Efficient Keyword Search for Smallest LCAs in XML Databases *

Yu Xu
Department of Computer Science & Engineering
University of California, San Diego
yxu@cs.ucsd.edu

Yannis Papakonstantinou
Department of Computer Science & Engineering
University of California, San Diego
yannis@cs.ucsd.edu

ABSTRACT

Keyword search is a proven, user-friendly way to query HTML documents in the World Wide Web. We propose keyword search in XML documents, modeled as labeled trees, and describe corresponding efficient algorithms. The proposed keyword search returns the set of smallest trees containing all keywords, where a tree is designated as "smallest" if it contains no tree that also contains all keywords. Our core contribution, the Indexed Lookup Eager algorithm, exploits key properties of smallest trees in order to outperform prior algorithms by orders of magnitude when the query contains keywords with significantly different frequencies. The Scan Eager variant is tuned for the case where the keywords have similar frequencies. We analytically and experimentally evaluate two variants of the Eager algorithm, along with the Stack algorithm [13]. We also present the XKSearch system, which utilizes the Indexed Lookup Eager, Scan Eager and Stack algorithms and a demo of which on DBLP data is available at `http://www.db.ucsd.edu/projects/xksearch`. Finally, we extend the Indexed Lookup Eager algorithm to answer Lowest Common Ancestor (LCA) queries.

1. INTRODUCTION

Keyword search is a proven user-friendly way of querying HTML documents in the World Wide Web. Keyword search is well-suited to XML trees as well. It allows users to find the information they are interested in without having to learn a complex query language or needing prior knowledge of the structure of the underlying data [1, 5, 12, 15, 16, 18]. For example, assume an XML document named "School.xml", modeled using the conventional labeled tree model in Figure 1, that contains information including classes, projects, etc. A user interested in finding the relationships between "John" and "Ben" issues a keyword search "John, Ben" and the search system returns the most specific relevant answers - the subtrees rooted at nodes 0.1.1, 0.1.2 and 0.2.0.0. The meaning of the answers is obvious to the user: "Ben" is a TA for "John" for the CS2A class, "Ben" is a student in the CS3A class taught by "John", both "John" and "Ben" are participants in a project.

According to the *Smallest Lowest Common Ancestor (SLCA)* semantics, the result of a keyword query is the set of nodes that (i) contain the keywords either in their labels or in the labels of their descendant nodes and (ii) they have no descendant node that also contains all keywords. The answer to the keyword search "John, Ben" is the node list [0.1.1, 0.1.2, 0.2.0.0]. The subtree rooted at the Class node with id 0.1.1 is a better answer than the subtrees rooted at "Classes" or "School" because it connects "John" and "Ben" more closely than the "Classes" or "School" elements.

We can use an XML query language such as XQuery to search on "John, Ben" to find the most specific elements. One possible query is shown in Figure 2. It is complex and difficult to be executed efficiently. We could write a query that is schema specific and more efficient. However it would require existence and knowledge of the schema of School.xml, knowledge of the "roles" John and Ben may play in the document and knowledge of the relationships they may have.

We make the following technical contributions in the paper:

- We propose two efficient algorithms, Indexed Lookup Eager and Scan Eager, for keyword search in XML documents according to the SLCA semantics. Both algorithms produce part of the answers quickly so that users do not have to wait long to see the first few answers.
- The Indexed Lookup Eager algorithm outperforms known algorithms and Scan Eager by orders of magnitude when the keyword search includes at least one low frequency keyword along with high frequency keywords. In particular, the performance of the algorithm primarily depends on the number of occurrences of the least frequent keyword and the number of keywords in the query; it does not depend significantly on the frequencies of the more frequent keywords of the query (the precise worst-case complexity for a main memory version and a disk-based version is provided). The Indexed Lookup Eager algorithm is important in practice since the frequencies of keywords typically vary significantly. In contrast Scan Eager is tuned for the case where the occurrences of the query's keywords do not vary significantly.
- We experimentally evaluate the Indexed Lookup Eager algorithm, the Scan Eager algorithm and the prior work Stack algorithm [13]. The algorithms are incorporated into the XKSearch system (XML Keyword Search), which utilizes B-tree indices provided by BerkeleyDB [4]. A demo of the XKSearch system, running on DBLP data, is available at `http://www.db.ucsd.edu/projects/xksearch`. The experiments show that the Indexed Lookup Eager algorithm outperforms the Scan Eager and the Stack algorithms

