to extended dewey: On efficient processing of xml twig pattern matching. In *VLDB*, pages 193–204, 2005.

[31] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: Top-k keyword query in relational databases. In *SIDMOD*, 2007.

[32] A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava. Adaptive processing of top-k queries in xml. In *ICDE*, pages 162–173, 2005.

[33] A. Markowetz, Y. Yang, and D. Papadias. Keyword search on relational data streams. In *SIDMOD*, 2007.

[34] S. Pradhan. An algebraic query model for effective and efficient retrieval of xml fragments. In *VLDB*, pages 295–306, 2006.

[35] M. Sayyadian, H. LeKhac, A. Doan, and L. Gravano. Efficient keyword search across heterogeneous relational databases. In *ICDE*, 2007.

[36] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. In *SIAM J. Comput. 17(6)*, pages 1253–1262, 1988.

[37] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD*, pages 204–215, 2002.

[38] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD*, pages 527–538, 2005.