Efficient LCA based Keyword Search in XML Data

Yu Xu
Teradata
yu.xu@teradata.com

Yannis Papakonstantinou
University of California, San Diego
yannis@cs.ucsd.edu

ABSTRACT

Keyword search in XML documents based on the notion of lowest common ancestors (*LCAs*) and modifications of it has recently gained research interest [10, 14, 20]. In this paper we propose an efficient algorithm called Indexed Stack to find answers to keyword queries based on XRank's semantics to LCA [10]. The complexity of the Indexed Stack algorithm is $O(kd|S_1|\log|S|)$ where k is the number of keywords in the query, d is the depth of the tree and $|S_1|$ ($|S|$) is the occurrence of the least (most) frequent keyword in the query. In comparison, the best worst case complexity of the core algorithms in [10] is $O(kd|S|)$. We analytically and experimentally evaluate the Indexed Stack algorithm and the two core algorithms in [10]. The results show that the Indexed Stack algorithm outperforms in terms of both CPU and I/O costs other algorithms by orders of magnitude when the query contains at least one low frequency keyword along with high frequency keywords.

1. INTRODUCTION

Keyword search in XML documents based on the notion of lowest common ancestors in the labeled trees modeled after the XML documents has recently gained research interest in the database community [10, 14, 20]. One important feature of keyword search is that it enables users to search information without having to know a complex query language or prior knowledge about the structure of the underlying data. Consider a keyword query Q consisting of k keywords $w_1, \ldots, w_k$. According to the LCA-based query semantics proposed in [10], named *Exclusive Lowest Common Ancestors* (*ELCA*) in the sequel, the result of the keyword query Q is the set of nodes that contain at least one occurrence of all of the query keywords either in their labels or in the labels of their descendant nodes, after *excluding* the occurrences of the keywords in the subtrees that already contain at least one occurrence of all the query keywords. For example, the answers to the keyword query "XML David" on the data in Figure 1 is the node list [0, 0.2, 0.2.2, 0.3, 0.3.2, 0.3.3, 0.3.4, 0.4.2]. The answers show that "David" is an author of five papers that have "XML" in the titles (rooted at 0.2.2, 0.3.2, 0.3.3, 0.3.4 and 0.4.2); and that "David" is the chair of two sessions that have "XML" in the titles (rooted at 0.2 and 0.3), and the chair of the conference (rooted at 0) whose name contains "XML". Notice that the node session with id 0.4 is not an $ELCA$ answer since the only "XML" instance (node 0.4.2.1.1) under 0.4 is under one of its children (0.4.2) which already contains keyword instances of both "XML" and "David". Therefore under the *exclusion* requirement in the $ELCA$ definition, the session node 0.4 is not an $ELCA$ answer. The node Conference rooted at 0 is an $ELCA$ answer since it contains the node 0.1.1 and the node 0.5.1 which are not under any child of the node 0 that contains instances of both keywords "XML" and "David".

We propose an efficient algorithm called Indexed Stack to answer keyword queries according to the $ELCA$ query semantics proposed in XRank [10] with complexity of $O(kd|S_1|\log|S|)$ where k is the number of keywords in the query, d is the depth of the tree, $|S_1|$ ($|S|$) is the occurrence of the least (most) frequent keyword in the query. In comparison, the complexity of the core algorithms in [10] is $O(kd|S|)$ and $O(k^2d|S|p\log|S| + k^2d|S|^2)$ respectively where p is the maximum number of children of any node in the tree. The algorithm in [10] with complexity $O(k^2d|S|p\log|S| + k^2d|S|^2)$ is tuned to return only the top m answers for certain queries where it may terminate faster than other algorithms. In particular, our contributions include:

- We propose an efficient algorithm, named Indexed Stack (IS) for keyword search in XML documents according to the $ELCA$ semantics proposed in XRank [10]. Our analysis of the algorithm shows that the complexity of the proposed algorithm is $O(kd|S_1|\log|S|)$.
- Our experiments evaluate the Indexed Stack algorithm, and the algorithms in [10] and show that the Indexed Stack algorithm outperforms in terms of both CPU and I/O costs other algorithms by orders of magnitude when the query contains at least one low frequency keyword along with high frequency keywords.

In Section 2 we provide the $ELCA$ query semantics and definitions used in the paper. Section 3 describes related work, with focus on LCA-based keyword search in XML documents based on the notation of lowest common ancestors [10, 14, 20]. Section 4 presents the Indexed Stack algorithm, and also provides the complexity analysis of the Indexed Stack algorithm and the algorithms in [10] for both main memory and disk accesses. Our experimental results comparing the Indexed Stack algorithm and the two core algorithms in [10] appear in Section 5. We conclude in Section 6.

2. ELCA QUERY SEMANTICS

We model XML documents as trees using the conventional labeled ordered tree model. Each node v of the tree corresponds to an XML element and is labeled with a tag $\lambda(v)$.

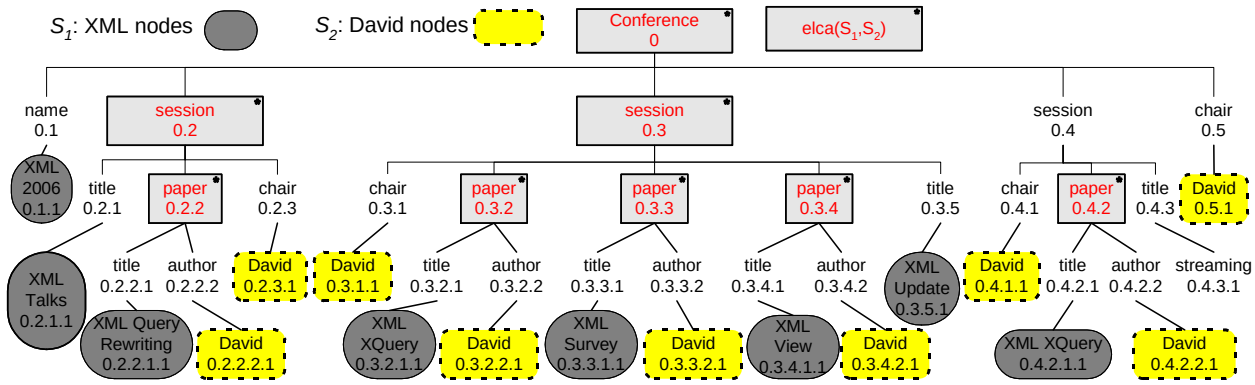


Figure 1: Example XML document

The notation $v \prec_a v'$ denotes that node v is an ancestor of node v'; $v \preceq_a v'$ denotes that $v \prec_a v'$ or $v = v'$.

We first introduce the *Lowest Common Ancestor* (*LCA*) of k nodes (sets) before we formally define the $ELCA$ query semantics.

The function $lca(v_1, \ldots, v_k)$ computes the *Lowest Common Ancestor* (*LCA*) of nodes $v_1, \ldots, v_k$. The LCA of sets $S_1, \ldots, S_k$ is the set of LCA's for each combination of nodes in S_1 through S_k.

$$lca(S_1, ..., S_k) = \{lca(n_1, \ldots, n_k) | n_1 \in S_1, \ldots, n_k \in S_k\}$$

For example, in Figure 1, $lca(S_1, S_2)$=[0, 0.2, 0.2.2, 0.3, 0.3.2, 0.3.3, 0.3.4, 0.4, 0.4.2].

A node v is called an *LCA of sets* $S_1, \ldots, S_k$ if $v \in lca(S_1, \ldots, S_k)$.

A node v is called an *Exclusive Lowest Common Ancestor (ELCA) of* $S_1, \ldots, S_k$ if and only if there exist nodes $n_1 \in S_1, \ldots, n_k \in S_k$ such that $v = lca(n_1, ..., n_k)$ and for every n_i $(1 \leq i \leq k)$ the child of v in the path from v to n_i is not an LCA of $S_1, \ldots, S_k$ itself nor ancestor of any LCA of $S_1, \ldots, S_k$.

According to the $ELCA$ query semantics proposed in XRank [10], the query result of a keyword query Q consisting of k keywords $w_1, \ldots, w_k$ is defined to be

$$elca(w_1, \ldots, w_k) = elca(S_1, \ldots, S_k)$$

where $elca(S_1, \ldots, S_k) = \{v \mid \exists n_1 \in S_1, \ldots, n_k \in S_k (v = lca(n_1, ..., n_k) \wedge \quad \forall i (1 \leq i \leq k) \nexists x (x \in lca(S_1, \ldots, S_k) \wedge child(v, n_i) \preceq_a x))\}$ where S_i denotes the *inverted list* of w_i, i.e., the list of nodes sorted by id whose label directly contains w_i and $child(v, n_i)$ is the child of v in the path from v to n_i. The node n_i is called an *ELCA witness node* of v in S_i. Note that a node v is an $ELCA$ of $S_1, \ldots, S_k$ if and only if $v \in elca(S_1, \ldots, S_k)$.