* Work supported by NSF ITR 313384

SIGMOD June 14-16, 2005, Baltimore, Maryland, USA

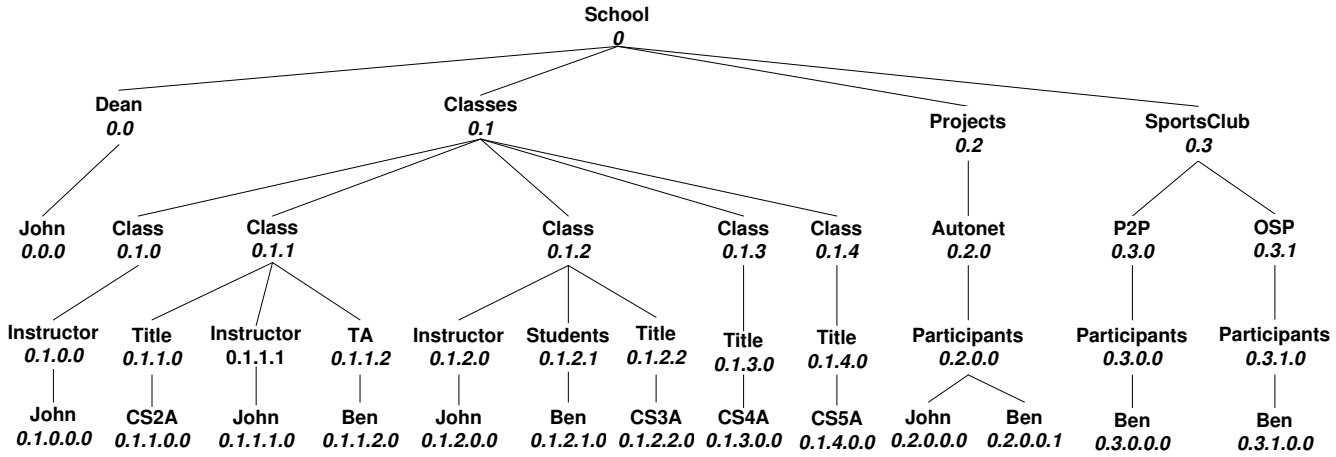


Figure 1: School.xml (each node is associated with its Dewey number)

```
⟨answer⟩ {
 for $a in document(''School.xml'')//*
 where empty(for $b in $a/*
             where some $c in $b//*
              satisfies $c=''John''
             and some $c in $b//*
              satisfies $c=''Ben''
            return $b)
 and some $d in $a//* satisfies $d=''John''
 and some $d in $a//* satisfies $d=''Ben''
 return $a }
⟨/answer⟩
```

Figure 2: XQuery: find answers for "John, Ben"

often by orders of magnitude when the keywords in the query have different frequencies, and loses only by a small margin when the keywords have similar frequencies. Indeed in the DBLP demo, only the Indexed Lookup Eager algorithm is used.

In Section 2 we introduce the notation and definitions used in the paper. Section 3 presents the SLCA problem and its solutions. We discuss the Indexed Lookup Eager and the Scan Eager algorithms, and the sort-merge based Stack algorithm [13]. Section 3 also provides the complexity of the main memory implementations of the algorithms. In Section 4, we discuss the XKSearch implementation of the Indexed Lookup Eager, Scan Eager and Stack algorithms and provide their complexity in terms of number of disk accesses. In Section 5, we discuss the *All Lowest Common Ancestor (LCA) problem* and extend the SLCA's Indexed Lookup algorithm to efficiently find LCAs in "shallow" trees. Our experimental results are discussed in Section 6. We discuss related work in Section 7 and conclude in Section 8.

2. NOTATION

We use the conventional labeled ordered tree model to represent XML trees. Each node v of the tree corresponds to an XML element and is labeled with a tag $\lambda(v)$. We assign to each node a numerical 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 $pre(v_1) < pre(v_2)$. The XKSearch implementation uses Dewey numbers as the id's. Prior work has shown that Dewey numbers are a good id choice [23]. In addition, Dewey numbers provide a straightforward solution to locating the LCA of two nodes. The usual $<$ relationship is assumed between any two Dewey numbers. For example, $0.1.0.0.0 < 0.1.1.1$. Obviously the $<$ relationship on Dewey numbers is compatible with the requirement for preorder numbering.

Given a list of k keywords $w_1, \ldots, w_k$ and an input XML tree T, an *answer subtree* of keywords $w_1, \ldots, w_k$ is a subtree of T such that it contains at least one instance of each keyword $w_1, \ldots, w_k$. A *smallest answer subtree* of keywords $w_1, \ldots, w_k$ is an answer subtree (of keywords $w_1, \ldots, w_k$) such that none of its subtrees is an answer subtree (of keywords $w_1, \ldots, w_k$). The result $slca(w_1, \ldots, w_k, T)$ of a keyword search $w_1, \ldots, w_k$ on an input XML tree T, is the set of the roots of all smallest answer subtrees of $w_1, \ldots, w_k$. For presentation brevity, we do not explicitly refer to the input XML tree T and simply write $slca(w_1, \ldots, w_k)$ when T is obvious.

