Efficient LCA based Keyword Search in XML Data

Yu Xu
Teradata
San Diego, CA
yu.xu@teradata.com

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

ABSTRACT

Keyword search in XML documents based on the notion of lowest common ancestors (*LCA*s) and modifications of it has recently gained research interest [2, 3, 4]. In this paper we propose an efficient algorithm called Indexed Stack to find answers to keyword queries based on XRank's semantics to LCA [2]. 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 [2] is $O(kd|S|)$. We analytically and experimentally evaluate the Indexed Stack algorithm and the two core algorithms in [2]. 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.

Categories and Subject Descriptors:
H.3.3 [**Information Systems**]:INFORMATION STORAGE AND RETRIEVAL—Information Search and Retrieval

General Terms: Algorithms

Keywords: LCA, XML, Keyword, Search

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 [2, 3, 4]. 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 [2], 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 [2] 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 [2] 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 [2] 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 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 [2, 3, 4]. Section 4 presents the Indexed Stack algorithm, and also provides the complexity analysis of the Indexed Stack algorithm and the algorithms in [2] for both main memory and disk accesses.

2. ELCA QUERY SEMANTICS

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 $v = v'$.

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

CIKM'07, November 6–8, 2007, Lisboa, Portugal.

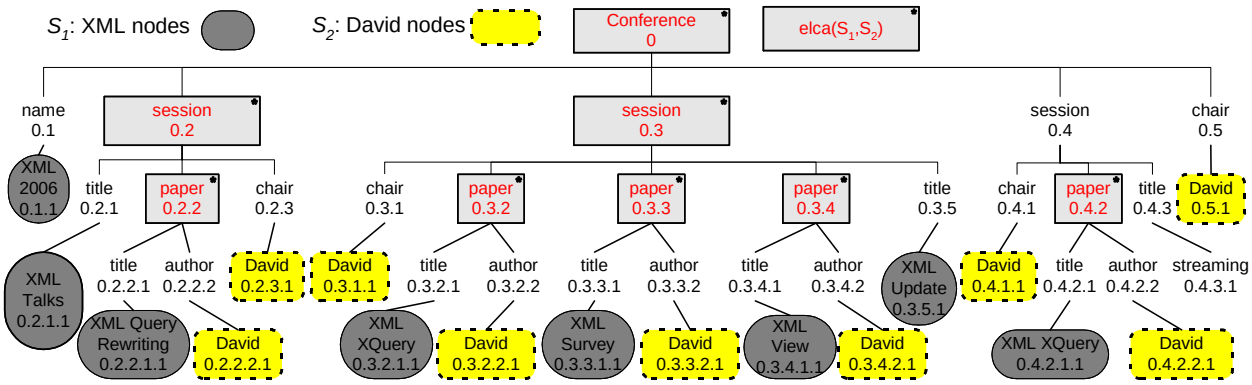

Figure 1: Example XML document

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 [2], 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 $elca(S_1, \ldots, S_k) = \{v \mid \exists n_1 \in S_1, \ldots, n_k \in S_k (v = lca(n_1, ..., n_k) \wedge \quad \forall i (1 \leq i \leq k) \nexists x (x \in lca(S_1, \ldots, S_k) \wedge child(v, n_i) \preceq_a x)) \}$, 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. Notice that the above definition is based on LCAs and is expressed differently than but it is equivalent to [2]. 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 *Smallest Lowest Common Ancestor (SLCA) of k sets* $S_1, \ldots, S_k$ is defined to be

$$slca(S_1, \ldots, S_k) = \{v | v \in lca(S_1, \ldots, S_k) \wedge \forall v' \in lca(S_1, \ldots, S_n) \ v \neq v'\}$$

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 $slca(S_1, \ldots, S_k) \subseteq elca(S_1, \ldots, S_k) \subseteq lca(S_1, \ldots, S_k)$.

Similarly to [2, 4], 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. We focus on the three most closely related works: XRank ([2]), Schema-Free XQuery ([3]) and XKSearch ([4]), all of which base keyword search in XML on the notation of LCAs of the nodes containing keywords.