Notice that the above definition is based on LCAs and is expressed differently than but it is equivalent to [10]. In Figure 1 $elca$("XML", "David")= $elca(S_1, S_2)$=[0, 0.2, 0.2.2, 0.3, 0.3.2, 0.3.3, 0.3.4, 0.4.2]. The node 0.1.1 is an $ELCA$ witness node of the node 0 in S_1 and the node 0.5.1 is an $ELCA$ witness node of the node 0 in S_2.

The *Smallest Lowest Common Ancestor (SLCA) of k sets* $S_1, \ldots, S_k$ is defined to be

$$slca(S_1, \ldots, S_k) = \{v | v \in lca(S_1, \ldots, S_k) \wedge \forall v' \in lca(S_1, \ldots, S_n) \ v \not\prec v'\}$$

A node v is called a *Smallest Lowest Common Ancestor (SLCA) of* $S_1, \ldots, S_k$ if $v \in slca(S_1, \ldots, S_k)$. Note that a node in $slca(S_1, \ldots, S_n)$ cannot be an ancestor node of any other node in $slca(S_1, \ldots, S_n)$.

In Figure 1, $slca(S_1, S_2)$=[0.2.2, 0.3.2, 0.3.3, 0.3.4, 0.4.2]. Clearly $slca(S_1, \ldots, S_k) \subseteq elca(S_1, \ldots, S_k) \subseteq lca(S_1, \ldots, S_k)$. For example, consider S_1 and S_2 in Figure 1. The node 0.2 is not in $slca(S_1, S_2)$ but in $elca(S_1, S_2)$ and the node 0.4 is not in $elca(S_1, S_2)$ but in $lca(S_1, S_2)$.

Similarly to [10, 20], each node is assigned a Dewey id $pre(v)$ that is compatible with preorder numbering, in the sense that if a node v_1 precedes a node v_2 in the preorder left-to-right depth-first traversal of the tree then $pre(v_1) < pre(v_2)$. Dewey numbers provide a straightforward solution to locating the LCA of two nodes. The usual $<$ relationship holds between any two Dewey numbers. Given two nodes v_1, v_2 and their Dewey numbers p_1, p_2, $lca(v_1, v_2)$ is the node with the Dewey number that is the longest common prefix of p_1 and p_2. The cost of computing $lca(v_1, v_2)$ is $O(d)$ where d is the depth of the tree. For example, in Figure 1 $lca(0.2.2.1.1, 0.2.2.2.1)$=0.2.2.

3. RELATED WORK

Extensive research has been done on keyword search in both relational and graph databases [9, 1, 11, 12, 3, 13]. There are works on keyword search on XML databases modeled as trees [10, 14, 20, 4, 17]. This work falls in this category. Finally [15, 6, 7, 17, 18, 19, 14, 2] integrate keyword search into XML query languages.

We focus on the three most closely related works: XRank ([10]), Schema-Free XQuery ([14]) and XKSearch ([20]), all of which base keyword search in XML on the notation of lowest common ancestors of the nodes containing keywords.

XRank ([10]) defines the answer to a keyword search query Q "$w_1, \ldots, w_k$" to be $elca(S_1, \ldots, S_k)$ where S_i is the inverted list of w_i $(1 \leq i \leq k)$. It also extends PageRank's ranking mechanism to XML by taking the nested structure of XML into account. Each node in the tree is assigned a precomputed ranking score which is independent of any keyword query. The ranking score of an answer node (i.e., an $ELCA$ node) v to the query Q is computed by XRank's aggregate ranking function which takes into account individual scores of the witness nodes of v and the distance between the witness nodes and the answer nodes— the contribution of a witness node x's ranking to the node v decays by the distance between v and x. [10] proposes two core algorithms, *DIL* (Dewey Inverted List) and *RDIL* (Ranked Dewey Inverted List), to return the top m answers from $elca(S_1, \ldots, S_k)$. Notice that the ranking functions and the search algorithms (DIL and RDIL) are independent of each other, in the sense that the same search algorithms could apply to other ranking functions [1].

[1]as long as the aggregate ranking functions are monotone with re-

The DIL algorithm in [10] keeps an inverted list sorted by Dewey id for each keyword. DIL (conceptually) sort merges the k inverted lists of the k query keywords and reads each node v in the sorted merged list in order. Intuitively it is easy to verify the correctness of the DIL algorithm since it reads all nodes in the k inverted lists in document order and has enough information to determine whether a lowest common ancestor of k nodes from the k inverted list is an $ELCA$ node or not. Notice that the DIL algorithm has to scan to the end of all inverted lists. The complexity of the DIL algorithm is $O(kd|S|)$ where $|S|$ is the size of the largest inverted list among $S_1, \ldots, S_k$ and d is the depth of the tree.

The RDIL algorithm in [10] maintains two separate data structures: inverted lists sorted by the individual nodes' ranking score in descending order and B+ trees built on inverted lists sorted by Dewey id in ascending order. The underlying assumption of RDIL is that higher ranked results ($ELCA$ nodes) are likely to come from nodes in the front of inverted lists sorted by ranking score in descending order and query processing may terminate without scanning to the end of all of the inverted lists. RDIL works as follows:

1. it reads a node v from the k inverted lists sorted by rank, in round-robin fashion [2].
2. then it uses the B+trees built on inverted lists sorted by Dewey id to find the lowest common ancestor l that contains v and all other keywords. The key observation is that given a node v, an inverted list S sorted by document order and the B+ tree BT built on S, it takes only a single range scan ([8]) in BT to find the node v' in S whose id is the least that is greater than the id of v such that either v' or its immediate predecessor in S shares the longest common prefix with v which is the Dewey id of l.
3. however the node l produced in the second step may not be an $ELCA$ node. RDIL first determines whether each child of l contains all keywords or not ($O(kdp \log |S|)$ where p is the maximum number of children of any node in the tree). Then for each keyword w_i, RDIL checks that keyword witness nodes of l are not under any of its children that contain all keyword instances. The complexity of RDIL is $O(k^2 d|S|p \log |S| + k^2 d|S|^2)$.

Given a node v in an inverted list, as can be seen from the above explanation, the RDIL algorithm does not completely scan other inverted lists in order to find an LCA node that contains v and all other keywords. However, in order to guarantee correctness (not losing any answer nodes and not returning non-answer nodes), scan is repeatedly performed and that is why the complexity of the RDIL is high in the worst case. Furthermore, it is not guaranteed that individual nodes with higher ranking scores always lead to answer nodes with higher overall ranking score because the combination ranking function takes into account the distance between witness nodes and answer nodes. Moreover, given a keyword query, there is no practical way to determine a priori whether the DIL or the RDIL algorithm will perform better. The experiments in [10] have demonstrated that the performance of RDIL can be significantly worse than that of DIL for returning the top m query answers. [10] proposes a hybrid algorithm which starts using RDIL and switches to DIL when it finds out that RDIL has spent too much time on answering the query.

XKSearch ([20]) defines the answers to a keyword query Q of "$w_1, \ldots, w_k$" to be $slca(S_1, \ldots, S_k)$ where S_i is the inverted list of the keyword w_i. The complexity of the Indexed Lookup Eager algorithm in [20] is $O(kd|S_1| \log |S|)$ and hence can be orders of magnitude better than the Stack based algorithm adopted from [10] or [14] when a query contains keywords of orders of magnitude of different frequencies. [20] also extends the algorithm computing $slca(S_1, \ldots, S_k)$ to compute all LCAs of k sets (i.e., $lca(S_1, \ldots, S_k)$). The intuition is that we can first compute all $SLCA$ nodes of $S_1, \ldots, S_k$. Then we visit every node u in the path from every $SLCA$ node to the root and determine whether u is a LCA node or not. The complexity of the algorithm in [20] based on the above intuition to compute all LCAs is $O(kd^2|S_1| \log |S|)$. We may attempt to compute $elca(S_1, \ldots, S_k)$ similarly. That is, in order to compute $elca(S_1, \ldots, S_k)$, we could do the following: (1) first compute $slca(S_1, \ldots, S_k)$ using the Indexed Lookup Eager algorithm in [20] whose complexity is $O(kd|S_1| \log |S|)$. (2) then for each $SLCA$ node v computed in the first step, we walk up from v to the root and determine whether each ancestor node l of v is an $ELCA$ node. However the difficulty is then that we have to perform the same expensive operations we described in the third step of the RDIL algorithm in [10] a few paragraphs before. Therefore the complexity of such an algorithm would be $O(k^2 d|S|p \log |S| + k^2 d|S|^2)$ where p is the maximum number of children of any node in the tree.