Given a list of k keywords $w_1, \ldots, w_k$ and an input XML tree T, we assume S_i denotes the keyword list of w_i, i.e., the list of nodes whose label directly contains w_i sorted by id. $v \prec v'$ denotes that node v is an ancestor of node v'; $v \preceq v'$ denotes that $v \prec v'$ or $v = v'$. Given a node $v \in S$, v is called an ancestor node in S if there exists a node v' in S such that $v \prec v'$. When the set S is implied by the context, we simply say v is an ancestor node. Notice that if $v \prec v'$ then $pre(v) < pre(v')$, and the other direction is not always true.

The function $lca(v_1, \ldots, v_k)$ computes the *lowest common ancestor* or *LCA* of nodes $v_1, \ldots, v_k$ and returns null if any of the arguments is null. 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 and the cost of computing $lca(v_1, v_2)$ is $O(d)$ where d is the maximum depth of the tree. For example, the LCA of nodes $0.1.1.1.0$ and $0.1.1.2.0$ is the node $0.1.1$ in Figure 1. Given sets of nodes $S_1, \ldots, S_n$, a node v belongs to $lca(S_1, \ldots, S_k)$ if there exist $v_1 \in S_1, \ldots, v_k \in S_k$ such that $v = lca(v_1, \ldots, v_k)$. Then v is called an LCA of sets $S_1, \ldots, S_n$. A node v belongs to the *smallest lowest common ancestor (SLCA)* $slca(S_1, \ldots, S_k)$ of $S_1, \ldots, S_k$ if $v \in lca(S_1, \ldots, S_k)$ and $\forall u \in lca(S_1, \ldots, S_n)$ $v \not\prec u$. v is called a SLCA of sets $S_1, \ldots, S_n$ if

$v \in slca(S_1, \ldots, S_k)$.

Notice that the query result $slca(w_1, \ldots, w_k)$ is $slca(S_1, \ldots, S_k)$ and that $slca(S_1, \ldots, S_k)$= $removeAncestor(lca(S_1, \ldots, S_k))$ where $removeAncestor$ removes ancestor nodes from its input.

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 $pre(v)$[1]. $rm(v, S)$ ($lm(v, S)$) returns null when there is no right (left) match node. The cost of $lm(v, S)$ ($rm(v, S)$) is $O(d \log |S|)$ since it takes $O(\log |S|)$ steps (Dewey number comparisons) to find the right (left) match node and the cost of comparing two Dewey numbers is $O(d)$. The function $descendant(v_1, v_2)$ returns the other argument when one argument is null and returns the descendant node when v_1 and v_2 have ancestor-descendant relationship. The cost of the function $descendant$ is $O(d)$.

3. ALGORITHMS FOR FINDING THE SLCA OF KEYWORD LISTS

This section presents the core Indexed Lookup Eager algorithm, its Scan Eager variation and the prior work Stack algorithm [13].

A brute-force solution to the SLCA problem computes the LCAs of all node combinations and then removes ancestor nodes. Its complexity is $O(kd|S_1| \ldots |S_k|)$. Besides being inefficient the brute-force approach is *blocking*. After it computes an LCA $v = lca(v_1, \ldots, v_k)$ for some $v_1 \in S_1, \ldots, v_k \in S_k$, it cannot report v as an answer since there might be another set of k nodes $u_1, \ldots, u_k$ such that $v \prec lca(u_1, \ldots, u_k)$.

The complexity analysis given in this section is for main memory cases. We will give disk access complexity in Section 4 after we discuss the implementation details of how we compress and store keyword lists on disk. In the sequel we choose S_1 to be the smallest keyword list since $slca(S_1, \ldots, S_k) = slca(S_{i_1}, \ldots, S_{i_k})$, where $i_1, .., i_k$ 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 algorithms.

3.1 The Indexed Lookup Eager Algorithm (IL)

The Indexed Lookup Eager algorithm is based on four properties of SLCAs, which we explain starting from the simplest case where $k = 2$ and S_1 is a singleton $\{v\}$.

Property (1)

$$slca(\{v\}, S) = \{descendant(lca(v, lm(v, S)), lca(v, rm(v, S)))\}$$

According to the above Property (1), we compute the LCA of v and its left match in S, the LCA of v and its right match in S, and the singleton formed from the deeper node from the two LCAs is $slca(\{v\}, S)$. Property (1) is based on the following two observations. For any two nodes v_1, v_2 to the right (according to preorder) of a node v, if $pre(v) < pre(v_1) < pre(v_2)$, then $lca(v, v_2) \preceq lca(v, v_1)$; similarly, for any two nodes v_1, v_2 to the left of a node v, if $pre(v_2) < pre(v_1) < pre(v)$, then $lca(v, v_2) \preceq lca(v, v_1)$[2].