XRank ([2]) 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. It also extends PageRank's ranking mechanism to XML by taking the nested structure of XML into account.

XKSearch ([4]) 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 $O(kd|S_1| \log |S|)$. [4] also extends the algorithm computing $slca(S_1, \ldots, S_k)$ to compute all LCAs of k sets (i.e., $lca(S_1, \ldots, S_k)$).

Schema-Free XQuery ([3]) uses the idea of *Meaningful LCA* (MLCA), and proposes a stack based sort merge algorithm which scans to the end of all inverted lists. The complexity of the algorithm in [3] is $O(kd|S|)$. [3] 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 [3] 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 $slca(S_1, \ldots, S_k) \subseteq elca(S_1, \ldots, S_k)$.

4. INDEXED STACK ALGORITHM

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 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.

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).

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

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].

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.

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

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. The following shows $elca\_can(v)$ for each v in S_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 briefly describe how prior work ([4]) can be used to efficiently compute $elca\_can(v)$.

The key property of SLCA search in [4] is that, given two keywords k_1 and k_2 and a node v that contains keyword k_1, one need not inspect the whole node list of keyword k_2 in order to discover potential solutions. Instead, one only needs to find the left and right match of in the list of k_2, where the left (right) match is the node with the greatest (least) id that is smaller (greater) than or equal to the id of v. The property generalizes to more than two keywords. Since $elca\_can(v)$ for $v \in S_1$ is the node l where

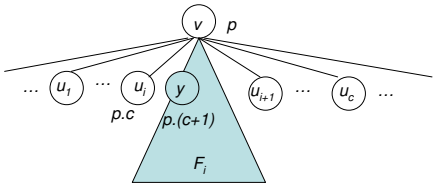

Figure 2: v and its $ELCA\_CAN$ children

$\{l\}$=$elca\_can(\{v\}; S_2, \ldots, S_k)$= $slca(\{v\}, S_2, \ldots, S_k)$, the time complexity of computing $elca\_can(v)$ is $O(kd\log|S|)$ by using the algorithm in [4].

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 $ELCA\_CAN$ 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, there must exist *witness nodes* $n_1, \ldots, n_k$ under an $ELCA$ node v 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 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. 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 $isELCA(v, ch)$ 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. 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 the first keyword list 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 the second keyword list 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

$isELCA(v, child\_elcacan(v))$ is $O(kd \log |S||child\_elcacan(v)|)$.

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 every $ELCA\_CAN$ node x_i under v [1], we can just first 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 the original ancestor-descendant relationships of all $ELCA\_CAN$ nodes in the input document. However, such a straightforward approach has the following disadvantages: 1) 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; 2) 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.

We propose a "one pass" stack-based algorithm named the Indexed Stack algorithm. The Indexed Stack algorithm need not keep all $ELCA\_CAN$ nodes in memory; it uses a stack whose depth is bounded by the depth of the tree. 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 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.

The challenging issue that the Indexed Stack Algorithm has to deal with is illustrated 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. 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 in the processing, 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 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. Based on some key tree properties, the Indexed Stack algorithm knows when an $ELCA\_CAN$ node needs to be stored in the stack and when it can be discarded.

[1] $child\_elcacan(v)$ 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 without any disk lookup.

	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	$O(k^2 d\|S\| p \log \|S\| + k^2 d\|S\|^2)$	$O(k^2 d\|S\| p \log \|S\| + k^2 d\|S\|^2)$

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

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 $isELCA(v, |child\_elcacan(v)|)$ once is $|child\_elcacan(v)|kd \log |S|$ (Section 4.3). 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. 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 [2, 4]. The complexity analysis of the Indexed Stack, the two algorithms in [2], 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. Algorithm details and experimental results on finding all query answers and only top m answers are in the full version of the paper [1].

5. REFERENCES

[1] http://db.ucsd.edu/pubsFileFolder/288.pdf.

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

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

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