Schema-Free XQuery ([14]) uses the idea of *Meaningful LCA* (MLCA), similar to *SLCA*, and proposes a stack based sort merge algorithm which scans to the end of all inverted lists. The complexity of the algorithm in [14] is the same as that of DIL ($O(kd|S|)$). [14] shows that keyword search functionality can be easily integrated into the structured query language XQuery as built-in functions, enabling users to query XML documents based on partial knowledge they may have over underlying data with different and potentially evolving structures. The recall and precision experiments in [14] shows that it is possible to express a wide variety of queries in a schema-free manner and have them return correct results over a broad diversity of schemas. The demonstrated integration of $MLCA$ based keyword search functionality into XQuery can also apply to the $ELCA$ query semantics.

In this paper we will only focus on the algorithmic aspects of the problem of efficiently finding answers to keyword queries in XML documents, and we will not attempt a comparison of the quality of different query semantics.

Intuitively answering a keyword query according to the $ELCA$ query semantics is more computationally challenging than according to the $SLCA$ query semantics. In the latter the moment we know a node l has a child c which contains all keywords, we can immediately determine that the node l is not a $SLCA$ node. However we cannot determine that l is not an $ELCA$ node because l may contain keyword instances that are not under c and are not under any node that contains all keywords. Notice that given the same query, the size of the answers of the $SLCA$ semantics cannot be more than that of the $ELCA$ semantics because $slca(S_1, \ldots, S_k) \subseteq elca(S_1, \ldots, S_k)$.

In this paper, we propose an efficient algorithm, Indexed Stack algorithm (IS), which takes advantage of the benefits of both stack based algorithms and indexed lookup based algorithms. The complexity is $O(kd|S_1| \log |S|)$.

4. INDEXED STACK ALGORITHM (IS)

This section presents the Indexed Stack (IS) algorithm that computes $elca(S_1, \ldots, S_k)$. We choose S_1 to be the smallest among $S_1, \ldots, S_k$ since $elca(S_1, \ldots, S_k) = elca(S_{j_1}, \ldots, S_{j_k})$, where $j_1, .., j_k$ is any permutation of $1, 2, \ldots, k$, and there is a benefit in using the smallest list as S_1 as we will see in the complexity

spect to individual keyword ranks (See Section 2.3 in [10] for more details).

[2] e.g., it reads a node from each inverted list in turn.

analysis of the algorithm. We assume $|S|$ denotes the size of the largest inverted list. The Indexed Stack algorithm, leveraging key tree properties described in this section, starts from the smallest list S_1, visits each node in S_1, but does not need to access every node in other lists. It achieves high efficiency, especially when the smallest list is significantly smaller than the largest list.

The algorithm's efficiency is based on first discovering the nodes of a set $elca\_can(S_1; S_2, \ldots, S_k)$ (short for $ELCA$ Candidates) defined in Section 4.1, which is a superset of $elca(S_1, \ldots, S_k)$ but can be computed efficiently in $O(kd|S_1| \log |S|)$, as shown in Section 4.2. Section 4.3 describes an efficient function $isELCA()$ that determines whether a given node of $elca\_can(S_1; S_2, \ldots, S_k)$ is a member of $elca(S_1, \ldots, S_k)$. Section 4.4 presents a stack-based algorithm that puts together the computation of $elcan\_can$ and $isELCA$, avoiding redundant computations. Section 4.4 also presents the complexity analysis of the algorithm.

4.1 The ELCA candidate set *elca_can()*

We define next the set $elca\_can(S_1; S_2, \ldots, S_k)$, whose members are called $ELCA\_CAN$ nodes (of S_1 among S_2, ..., S_k).

$$elca\_can(S_1; S_2, \ldots, S_k) = \bigcup_{v_1 \in S_1} slca(\{v_1\}, S_2, \ldots, S_k)$$

Note that a node v is an $ELCA\_CAN$ node iff there exist $n_1 \in S_1, \ldots, n_k \in S_k$ such that $v = lca(n_1, \ldots, n_k)$ and there must not exist $n'_2 \in S_2, \ldots, n'_k \in S_k$ such that $v' = lca(n_1, n'_2, \ldots, n'_k)$ and $v \prec_a v'$. Every n_i ($1 \leq i \leq k$) is called an $ELCA\_CAN$ *witness node* of v in S_i.

For example, in Figure 1 $elca\_can(S_1; S_2)$=[0, 0.2, 0.2.2, 0.3, 0.3.2, 0.3.3, 0.3.4, 0.4.2]. Next, consider the $ELCA\_CAN$ node 0.2. The nodes 0.2.1.1 and 0.2.2.2.1 are its witness nodes in S_1 and S_2 respectively. However the node 0.2.2.1.1 is not a witness node for 0.2 in S_1. This is because although the node 0.2 is the LCA of the node 0.2.2.1.1 from S_1 and the node 0.2.3.1 from S_2, there exists the node 0.2.2.2.1 from S_2 such that the LCA of 0.2.2.1.1 and 0.2.2.2.1 (i.e., 0.2.2) is a descendant of 0.2.

Note that $elca\_can(S_1; S_2, \ldots, S_k)$ may contain nodes that are ancestors of other nodes of $elca\_can(S_1; S_2, \ldots, S_k)$. The following inclusion relationship between $elca$ and $elca\_can$ applies.

PROPERTY 1.

$\forall i \in [1, \ldots, k]$,
$elca(S_1, \ldots, S_k) \subseteq elca\_can(S_i; S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_k)$.

PROOF. If $v \in elca(S_1, \ldots, S_k)$, there must exist $ELCA$ witness nodes $n_1 \in S_1, \ldots, n_k \in S_k$ such that $v = lca(n_1, \ldots, n_k)$ and there must not exist $n'_1 \in S_1, \ldots, n'_{i-1} \in S_{i-1}, n'_{i+1} \in S_{i+1}, \ldots, n'_k \in S_k$ such that $v' = lca(n'_1, \ldots, n'_{i-1}, n_i, n'_{i+1}, \ldots, n'_k)$ and $v \prec_a v'$ (Otherwise n_i cannot be an $ELCA$ witness node of v). Thus $v \in elca\_can(S_i; S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_k)$ by definition. □

Of particular importance is the instantiation of the above property for $i = 1$ (i.e., $elca(S_1, \ldots, S_k) \subseteq elca\_can(S_1; S_2, \ldots, S_k)$) since $elca\_can(S_1; S_2, \ldots, S_k)$ has the most efficient computation (recall S_1 is the shortest inverted list).

In Figure 1, $elca(S_1, S_2)$ and $elca\_can(S_1; S_2)$ happen to be the same. However if we remove the node 0.3.1.1 from the tree of Figure 1, then $elca\_can(S_1; S_2)$ stays the same but the node 0.3 would not be in $elca(S_1, S_2)$ anymore. Therefore, it would be $elca(S_1, S_2) \subset elca\_can(S_1; S_2)$.

For presentation brevity, we define $elca\_can(v)$ for $v \in S_1$ to be the node l where $\{l\}$=$elca\_can(\{v\}; S_2, \ldots, S_k)$= $slca(\{v\}, S_2, \ldots, S_k)$. The node $elca\_can(v)$ is called the *exclusive lowest common ancestor candidate or* $ELCA\_CAN$ of v (in sets of $S_2, \ldots, S_k$). Note that each node in $lca(\{v\}, S_2, \ldots, S_k)$ is either an ancestor node of v or v itself and $elca\_can(v)$ is the lowest among all nodes in $lca(\{v\}, S_2, \ldots, S_k)$. For instance, consider S_1 and S_2 in Figure 1. $elca\_can(0.1.1) = 0$, $elca\_can(0.2.1.1) = 0.2$, $elca\_can(0.2.2.1.1) = 0.2.2$, $elca\_can(0.3.2.1.1) = 0.3.2$, $elca\_can(0.3.3.1.1) = 0.3.3$, $elca\_can(0.3.4.1.1) = 0.3.4$, $elca\_can(0.3.5.1) = 0.3$ and $elca\_can(0.4.2.1.1) = 0.4.2$.

4.2 Computing elca_can(v)

In this section we describe how prior work ([20]) can be extended to efficiently compute $elca\_can(v)$ in the interest of completeness and clarity.

Let us assume that we want to compute $slca(\{v\}, S_2)$ where $S_2 = \{u_1, \ldots, u_n\}$. The key observation in [20] is that the witness node in S_2 for $slca(v_1, S_2)$ must be one of the two closest nodes (in document order) to v among all nodes in the set S_2. We can efficiently find the only two nodes of $\{u_1, \ldots, u_n\}$ that are candidates for witnessing the $SLCA$, by using two important functions: the function $rm(v, S)$ computes the *right match* of v in a set S, that is the node of S that has the smallest id that is greater than or equal to $pre(v)$; $lm(v, S)$ computes the *left match* of v in a set S, that is the node of S that has the biggest id that is less than or equal to $pre(v)$. The function $rm(v, S)$ ($lm(v, S)$) returns null when there is no right (left) match node. For example, consider again S_1 and S_2 in Figure 1 and the node v = 0.3.2.1.1 from S_1. The right match for v in S_2 is the node 0.3.2.2.1, and the left match for v in S_2 is the node 0.3.1.1. Consequently $slca(\{v\}, S_2)$ is the lower node from $lca(v, rm(v, S_2))$ and $lca(v, lm(v, S_2))$. Consider again S_1, S_2, and v = 0.3.2.1.1 from S_1 in Figure 1, $elca\_can(0.3.2.1.1)$=0.3.2. This is because $lca(v, rm(v, S_2))$=$lca(v, 0.3.2.2.1)$=0.3.2, $lca(v, lm(v, S_2))$=$lca(v, 0.3.1.1)$=0.3, and $0.3 \prec_a 0.3.2$.