We generalize to arbitrary k when the first set is a singleton. Notice the recursiveness in Property (2).

Property (2) $slca(\{v\}, S_2, \ldots, S_k) = slca(slca(\{v\}, S_2, \ldots, S_{k-1}), S_k)$ *for* $k > 2$

[1]The right or left match of a node v in S is itself if $v \in S$. This may happen when a node's label contains multiple keywords.

[2]The two observations apply to inorder and postorder as well.

Next we generalize to arbitrary S_1.

Property (3)

$$slca(S_1, \ldots, S_k) = removeAncestor(\bigcup_{v_1 \in S_1} slca(\{v_1\}, S_2, .., S_k))$$

Property (3) straightforwardly leads to an algorithm to compute $slca(S_1, S_2, \ldots, S_k)$: first computes $\{x_i\} = slca(\{v_i\}, S_2, .., S_k)$ for each $v_i \in S_1$ ($1 \le i \le n$), $removeAncestor(\{x_1, \ldots, x_n\})$ is the answer. Each x_i is computed by using Properties (2) and (1).

The benefit of the above algorithm over the brute force approach is that for each node x_1 in S_1, the algorithm does not compute $lca(x_1, v_2, \ldots, v_k)$ for all $v_2 \in S_2, \ldots, v_k \in S_k$, but computes a single $lca(x_1, v_2, \ldots, v_k)$ where each v_i ($2 \le i \le k$) is computed by the match functions (lm and rm). The complexity of the algorithm is $O(|S_1| \sum_{i=2}^{k} d \log |S_i| + |S_1|^2)$ or $O(|S_1| kd log|S| + |S_1|^2)$ where $|S_1|$ ($|S|$) is the minimum (maximum) size of keyword lists S_1 through S_k because for each node x_1 in S_1 the algorithm needs to find a left and a right match in each one of the other $k-1$ keyword lists. Finding a match in list S_i costs $O(d \log |S_i|)$. Hence the total cost of match operations is $O(|S_1| d \sum_{i=2}^{k} \log |S_i|)$. The total cost of the lca and $descendant$ operations is $O(|S_1| kd)$ and hence is dominated by the cost of the match operations. The $|S_1|^2$ factor is attributed to the cost of removing ancestors operation.

The subroutine get_slca, based on the following two lemmas, computes $slca(S_1, S_2)$ efficiently by removing ancestor nodes on the fly.

LEMMA 1. *Given any two nodes v_i, v_j and a set S, if $pre(v_i) < pre(v_j)$ and $pre(slca(\{v_i\}, S)) > pre(slca(\{v_j\}, S))$, then $slca(\{v_j\}, S) \prec slca(\{v_i\}, S)$.*

LEMMA 2. *For any two nodes v_i, v_j and a set S such that $pre(v_i) < pre(v_j)$ and $pre(slca(\{v_i\}, S)) < pre(slca(\{v_j\}, S))$, if $slca(\{v_i\}, S)$ is not an ancestor of $slca(\{v_j\}, S)$, then for any v such that $pre(v) > pre(v_j)$, $slca(\{v_i\}, S) \not\prec slca(\{v\}, S)$.*

Consider S_1, S_2 sorted by id, where $S_1 = [v_1, \ldots, v_q]$. Let $L = [x_1, \ldots, x_q]$ where $x_1 = slca(\{v_1\}, S_2)$,. . .,$x_q = slca(\{v_q\}, S_2)$. According to Lemma 1, if $pre(x_i) > pre(x_j)$ where node x_j appears after x_i in L (that is, $j > i$), then x_j is an ancestor node. Thus when computing the list L, we can discard the out-of-order nodes such as x_j. The resulting list L' is in order and contains the nodes of $slca(S_1, S_2)$. However L' is not necessarily ancestor node free.

Consider any two adjacent nodes x, x' in L' where x' is after x. If x is not an ancestor of x', then x cannot be an ancestor of any node x'' that is after x' in L' (according to Lemma 2), which means x is a $SLCA$ of S_1, S_2. Lemma 1 and 2 together lead to the subroutine get_slca that computes $slca(S_1, S_2)$ efficiently. Line #5 in get_slca applies Lemma 1 to remove out-of-order nodes, and lines #6-8 apply Lemma 2 to identify a SLCA as early as possible. As can be seen from get_slca, at any time only three nodes (u, v, x) are needed in memory.