The cost of computing $rm(v, S)$ ($lm(v, S)$) is $O(d \log |S|)$ since it takes $O(\log |S|)$ steps (each step being a Dewey number comparison) to find the right (left) match node and the cost of comparing the Dewey ids of two nodes is $O(d)$.

The key point in [20] applies to the computation of $slca(\{v\}, S_2, \ldots, S_k)$. The node $elca\_can(v)$ (i.e., $slca(\{v\}; S_2, \ldots, S_k)$) can be efficiently computed as follows: First we compute the (unique) $SLCA$ v_2 of v and of the nodes of S_2. It continues by iteratively computing the (unique) $SLCA$ v_i of v_{i-1} and S_i, until i becomes k. The node v_k is the result.

Notice though that the nodes of $elca\_can(S_1; S_2, \ldots, S_k)$ may be obtained out of order by applying the above computation on each node in S_1. For example in Figure 1, $elca\_can(0.3.2.1.1) = 0.3.2$ and $elca\_can(0.3.5.1) = 0.3$. Thus the $ELCA\_CAN$ node 0.3 is computed after the $ELCA\_CAN$ node 0.3.2. The time complexity of computing $elca\_can(v)$ is $O(kd \log |S|)$.

4.3 Determine whether an $ELCA\_CAN$ node is an $ELCA$ node

This section presents the function $isELCA$ which is used to determine whether an $ELCA\_CAN$ node v is an $ELCA$ node or not. Let $child\_elcacan(v)$ be the set of children of v that contain all keyword instances. Equivalently $child\_elcacan(v)$ is the set of child nodes u of v such that either u or one of u's descendant nodes is an $ELCA\_CAN$ node, i.e.,

$$child\_elcacan(v) = \{u | u \in child(v) \ \wedge \ \exists x \ (u \preceq_a x \ \wedge \ x \in elca\_can(S_1; S_2, \ldots, S_k))\}$$

where $child(v)$ is the set of child nodes of v. We use $ELCA\_CAN$

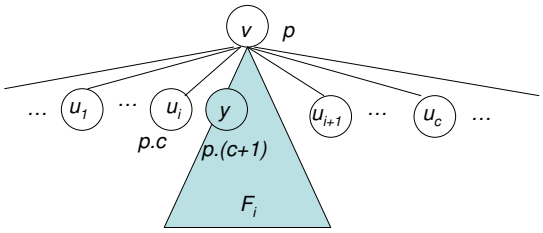


Figure 2: v and its $ELCA\_CAN$ children

in the above definition of $child\_elcacan(v)$ because we can efficiently compute $elca\_can(S_1; S_2, \ldots, S_k)$ as discussed in Section 4.2. For S_1 and S_2 of the running example in Figure 1, $child\_elcacan(0)$=[0.2, 0.3, 0.4] and $child\_elcacan(0.2)$=[0.2.2].

Assume $child\_elcacan(v)$ is $\{u_1, \ldots, u_c\}$ (See Figure 2). By definition, an $ELCA$ node v must have $ELCA$ witness nodes $n_1, \ldots, n_k$ such that $n_1 \in S_1, \ldots, n_k \in S_k$ and every n_i is not in the subtrees rooted at the nodes from $child\_elcacan(v)$.

To determine whether v is an $ELCA$ node, we probe every S_i to see if there is a node $x_i \in S_i$ such that x_i is either in the forest under v to the left of the path vu_1, or in the forest under v to the right of the path vu_c, or in any forest F_i that is under v and between the paths vu_i and vu_{i+1}, $i = 1, \ldots, c-1$. The last case can be checked efficiently by finding the right match $rm(y, S_i)$ of the node y in S_i where y is the immediate right sibling of u_i among the children of v. Assume $pre(v) = p$, $pre(u_i) = p.c$ where c is a single number, then $pre(y) = p.(c+1)$, as shown in Figure 2. Let the right match of y in S_i be x (i.e., $x = rm(y, S_i)$). Then x is a witness node in the forest F_i if and only if $pre(x) < pre(u_{i+1})$.

Given the list ch which is the list of nodes in $child\_elcacan(v)$ sorted by id, the function $isELCA(v, ch)$ (Figure 3) returns true if v is an $ELCA$ node by applying the operations described in the previous paragraph. As an example, consider the query "XML David" and the inverted lists S_1 and S_2 in Figure 1. $child\_elcacan(0)$= $[0.2, 0.3, 0.4]$. We will see how $isELCA(0, [0.2, 0.3, 0.4])$ works and returns true. In this example, the number of keywords is two ($k = 2$) and $|ch|$=3. First the function $isELCA$ searches and finds the existence of an $ELCA$ witness node (i.e., the node $0.1.1$) for 0.2 in S_1 in the subtree rooted under 0 to the left of the path from 0 to 0.2 (0.2 is the first child $ELCA\_CAN$ node of 0). Then the function searches the existences of an $ELCA$ witness node in S_2 for 0 in the forest to the left of the path from 0 to 0.2; in the forest between the path from 0 to 0.2 and the path from 0 to 0.3; in the forest between the path from 0 to 0.3 and the path from 0 to 0.4; in the forest to the right of the path from 0 to 0.4. All of the above searches fail except that the last search successfully finds a witness node ($0.5.1$) for 0.2 in S_2. Therefore, $isELCA(0, [0.2, 0.3, 0.4])$ returns true.

The time complexity of $isELCA(v, ch)$ is $O(kd \log |S||child\_elcacan(v)|)$ (line 1, 3 and 4).

4.4 Indexed Stack Algorithm

In Section 4.1 we stated that $elca\_can(S_1; S_2, \ldots, S_k)$ is a superset of $elca(S_1, \ldots, S_k)$. Section 4.2 described how to efficiently compute $elca\_can(S_1; S_2, \ldots, S_k)$ and Section 4.3 described how to efficiently check whether an $ELCA\_CAN$ node in $elca\_can(S_1; S_2, \ldots, S_k)$ is an $ELCA$ node, when the list of child nodes of v that contain all keyword instances are given. Therefore, the only missing part of efficient computation of $elca(S_1, \ldots, S_k)$ is how to compute $child\_elcacan(v)$ for each $ELCA\_CAN$ node v. Since we can easily compute $child\_elcacan(v)$ if we know

```
isELCA(v, ch){
(* return true if v is an ELCA node. ch=child_elcacan(v) *)

1   for 1 ≤ i ≤ k {
2     x=v
3     for 1 ≤ j ≤ |ch| {
4       x = rm(x, S_i) (* x is a witness node for v in S_i*)
5       if( pre(x) < pre(ch[j]) ) break;
6       else {
// The function sibling(u) returns the immediate right
//sibling node of u among the list of child nodes of v.
7           x=sibling(ch[j])
8       }
9     }
10    if (j==|ch| + 1) {
11      x = rm(sibling(ch[|ch|]), S_i)
12      if(v ⊀_a x) return false;
13  }
14  return true;
```

Figure 3: Determine whether an $ELCA\_CAN$ node is an $ELCA$ node

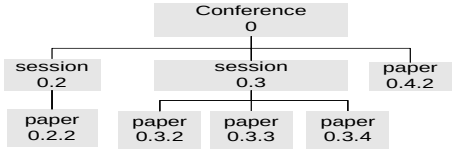


Figure 4: The tree structure of all $ELCA\_CAN$ nodes in Figure 1

every $ELCA\_CAN$ node x_i under v [3], we can just compute all $ELCA\_CAN$ nodes and then compute $child\_elcacan(v)$ for each $ELCA\_CAN$ node v.

A straightforward approach would compute all $ELCA\_CAN$ nodes and store them in a tree which keeps only the original ancestor-descendant relationships of all $ELCA\_CAN$ nodes in the input document. As an example, such a tree describing all $ELCA\_CAN$ nodes in Figure 1 is shown in Figure 4. Note that though $0.4.2$ is a descendant of 0 in Figure 1, it is a child of 0 in Figure 4.

A straightforward algorithm to compute $elca(S_1, \ldots, S_k)$ works as follows where TS is a tree structure initialized to empty:

1. For each node v in S_1, compute $l = elca\_can(v)$ based on Section 4.2 and do $TS.insert(l)$ which inserts l to the appropriate place in TS based on l's ancestor-descendant relationship with nodes already inserted in TS. The tree in Figure 4 shows the result from this step for computing $elca(S_1, S_2)$ in Figure 1.

2. For each node l in TS check whether l is an $ELCA\_CAN$ node or not by calling $isELCA(l, child\_elcacan(l))$ where $child\_elcacan(l)$ can be easily computed from the list of child nodes of l in TS.

However the above approach has the following disadvantages:

- the complexity of the approach is $O(d|S_1|^2 + |S_1|kd \log |S|)$ where the $O(d|S_1|^2)$ component comes from the cost of creating and maintaining the tree structure;

- and all ($O(|S_1|)$) $ELCA\_CAN$ nodes have to be computed first and kept in memory before we can start to recognize any $ELCA$ nodes.

[3] $child\_elcacan(v)$ is the set of child nodes u_i of v on the paths from v to x_i, which can be efficiently computed with Dewey numbers.

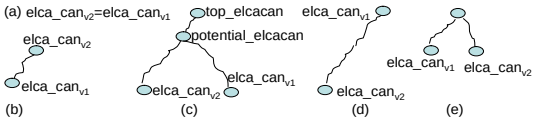


Figure 5: Relationships between any two nodes l and s

Instead, a "one pass" stack-based algorithm, whose complexity is $O(|S_1|kd\log|S|)$, is presented in Figure 6. The Indexed Stack algorithm does not have to keep all $ELCA\_CAN$ nodes in memory; it uses a stack whose depth is bounded by the depth of the tree based on some key tree properties. At any time during the computation any node in the stack is a child or descendant node of the node below it (if present) in the stack. Therefore the nodes from the top to the bottom of the stack at any time are from a single path in the input tree.

We will first present the Indexed Stack algorithm, illustrated with a running example, then discuss at the end of this section optimization techniques in the implementation of the algorithm.

4.4.1 Algorithm Description

We go through every node v_1 in S_1 in order, compute $elca\_can_{v_1} = elca\_can(v_1)$ and create a stack entry $stackEntry$ consisting of $elca\_can_{v_1}$. If the stack is empty, we simply push $stackEntry$ to the stack to determine later whether $elca\_can_{v_1}$ is an $ELCA$ node or not. If the stack is not empty, what the algorithm does depends on the relationship between $stackEntry$ and the top entry in the stack. The algorithm either discards $stackEntry$ or pushes $stackEntry$ to the stack (with or without first popping out some stack entries). The algorithm does not need to look at any other non top entry in the stack at any time and only determines whether an $ELCA\_CAN$ node is an $ELCA$ node at the time when a stack entry is popped out.

Each stack entry $stackEntry$ created for a node v_1 in S_1 has the following three components.

- $stackEntry.elca\_can$ is $elca\_can(v_1)$;
- $stackEntry.CH$ records the list of child or descendant $ELCA\_CAN$ nodes of $stackEntry.elca\_can$ seen so far, which will be used by $isELCA()$ to determine whether $stackEntry.elca\_can$ is an $ELCA$ node at the time when this entry is popped out from the stack;
- and $stackEntry.SIB$ (short for siblings) is the list of $ELCA\_CAN$ nodes before $stackEntry.elca\_can$ (in document order) such that the LCA node of nodes from the list and $stackEntry.elca\_can$ potentially can be an $ELCA\_CAN$ node that has not been seen so far.

Let us illustrate the need for and role of $stackEntry.SIB$ with the running example "XML David". Before we compute $elca\_can(0.3.5.1)$=0.3, we have already computed 0.3.2, 0.3.3, 0.3.4 as $ELCA\_CAN$ nodes which are the child $ELCA\_CAN$ nodes of 0.3. We have to store these three $ELCA\_CAN$ nodes in order to determine whether 0.3 is an $ELCA$ node or not before we see 0.3 in the processing, which is achieved by first storing 0.3.2 in the SIB component of the stack entry associated with 0.3.3 and then storing 0.3.2 and 0.3.3 in the SIB component of the stack entry associated with 0.3.4 (after the stack entry associated with 0.3.3 is popped out) during the processing before we see 0.3. Note that if the node 0.3.1.1 was not in the tree in Figure 1, we would still see 0.3 in the processing as an $ELCA\_CAN$ node and still see 0.3 after 0.3.2, 0.3.3, and 0.3.4, but then 0.3 would not be an $ELCA$ node, which could be determined only if we have kept the information that 0.3.2, 0.3.3 and 0.3.4 are $ELCA\_CAN$ nodes until we see 0.3 and know that 0.3 would not have any child or descendant $ELCA\_CAN$ nodes in the processing later after we see 0.3. It is possible that we would not see 0.3 at all in the processing (i.e., if the node 0.3.5.1 was not in the tree, 0.3 would be not be an $ELCA\_CAN$ node) in which case we still need to keep 0.3.2, 0.3.3 and 0.3.4 until the point we are sure that those nodes cannot be child or descendant of any other $ELCA\_CAN$ nodes.

Figure 6, which presents the Indexed Stack pseudo-code, and Figure 7, which has snapshots of the stack during operation of the algorithm, also include an entry $stackEntry.witNodes$, which we temporarily ignore, as it is used only in the optimization version of the algorithm, described at the end of this section.

For each node v_1 in S_1 (line 2), the Indexed Stack algorithm computes $elca\_can_{v_1} = elca\_can(v_1)$ as discussed in Section 4.2 (line 4). We create a stack entry $stackEntry$ consisting of $elca\_can_{v_1}$ (line 6). If the stack is empty (line 7), we simply push $stackEntry$ to the Stack to determine later whether $elca\_can_{v_1}$ is an $ELCA$ node or not. If the stack is not empty, let the node at the top of the stack be $elca\_can_{v_2}$ (line 9-10). Figure 5 shows the only five relationships the two $ELCA\_CAN$ nodes $elca\_can_{v_2}$ and $elca\_can_{v_1}$ (in fact any two nodes) can have.

- In the first case where $elca\_can_{v_1}$ and $elca\_can_{v_2}$ are the same (Figure 5(a), line 11), $elca\_can_{v_1}$ is discarded.
- In the second case where $elca\_can_{v_2}$ is an ancestor of $elca\_can_{v_1}$ (Figure 5(b)), we push $stackEntry$ to the stack to determine later whether $elca\_can_{v_1}$ is an $ELCA$ node or not (line 12).
- In the third case (Figure 5(c)) where $elca\_can_{v_2}$ and $elca\_can_{v_1}$ have no ancestor-descendant relationship and $elca\_can_{v_1}$ appears after $elca\_can_{v_2}$ in document order (line 13), we pop the top stack entry repeatedly (line 14) until either the stack is empty or the $ELCA\_CAN$ node of the top entry in the Stack (named $top\_elcacan$ in Figure 5(c), line 15) is an ancestor of $elca\_can_{v_1}$ by calling the function $popStack()$. When a stack entry is popped out, the $ELCA\_CAN$ node in the stack entry is checked whether it is an $ELCA$ node or not (by $isELCA()$). Note that there will not be any $ELCA\_CAN$ node in later processing that can be a child or descendant node of any popped out $ELCA\_CAN$ node. That is why we can pop out those entries and check for $ELCA$ nodes. Let $popEntry$ be the last popped out entry and $potential\_elcacan$ be the LCA of $popEntry.elca\_can$ and $elca\_can_{v_1}$ (Figure 5(c), line 16). If the stack is not empty and the top stack entry's node $top\_elcacan$ is an ancestor of $potential\_elcacan$ (Figure 5(c), line 17), then we set the SIB list associated with $elca\_can_{v_1}$ to be the concatenation of the SIB list in $popEntry$ and $popEntry.elca\_can$ (line 18). We then push $stackEntry$ to the stack (line 19). The reason that we need to carry on the nodes stored in the SIB component of $popEntry$ to $stackEntry$ was explained a few paragraphs before in the example illustrating the need for and role of the SIB component in a stack entry. During the processing of the example, at one point $elca\_can_{v_2}$ is 0.3.2, $elca\_can_{v_1}$ is 0.3.3, $potential\_elcacan$ is 0.3, $top\_elcacan$ is 0, and after the stack entry for 0.3.2 is popped out, 0.3.2 is stored in the SIB component of the stack entry for 0.3.3. Notice that $potential\_elcacan$ could be a node that we have not seen so far in the processing (i.e., it has not been computed as an $ELCA\_CAN$ node) and it could be an $ELCA\_CAN$ and an $ELCA$ node. Although we have guessed

its existence here, it may or may not appear later in the processing. That is why we need to carry $elca\_can_{v_2}$ and nodes in the SIB component of $elca\_can_{v_2}$ to the SIB component of $elca\_can_{v_1}$ for $potential\_elcacan$.