Consider S_1=[0.0.0, 0.1.0.0.0, 0.1.1.1.0, 0.1.2.0.0, 0.2.0.0.0], S_2=[0.1.1.2.0, 0.1.2.1.0, 0.2.0.0.1, 0.3.0.0.0, 0.3.1.0.0] (the keyword lists for "John" and "Ben" respectively). In the first iteration of the loop at line #3, $u = 0$, $v = 0.0.0$, x=0 (line #4). At the end of the first iteration $u = 0$ (line #8). In the second iteration, $v = 0.1.0.0.0$, $x = 0.1$, $u = 0.1$. In the third iteration, $v = 0.1.1.1.0$, $x = 0.1.1$, $u = 0.1.1$. In the fourth iteration, $v = 0.1.2.0.0$, $x = 0.1.2$ (line #4). Notice that the condition at

line #6 is true in the fourth iteration, and u (node 0.1.1) is determined to be a SLCA (line #7). Then u = 0.1.2 (line #8). In the last iteration, v = 0.2.0.0.0, x = 0.2.0.0, and node 0.1.2 is determined to be a SLCA (line #7). Finally node 0.2.0.0 is returned as a SLCA (line #10). Thus the answer to "John Ben" is [0.1.1, 0.1.2, 0.2.0.0].

```
subroutine get_slca(S1, S2)
1   Result = {}
2   u = 0        //u = root initially
3   for each node v ∈ S1 {
4      x = descendant(lca(v, lm(v, S2), lca(v, rm(v, S2))))
5      if (pre(u) ≤ pre(x))
6           if (u ⋠ x)
7              Result = Result ∪ {u};
8           u = x;
9   }
10  return Result ∪ {u}
```

We can derive an algorithm from get_slca to compute $slca(S_1, \ldots, S_k)$ efficiently:

$$slca(S_1, \ldots, S_k) = get\_slca(get\_slca(S_1, \ldots, S_{k-1}), S_k) \quad for \quad k > 2$$

based on the following Property (4),

$$Property\ (4) \quad slca(S_1, \ldots, S_k) = slca(slca(S_1, \ldots, S_{k-1}), S_k) \quad for \quad k > 2$$

An algorithm based on get_slca accesses S_1 and S_2 first, then S_3, S_4,..., S_k in order. It accesses the k keyword lists in just one round. It is a blocking algorithm since it only processes the last keyword list after it completely processes the first $k-1$ keyword lists and then starts to produce answers. Obviously the algorithm based on Property (3) is also a blocking algorithm.

The Indexed Lookup Eager Algorithm improves the algorithm based on get_slca by adding "eagerness"- it returns the first part of the answers without having to completely go through any of the keyword lists and it pipelines the delivery of SLCAs. Assume there is a memory buffer size of P nodes. The Indexed Lookup Eager algorithm first computes $X_2 = slca(X_1, S_2)$ where X_1 is the first P nodes of S_1. Then it computes $X_3 = scla(X_2, S_3)$ and so on, until it computes $X_k = slca(\ldots slca(X_1, S_1) \ldots S_k)$. All nodes in X_k except the last node are guaranteed to be SLCAs because of Lemma 1 and 2 and are returned. The last node of X_k (v in line #10) is carried on to the next operation (lines #6-9) to be determined whether it is a $SLCA$ or not. The above operation is repeated for the next P nodes of S_1 until all nodes in S_1 have been processed. The smaller P is, the faster the algorithm produces the first SLCA. If $P = 1$, again only three nodes are needed to be kept in memory in the whole process. However, a smaller P may delay the computation of all SLCAs when considering disk accesses. If $P \geq |S_1|$, the Indexed Lookup Eager algorithm is exactly the same as the algorithm based on get_slca. The complexity analysis of the Indexed Lookup Eager algorithm is the same as that of the algorithm based on Property (3) except that there is no operation to remove ancestor nodes from a set. Thus the complexity of the Indexed Lookup Eager algorithm is $O(|S_1|kdlog|S|)$ where $|S_1|$ ($|S|$) is the minimum (maximum) size of keyword lists S_1 through S_k. Consider the query "John Ben Class" applied on the data of Figure 1. The keyword lists for "John", "Ben", "Class" are S_1=[0.0.0, 0.1.0.0.0, 0.1.1.1.0, 0.1.2.0.0, 0.2.0.0.0], S_2=[0.1.1.2.0, 0.1.2.1.0, 0.2.0.0.1, 0.3.0.0.0, 0.3.1.0.0] and S_3=[0.1.0, 0.1.1, 0.1.2, 0.1.3, 0.1.4] respectively. Assume $P = 3$. In the first iteration of the loop at line #2, B=[0.0.0, 0.1.0.0.0,0.1.1.1.0] (line #3). After the computation at line #4-5, B=$get\_slca(get\_slca$([0.0.0, 0.1.0.0.0, 0.1.1.1.0],S_2) ,S_3), which is B=get_slca([0.1.1],S_3)= [0.1.1]. Initially $v = null$. Line #6-9 has no effect. As mentioned before, every node in B except the last one is a SLCA and returned (line #10, #11). In the first iteration, line #11 outputs nothing and v (node 0.1.1) is carried to the next iteration to be determined whether it is a SLCA or not. In the second iteration of loop at line #2, the rest nodes of S_1 is read, B=[0.1.2.0.0, 0.2.0.0.0] (line #3). After executing line #4 and #5, $B = get\_slca(get\_slca([0.1.2.0.0, 0.2.0.0.0], S_2), S_3)$, which is $B = get\_slca([0.1.2, 0.2.0.0], S_3)$ = [0.1.2]. The condition at line #8 is true, thus v (node 0.1.1) carried from the first iteration is determined to be a SLCA and returned (line #9). Then v = 0.1.2 at line #10. Line #11 outputs nothing in this iteration again. Since there are no more nodes in S_1, line #13 is executed. Thus the Indexed Lookup Algorithm returns [0.1.1, 0.1.2] for "John Ben Class".