- In the fourth case where $elca\_can_{v_1} \prec_a elca\_can_{v_2}$ (line 21, Figure 5(d)), it is certain that $elca\_can_{v_2}$ has no more descendant $ELCA\_CAN$ nodes. Thus we pop from the stack repeatedly until either the stack is empty or the $ELCA\_CAN$ node in the top entry is an ancestor of $elca\_can_{v_1}$ (line 22). Again, the $ELCA\_CAN$ node in each popped out entry is checked whether it is an $ELCA$ node or not. Let the last popped out entry be $popEntry$ (line 22). We copy the SIB list in $popEntry$ and $popEntry.elca\_can$ to the CH field of $elca\_can_{v_1}$ (line 23). Then $stackEntry$ is pushed to the top of the stack (line 24). Notice that nodes stored in the SIB field by the processing in the third case are used in the fourth case to set the CH field.

- The fifth case, where $elca\_can_{v_1}$ and $elca\_can_{v_2}$ have no ancestor-descendant relationship and $elca\_can_{v_1}$ appears before $elca\_can_{v_2}$, is not possible in the computation when S_1 is sorted in document order.

Now we discuss the details of the function $popStack(elca\_can_{v_1})$ (called in the processing of the third and fourth cases in Figure 5). It repeatedly pops out the top entry (line 31) until the $ELCA\_CAN$ node in the top entry is an ancestor of $elca\_can_{v_1}$ or the stack becomes empty. Each popped out entry is checked on whether it contains an $ELCA$ node or not by calling the function $isELCA$ presented in Section 4.3 (line 33). Notice that the function $toChildELCA\_CAN(v, L)$ inputs a node v and a list L each node of which is a child or descendant $ELCA\_CAN$ node of v and returns $child\_elcacan(v)$. Each popped out node is added to the top entry's CH field (line 36) because at any time any $ELCA\_CAN$ node in a stack entry is a child or descendant node of the $ELCA\_CAN$ node in the stack entry below it (if present).

The time complexity of the Indexed Stack algorithm is $O(|S_1|kd \log |S|)$ where k is the number of keywords in the query, d is the depth of the tree and $|S_1|$ ($|S|$) is the occurrence of the least (most) frequent keyword in the query. The time complexity comes from two primitive operations: $elca\_can()$ and $isELCA()$. The total cost of calling $elca\_can(v)$ is $O(kd|S_1| \log |S|)$ as discussed in Section 4.2. The cost of calling the function $isELCA(v, CH)$ once is $O(|CH|.kd \log |S|)$ or $|child\_elcacan(v)|kd \log |S|$ (see Figure 2). The accumulated total cost of calling $isELCA$ is $O(\sum_{v \in elca\_can(S_1;S_2,\ldots,S_k)} |child\_elcacan(v)|kd \log |S|)$. Let $Z = \sum_{v \in elca\_can(S_1;S_2,\ldots,S_k)} |child\_elcacan(v)|$. Note that $|elca\_can(S_1; S_2, \ldots, S_k)| \leq |S_1|$ and $|child\_elcacan(v)| \leq |S_1|$. Each node in $elca\_can(S_1; S_2, \ldots, S_k)$ increases the value of Z by at most one (see Figure 4). Thus $O(\sum_{v \in elca\_can(S_1;S_2,\ldots,S_k)} |child\_elcacan(v)|) = O(|S_1|)$. Therefore the time complexity of the Indexed Stack algorithm is $O(|S_1|kd \log |S|)$.

The number of disk access needed by the Indexed Stack algorithm is $O(k|S_1|)$ because for each node in S_1 the Indexed Stack algorithm just needs to find the left and match nodes in each one of the other $k-1$ keyword lists. Note that the number of disk accesses of the Indexed Stack algorithm cannot be more than the total number of blocks of all keyword lists on disk because the algorithm accesses all keyword lists strictly in order and there is no repeated scan on any keyword list. Since B+ tree implementations usually buffer non-leaf nodes in memory, we assume the number of disk accesses of a random search in a keyword search is $O(1)$ as in [10, 20]. The complexity analysis of the Indexed Stack, the two algorithms in [10], DIL and RDIL are summarized in Table 1 for both main memory and disk accesses for finding all query answers and only top m query answers where $|S_1|$($|S|$) is the occurrence of the least (most) frequent keyword in the query, B is the total number of blocks of all inverted lists on disk, d is the maximum depth of the tree and p is the maximum number of children of any node in the tree.

```
1  Stack = empty
2  for each node v1 in S1 {
3  (* elca_can_{v1} :the ELCA_CAN of v1; *)
4     elca_can_{v1} = elca_can(v1)
5        (* create a Stack entry stackEntry for elca_can_{v1} *)
6     stackEntry = [elca_can = elca_can_{v1}; SIB = []; CH = []]
7     if (Stack.isEmpty()) Stack.push(stackEntry)
8     else {
9        topEntry = Stack.top()
10       elca_can_{v2} = topEntry.elca_can
11       if(pre(elca_can_{v2}) == pre(elca_can_{v1})) {} (*  Figure 5(a)  *)
12       else if (elca_can_{v2} ≺_a elca_can_{v1})
            Stack.push(stackEntry)(* Figure 5(b) *)
13       else if (pre(elca_can_{v2}) < pre(elca_can_{v1})){ (* Figure 5(c)  *)
14          popEntry = popStack(elca_can_{v1})
15          top_elcacan = Stack.top().elca_can
16          potential_elcacan = lca(elca_can_{v1}, popEntry.elca_can)
17          if(!Stack.isEmpty() && top_elcacan ≺_a potential_elcacan)
18                stackEntry.SIB= [popEntry.SIB,popEntry.elca_can]
19          Stack.push(stackEntry)
20       }
21       else if (elca_can_{v1} ≺_a elca_can_{v2}) { (* Figure 5(d)  *)
22          popEntry = popStack(elca_can_{v1})
23          stackEntry.CH=[popEntry.SIB, popEntry.elca_can]
24          Stack. push(stackEntry)
25       }
26    }
27 } (* end of for loop *)
28 popStack(0) (* clean up the stack *)

29 popStack(elca_can_{v1}) : StackEntry
(* pop out all top entries of the stack whose nodes are not ancestors of elca_can_{v1} *)
30 popEntry=null;
31 while(Stack.top() !=NULL
          && Stack.top().elca_can ⊀ elca_can_{v1} ) {
32    popEntry=Stack.pop()
33    if(isELCA(popEntry.elca_can,
34       toChildELCA_CAN(popEntry.elca_can, popEntry.CH))
35       output popEntry.elca_can as an ELCA
36    Stack.top().CH += popEntry.elca_can
37 }
38 return popEntry;
```

Figure 6: The Indexed Stack Algorithm

	number of disk accesses	main memory complexity
Indexed Stack	$O(k\|S_1\|)$	$O(kd\|S_1\| \log \|S\|)$
DIL	$O(B)$	$O(kd\|S\|)$
RDIL	$O(k^2d\|S\|p \log \|S\| + k^2d\|S\|^2)$	$O(k^2d\|S\|p \log \|S\| + k^2d\|S\|^2)$

Table 1: Main memory and Disk Complexity Analysis of Indexed Stack, DIL and RDIL

4.4.2 Running Example

We illustrate the algorithm using the query "XML David" on the data of Figure 1. Figure 7 shows the states of the stack after the processing of each node in S_1 for computing $elca(S_1; S_2)$. The caption under each figure describes which node v_1 in S_1 has just been processed, the id of the node $elca\_can_{v_1} = elca\_can(v_1)$, which of the four cases in Figure 5 has happened, and the pop/push actions that happened.

Figures 7(a), 7(b), and 7(c) show the processing of the first three S_1 nodes, 0.1.1, 0.2.1.1 and 0.2.2.1.1. The case of Figure 5(b) is

applied.

Figure 7(d) shows the processing of the node 0.3.2.1.1. The case of Figure 5(c) is applied. The two nodes 0.2.2 and 0.2 are popped out from the stack and determined to be $ELCA$ nodes; the CH field associated with the node 0 is updated with the addition of the node 0.2; and $elca\_can(0.3.2.1.1)$=0.3.2 is pushed onto the stack.

Figure 7(e) shows the result of processing 0.3.3.1.1 from S_1. Note that $elca\_can_{v_1} = 0.3.3$. The processing for the case of Figure 5(c) is applied. The node 0.3.2 is popped out and reported as an $ELCA$. Also 0.3.2 is stored in the SIB field of the entry associated with 0.3.3. Figure 7(f) shows the processing of the node 0.3.4.1.1 from S_1 which is similar to the processing shown in Figure 7(e). The node 0.3.3 is popped out and reported as an $ELCA$, and added to the SIB field of the stack entry associated with 0.3.4. Note that the $ELCA\_CAN$ node 0.3 has not been seen yet.

The processing for the node 0.3 shown in Figure 7(g) is interesting in that it picks up the nodes previously stored in SIB and uses it to update the CH field of the stack entry associated with 0.3. Without this action, we cannot determine whether the node 0.3 is an $ELCA$ or not because some of its child $ELCA\_CAN$ nodes (0.3.2, 0.3.3 and 0.3.4) have been seen and they have to been stored. The node 0.3.4 is popped out and determined to be an $ELCA$ node.

Figure 7(h) shows the processing of the last node 0.4.2.1.1 from S_1 which is similar to the processing shown in Figure 7(d). The node 0.3 is popped out and determined to be an $ELCA$ node. The node 0.4.2 is pushed onto the stack. At this stage every node in S_1 has been processed. Figure 7(i) shows that after cleaning up the stack, the stack becomes empty and nodes 0.4.2 and 0 are determined to be $ELCA$ nodes.

4.4.3 Algorithm Optimization

To emphasize the key ideas behind the Indexed Stack algorithm and for presentation simplicity, we did not present some optimization techniques in the implementation of the algorithm shown in Figure 6.

Incremental $isELCA()$**.** Notice that we can do without storing the child or descendant $ELCA\_CAN$ nodes of an $ELCA\_CAN$ node in the stack. That is, we can remove the CH field in the structure of a stack entry. The above can be achieved by the following two changes: i) extending the computation of $elca\_can(v)$ along with an array of $ELCA\_CAN$ witness nodes of $elca\_can(v)$; ii) changing the function $isELCA$'s signature accordingly to $isELCA(l, WN)$ where l is an $ELCA\_CAN$ node and WN is the list of l's $ELCA\_CAN$ witness nodes. The idea is that some of the $ELCA\_CAN$ witness nodes of $elca\_can(v)$ kept along the way of computing $elca\_can(v)$ may be $ELCA$ witness node for $elca\_can(v)$. If an $ELCA\_CAN$ witness node x is also an $ELCA$ witness node for $elca\_can(v)$ in a set S_i, then there is no need in $isELCA()$ to search for $ELCA$ witness nodes for $elca\_can(v)$ in S_i. For example in the stack state shown in Figure 7(h), the child $ELCA\_CAN$ node 0.2 of the node 0 is stored in the CH field associated with the node 0 at the bottom of the stack. Instead of carrying the child $ELCA\_CAN$ 0.2 of the node 0 from the state shown in Figure 7(d) to the state shown in Figure 7(h), we can at the step shown in Figure 7(d) update the witness node of 0 from [0.1.1, 0.2.2.2.1] to [0.1.1, 0.3.1.1] after 0.2.2 and 0.2 are popped out and before 0.3 is pushed onto the stack, and update at the step shown in Figure 7(e) the witness node array of 0 from [0.1.1, 0.3.1.1] to [0.1.1, 0.4.1.1]. In the last step (Figure 7(i)) after popping out 0.4.2, we update the witness node array of 0 to [0.1.1, 0.5.1] and determine that 0 is an $ELCA$ node. Essentially, we remove the need of storing child $ELCA\_CAN$ nodes in the

elca_can	witness nodes	SIB	CH
0	[0.1.1, 0.2.2.2.1]	[]	[]

(a) $v_1 = 0.1.1$; $elca\_can_{v_1} = 0$; Figure 5(b); push 0 to stack.

elca_can	witness nodes	SIB	CH
0.2	[0.2.1.1, 0.2.2.2.1]	[]	[]
0	[0.1.1, 0.2.2.2.1]	[]	[]

(b) $v_1 = 0.2.1.1$: $elca\_can_{v_1} = 0.2$; Figure 5(b); push 0.2 to stack.

elca_can	witness nodes	SIB	CH
0.2.2	[0.2.2.1.1, 0.2.2.2.1]	[]	[]
0.2	[0.2.1.1,0.2.2.2.1]	[]	[]
0	[0.1.1,0.2.2.2.1]	[]	[]

(c) $v_1 = 0.2.2.1.1$: $elca\_can_{v_1} = 0.2.2$; Figure 5(b); push 0.2.2 to stack.

elca_can	witness nodes	SIB	CH
0.3.2	[0.3.2.1.1,0.3.2.2.1]	[]	[]
0	[0.1.1, 0.3.1.1]	[]	[0.2]

(d) $v_1 = 0.3.2.1.1$: $elca\_can_{v_1} = 0.3.2$; Figure 5(c); pop out 0.2.2 and 0.2 and determine them as ELCAs; add 0.2 to top entry's CH; push 0.3.2 to stack.

elca_can	witness nodes	SIB	CH
0.3.3	[0.3.3.1.1, 0.3.3.2.1]	[0.3.2]	[]
0	[0.1.1, 0.4.1.1]	[]	[0.2,0.3.2]

(e) $v_1 = 0.3.3.1.1$: $elca\_can_{v_1} = 0.3.3$; Figure 5(c); pop out 0.3.2 and determine it as an ELCA; add 0.3.2 to 0.3.3's SIB; push 0.3.3 to stack.

elca_can	witness nodes	SIB	CH
0.3.4	[0.3.4.1.1, 0.3.4.2.1]	[0.3.2,0.3.3]	[]
0	[0.1.1, 0.4.1.1]	[]	[0.2,0.3.2,0.3.3]

(f) $v_1 = 0.3.4.1.1$; $elca\_can_{v_1} = 0.3.4$; Figure 5(c); pop out 0.3.3 and determine it as an ELCA; add 0.3.3 to 0.3.4's SIB; push 0.3.4 to stack.

elca_can	witness nodes	SIB	CH
0.3	[0.3.1.1, 0.3.4.2.1]	[]	[0.3.2, 0.3.3, 0.3.4]
0	[0.1.1, 0.4.1.1]	[]	[0.2,0.3.2,0.3.3,0.3.4]

(g) $v_1 = 0.3.5.1$; $elca\_can_{v_1} = 0.3$; Figure 5(d); pop out 0.3.4 and determine it as an ELCA; add 0.3.4 entry's SIB list and 0.3.4 to 0.3's CH; push 0.3 to stack.

elca_can	witness nodes	SIB	CH
0.4.2	[0.4.2.1.1,0.4.2.2.1]	[]	[]
0	[0.1.1, 0.4.1.1]	[]	[0.2,0.3.2,0.3.3,0.3.4,0.3]

(h) $v_1 = 0.4.2.1.1$; $elca\_can_{v_1} = 0.4.2$; Figure 5(c); pop out 0.3 and determine it as an ELCA; push 0.4.2 to stack.

(i) No more "XML" nodes: clean up the stack; pop out 0.4.2 and 0 and determine them as ELCAs; Stack becomes empty.

Figure 7: States of stack during evaluation of "XML David"

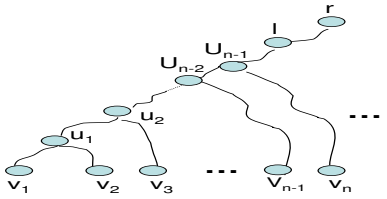


Figure 8: Optimizing the history information of an $ELCA\_CAN$ node

stack's CH fields and carrying them around by reusing the computation of $elca\_can()$ in the function $isELCA()$ and by doing some of the work in $isELCA()$ (searching for $ELCA$ witness nodes) as early as possible.

Reducing $|SIB|$. Assume at some point in the processing of the algorithm, the following list of $ELCA\_CAN$ nodes are computed in the exact order as they appear—r, v_1, v_2, ..., v_n, l (See Figure 8). The algorithm presented in Figure 6 will at some point push the node r onto the stack; push v_1 onto the stack; pop out v_1, push v_2, and add v_1 to the SIB field associated with v_2; pop out v_2, push v_3, and add v_1 and v_2 to the SIB field associated with v_3. When the algorithm pushes v_n onto the stack, the SIB field associated with v_n contains $v_1, \ldots, v_{n-1}$. We only describe the basic idea of the optimization to reduce the number of nodes stored in the SIB field. The idea is that we only need to store v_1 in the SIB field of v_2; u_1 in the SIB field of v_3; ...; u_{n-2} in the SIB field of v_n.

5. EXPERIMENTAL EVALUATION

System Implementation and Setup We have implemented in Java a prototype called XKeywordSearch to evaluate the proposed Indexed Stack algorithm and the two core algorithms in [10].

We have run XKeywordSearch on both real and synthetic data, respectively, DBLP [5] and XMark [16] data. The experiments have been done on a 766 MHz computer with 512MB of RAM. We only report the experimental results on the DBLP data in this paper; the results on XMark are similar.

The DBLP data was first grouped by journal and conference names, then by years. The size of the XML file of DBLP data after grouping is 120MB. The depth of the DBLP tree is 10; the number of distinct keywords is $180,126$; the number of nodes in the tree is $6,267,592$.