ALGORITHM 1 (INDEXED LOOKUP EAGER ALGORITHM).

```
Assume we have a memory buffer B of size P nodes
1  v = null
2  while( there are more nodes in S1) {
3     Read P nodes of S1 into buffer B.
4     for i = 2 → k
5        B = get_slca(B, Si)
6     if (v ≠ null && getFirstNode(B) ≺ v)
7           removeFirstNode(B)
8     if ( v ≠ null && v ⊀ getFirstNode(B))
9        output v
10    v = removeLastNode(B)
11    output B; B={}
12 }
13 output v
```

3.2 Scan Eager Algorithm

When the occurrences of keywords do not differ significantly, the total cost of finding matches by lookups may exceed the total cost of finding matches by scanning the keyword lists. We implement a variant of the Indexed Lookup Eager Algorithm, named Scan Eager Algorithm, to take advantage of the fact that the accesses to any keyword list are strictly in increasing order in the Indexed Lookup Eager algorithm. The Scan Eager algorithm is exactly the same as the Indexed Lookup Eager algorithm except that its lm and rm implementations scan keyword lists to find matches by maintaining a cursor for each keyword list. In order to find the left and right match of a given node with id p in a list S_j, the Scan Eager algorithm advances the cursor of S_j until it finds the node that is closest to p from the left or the right side. Notice that nodes from different lists may not be accessed in order, though nodes from the same list are accessed in order.

The complexity of the Scan Eager algorithm is $O(k|S_1|+ d\sum_2^k |S_i|)$, or $O(kd|S|)$ because there are $O(\sum_2^k |S_i|)$ Dewey number comparisons, $O(k|S_1|)$ lca and $descendant$ operations.

3.3 The Stack Algorithm

The stack based sort-merge algorithm (DIL) in XRANK [13], which also uses Dewey numbers, is modified to find SLCAs and is called the Stack Algorithm here. Each stack entry has a pair of components $(id, Keywords)$. Assume the id components from the bottom entry to a stack entry en are $id_1, id_2, \ldots, id_m$ respectively. Then the stack entry en denotes the node with the Dewey number $id_1.id_2. \ldots .id_m$. $Keywords$ is an array of length k of boolean values where $Keywords[i] = T$ means that the subtree rooted at the node denoted by the stack entry directly or indirectly

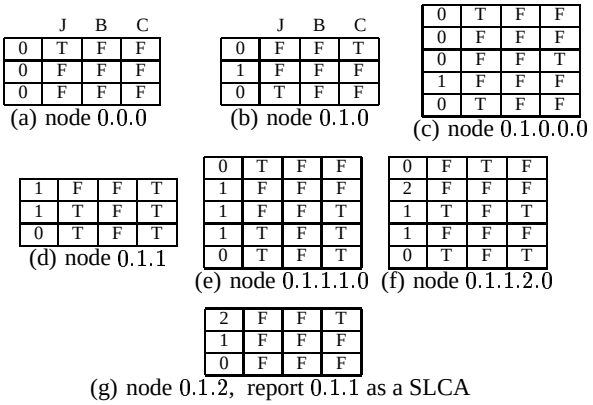


Figure 3: States of stack, where J stands for "John", B stands for "Ben" and C stands for "Class"

contains the keyword w_i. For example, the top entry of the stack in Figure 3(b) denotes the node 0.1.0, and the middle entry denotes the node 0.1. The Stack algorithm merges all keyword lists and computes the longest common prefix of the node with the smallest Dewey number from the input lists and the node denoted by the top entry of the stack. Then it pops out all top entries containing Dewey components that are not part of the common prefix. If a popped entry en contains all keywords, then the node denoted by en is a SLCA. Otherwise the information about which keywords en contains is used to update its parent entry's $Keywords$ array. Also a stack entry is created for each Dewey component of the smallest node which is not part of the common prefix, effectively pushing the smallest node onto the stack. The above action is repeated for every node from the sort merged input lists.