We evaluated the Indexed Stack algorithm, DIL and RDIL discussed in Section 1 for the $ELCA$ query semantics by varying the number and frequency of keywords both on hot cache and on cold cache. We report only results on hot cache in this paper. The relationships among three evaluated algorithms on cold cache are similar in the sense that if one algorithm wins another algorithm in the hot cache it also wins in the corresponding cold cache experiment but the differences are smaller because of dominance of the disk access. For example, in the hot cache experiments shown in Figure 9(a), the response time of the Indexed Stack algorithm for a query with two keywords of frequencies of 10 and 10000 is below 10 milliseconds; in the cold cache experiments, the response time of the Indexed Stack algorithm for the same query is close to 100 milliseconds. But the response time of the DIL algorithm does not increase significantly from hot cache to cold cache experiments.

One hundred queries were randomly selected for each experiment by a script. Note that when the script fails to choose a sufficient number of keywords of a specified frequency, it chooses keywords with frequencies close to the specified frequency. Each query was run three times and the average time was reported.

Search Performance First, we compare the search performances of the Indexed Stack (IS) algorithm and the DIL algorithm for finding all query results. There is no point to run the RDIL algorithm to find all query results because it is designed for returning top m answers and it has higher complexity than the IS algorithm. For space reason, we do not report experiments where the response time of both algorithms are less than 100 milliseconds.

In Figure 9(a) each query contains two keywords. The performance of the DIL algorithm degrades linearly when the size of the large inverted list increases, while the response time of the IS algorithm is almost constant, linear in the size of the smaller keyword list, and its performance is orders of magnitude better than DIL.

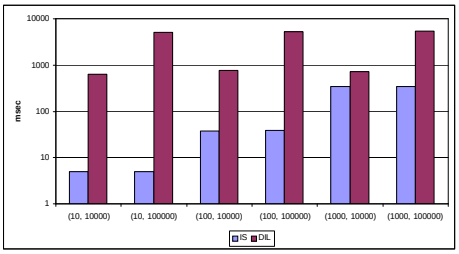
In the experiments shown in Figure 9(b), we vary the number of keywords from two to five. Each query has a keyword of small frequency shown on the top of Figure 9(b), while the frequency of all other keywords in the query is fixed at 100000. We vary the small frequency from 10 to 10000. As can been seen from Figure 9(b), when the number of the keywords is fixed, the performance of the DIL algorithm is essentially independent of $|S_1|$ when the small frequency increases from 10 to 10000, while the performance of the IS algorithm degrades linearly when the size of the small frequency increases. When the small frequency is fixed, the performance of IS is essentially constant while the performance of DIL degrades linearly when the number of keywords increases. As demonstrated in Figure 9(a), Figure 9(b) shows that the performance of the IS algorithm is orders of magnitude better than DIL.

We also stress tested the Indexed Stack algorithm on queries where all keywords have the same frequency. The experiments showed that although DIL often performs better than IS, the difference is not significantly. It is less than 5% in most experiments and less than 12% on average.

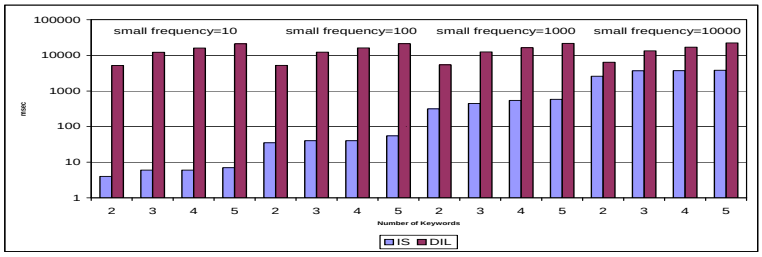
Next, we compare the search performance of the Indexed Stack algorithm and the RDIL algorithm for returning only the top ten query results. The DIL algorithm is not evaluated in this set of experiments because both the DIL and IS algorithms have to find all query results to determine the top ten answers and the experiments shown in Figures 9(a) and 9(b) in finding all query results have showed that IS is a better choice than DIL. As discussed in Section 3, there is no guarantee that RDIL can always find the top ten queries without having to compute all query results.

We evaluated the queries in Figure 9(a) and Figure 9(b) and reported the time on returning the top ten query answers in Figure 10(a) and Figure 10(b) respectively. We used a ranking module that is identical to the one used in the experiments of [10]. Both Figure 10(a) and Figure 10(b) show that the Indexed Stack algorithm performed significantly better than the RDIL algorithm.

There is a space where RDIL can outperform IS (and DIL) and here is a scenario that exhibits the conditions under which this happens. Consider a query "w_1 w_2" on a XML document that contains a large number of occurrences of w_1 and w_2, and only ten pairs of w_1 and w_2 have non-root nodes as their lowest common ancestors. Assume that the ten pairs of w_1 and w_2 nodes have higher ranking than all other w_1 and w_2 nodes before them in the document. The RDIL algorithm outperforms the IS algorithm for the above query "w_1 w_2" because the IS algorithm has to scan to the end of one of the two inverted lists to return the top ten answers while the RDIL algorithm starts from inverted lists sorted by ranking scores and can terminate much earlier than IS. As one direction of future work, we plan to investigate how to return top m answers without having to completely scan the smallest inverted list, by either adjusting the ranking mechanism or relaxing the exact top m requirement to approximate top m query answers.

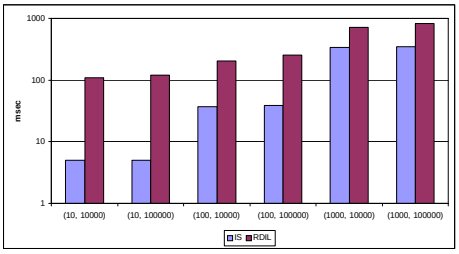


(a) queries contain two keywords; frequencies shown on X-axis

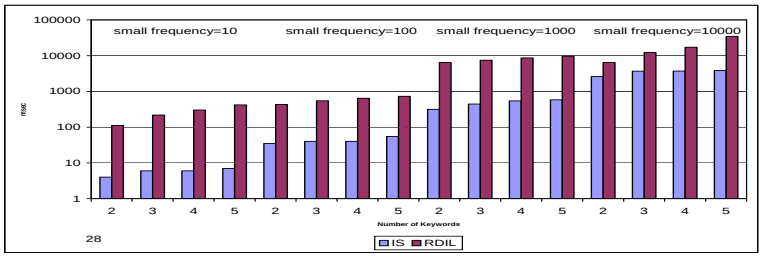


(b) varying the number of keywords from 2 to 5; large Frequency= 100000

Figure 9: Finding all query answers (evaluating the Indexed Stack algorithm and DIL)



(a) queries contain two keywords; frequencies shown on X-axis



(b) varying the number of keywords; large Frequency= 100000

Figure 10: Finding top 10 query answers (evaluating the Indexed Stack algorithm and RDIL)

6. CONCLUSIONS

We have presented an efficient keyword search algorithm, named Indexed Stack, that returns nodes that contain all instances of all keywords in the query, after excluding the keyword instances that appear under nodes whose children already contain all keyword instances according to the query semantics proposed in [10]. We demonstrated the superiority of the Indexed Stack algorithm over DIL and RDIL in [10] both analytically and experimentally. We showed that the complexity is $O(kd|S_1|\log|S|)$ where k is the number of keywords in the query, d is the depth of the tree and $|S_1|$ ($|S|$) is the occurrence of the least (most) frequent keyword in the query. In comparison, the complexity of the best prior work algorithm is $O(kd|S|)$.

7. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[2] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. TeXQuery: A full-text search extension to XQuery. In *WWW*, 2004.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[4] S. Cohen, J. Namou, Y. Kanza, and Y. Sagiv. XSEarch: A semantic search engine for XML. In *VLDB*, 2003.

[5] DBLP. http://www.informatik.uni-trier.de/ ley/db.

[6] D. Florescu, D. Kossmann, and I. Manolescu. Integrating keyword search into XML query processing. In *WWW9*, 2000.

[7] N. Fuhr and K. GroSSjohann. XIRQL: A Query Language for Information Retrieval in XML documents. In *SIGIR*, 2001.

[8] H. Garcia-Molina, J. Ullman, and J. Widom. Database System Implementation. Prentice-Hall, 2000.

[9] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity Search in Databases. In *VLDB*, 1998.

[10] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

[11] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[12] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *ICDE*, 2003.

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

[14] Y. Li, C. Yu, and H. V. Jagadish. Schema-free XQuery. In *VLDB*, 2004.

[15] D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Deductive and Object-Oriented Databases*, pages 319–344, 1995.

[16] R.Busse et al. XMark, the XML benchmark project, http://monetdb.cwi.nl/xml.

[17] A. Schmidt, M. L. Kersten, and M. Windhouwer. Querying XML documents made easy: Nearest concept queries. In *ICDE*, 2001.

[18] A. Theobald and G. Weikum. Adding relevance to XML. In *WebDB*, 2000.

[19] A. Theobald and G. Weikum. The index-based XXL search engine for querying XML data with relevance ranking. In *EDBT*, 2002.

[20] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *SIGMOD*, 2005.