Consider again the query "John, Ben, Class" applied on the data of Figure 1. The keyword lists for "John, Ben, Class" are [0.0.0, 0.1.0.0.0, 0.1.1.1.0, 0.1.2.0.0, 0.2.0.0.0], [0.1.1.2.0, 0.1.2.1.0, 0.2.0.0.1, 0.3.0.0.0, 0.3.1.0.0] and [0.1.0, 0.1.1, 0.1.2, 0.1.3., 0.1.4] respectively. Initially, the smallest node is 0.0.0 and Figure 3(a) shows the initial state of the stack where $Keywords[1] = T$ in the top entry denotes that the node (0.0.0) represented by the top entry contains the first keyword "John". The next smallest node is the "Class" node 0.1.0 . Since the longest common prefix of 0.0.0 and 0.1.0 is 0 (line #4), the top two entries are popped out (line #6). 0.0.0 contains "John" and this information is passed to the current top entry (lines #12-13). Then two new entries from the two components of node 0.1.0 that are not among the longest common prefix are pushed into the stack (line #16). Notice that in Figure 3(b) $Keywords[1] = T$ in the bottom entry denotes that the node 0 (School) contains the keyword "John". Each figure in Figure 3 shows the state of the stack after processing the node shown in the caption. For example, when the algorithm processes node 0.1.2, the initial stack is shown in Figure 3(f) and the stack after processing 0.1.2 is shown in Figure 3(g). The longest common prefix of 0.1.2 and the stack (0.1.1.2.0) is 0.1 (line #4). Thus the top three entries are popped out (line #6). When popping out the third entry, the algorithm reports 0.1.1 as a SLCA since its $Keywords$ array contains all T (line #7). Notice that the T for "Ben" from the top entry in Figure 3(f) is used in the decision that the third entry is a SLCA.

The complexity of Stack is $O(d\sum_{i=1}^{k}|S_i|)$ since both the number of lca operations and the number of Dewey number comparisons are $\sum_{i=1}^{k}|S_i|$. The Scan Eager algorithm has several advantages over the Stack algorithm. First, the Scan Eager algorithm starts from the smallest keyword list, does not have to scan to the end of every keyword list and may terminate much earlier than the Stack algorithm as we will see in an example soon. Second, the number of lca operations of the Scan Eager algorithm ($O(k|S_1|)$) is usually much less than that of the Stack algorithm ($O(\sum_{i=1}^{k}|S_i|)$). Third, the Stack algorithm operates on a stack whose depth is bounded by the depth of the input tree while the Scan Eager algorithm with $P = 1$ only needs to keep three nodes in the whole process and no push/pop operations are involved.

```
ALGORITHM 2  (STACK ALGORITHM). {

1  stack=empty
2  while (has not reached the end of all keyword lists) {
3    v = getSmallestNode()
     //find the largest p such that stack[i] = v[i], 1 ≤ i ≤ p
4    p = lca(stack, v)
5    while (stack.size > p) {
6     stackEntry=stack.pop()
7     if (isSLCA(stackEntry)) {
       //Any other stack entry cannot represent a SLCA
8      output stackEntry as a SLCA
9      set all entries of the Keywords array of
       any stack entry all falses
10    }
11    else { //pass keyword witness information to the top entry
12     for (j = 1 → k)
13      if (stackEntry.Keywords[j]=true)
        stack.top.Keywords[j]=true
14    }
15   }
     //add non-matching components of v to stack
16   for (p < j ≤ v.length)  stack.push(v[j],[])
     stack.top.Keywords[i]=true        17
18 }
19 check entries of the stack and return any SLCA if exists
}
isSLCA(stackEntry) {
   if (stackEntry.Keywords[i]=true     for 1 ≤ i ≤ k)
   return true
   else return false }

getSmallestNode {
/* returns the node v with the smallest Dewey number from
all keyword lists and advances the cursor of the list where v
is from. Assume v is an array consisting of its Dewey num-
ber components. For example v is [0|1|3] if its Dewey
number is 0.1.3 */ }
```

We consider again the query "John, Ben, Class" applied on the data of Figure 1 and assume the tree does not have the "Ben" node with id 0.2.0.0.1 to show the first advantage of the Scan Eager algorithm over the Stack algorithm. Hence S_1=[0.0.0,0.1.0.0.0, 0.1.1.1.0, 0.1.2.0.0, 0.2.0.0.0], S_2=[0.1.1.2.0, 0.1.2.1.0, 0.3.0.0.0, 0.3.1.0.0][3] , and S_3=[0.1.0, 0.1.1, 0.1.2, 0.1.3, 0.1.4].

For node 0.1.2.0.0 in S_1 the Scan Eager algorithm finds its match 0.1.2.1.0 in S_2, and computes their LCA 0.1.2; then it finds the match 0.1.2.1.0 for 0.2.0.0.0 in S_2, and computes their LCA 0. Since node 0 is an ancestor of node 0.1.2, node 0 is discarded and no further access to S_3 is needed. The last node in S_3 accessed by the Scan Eager algorithm is node 0.1.2. The Stack algorithm has to visit all S_3 nodes because it cannot tell whether the last node

[3]For the sake of this example, we neglect that $|S_2| < |S_1|$.

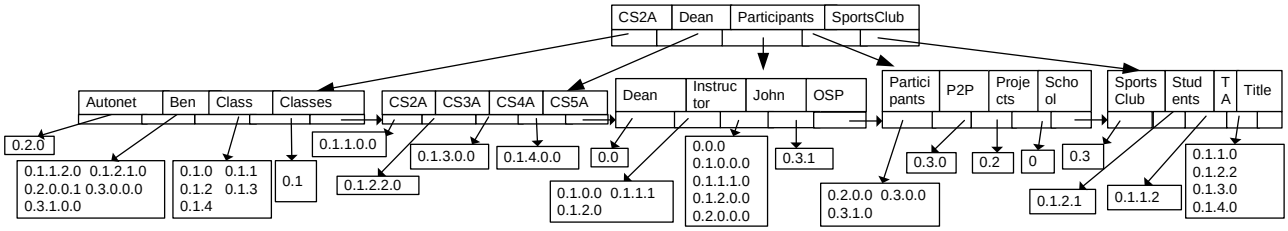


Figure 4: B+ tree from the data of Figure 1 for Scan Eager and Stack algorithms

$0.2.0.0.0$ may lead to a SLCA or not until it comes to process node $0.2.0.0.0$ and it has to repeatedly compute the longest common ancestor of each S_3 node with the node represented by the top entry of the stack. Notice that the "Class" list may have arbitrarily many nodes after node $0.1.2$ and before node $0.2.0.0.0$[4] that Stack has to access but Scan Eager does not need to.

4. XKSEARCH SYSTEM IMPLEMENTATION

In this section we present the architecture of the XKSearch implementation, then discuss how the keyword lists are compressed and stored on disk-based B tree index structures, and finally provide disk access complexity analysis summarized in Table 1 for the three algorithms discussed in Section 3 - Indexed Lookup Eager, Scan Eager and Stack.

We implemented the Indexed Lookup Eager, Scan Eager and Stack algorithms in Java using the Apache Xerces XML parser and Berkeley DB [4]. The architecture of the implementation (XKSearch) is shown in Figure 6. The LevelTableBuilder reads an input XML document T and outputs a level table LT. The inverted index builder reads in the level table LT and outputs a keyword list kl for each keyword w in T. Those keyword lists are stored in a B-tree structure that allows efficient implementation of the match operations.

The index builder also generates a frequency table, which records the frequencies of keywords in T, is read into memory by the initializer, and is stored as a hash table. The query engine accepts a keyword search, uses the frequency hash table to locate the smallest keyword list, executes the Indexed Lookup Eager, Scan Eager and the Stack algorithms and returns all SLCAs.

For performance reasons, Dewey numbers are compressed. We introduce a *level table* LT with d entries where d is the depth of the input tree. The entry $LT(i)$ denotes the maximum number of bits needed to store the i-th component in a Dewey number, i.e., $LT(i) = \lceil \log(c) \rceil$, where c is the number of children of the node at the level of $i-1$ that has the maximum number of children among all nodes at the same level. The root is at level 1, $LT(1) = 1$[5]. In general $\lceil \frac{\sum_1^i LT(i)}{8} \rceil$ bytes are needed to store the Dewey number of a node at level i. The level table LT for Figure 1 is

i	1	2	3	4	5
LT(i)	1	2	3	2	1

There are two types of B tree structures implemented in XKSearch; the first is for the Indexed Lookup Eager algorithm, the second is for the Scan Eager and the Stack algorithms. In the implementation of the Indexed Lookup Eager algorithm, we put all keyword lists in a single B+ tree where keywords are the primary key and Dewey numbers are the secondary key (See Figure 5 where each block can store up to four entries). No data values are associated with keys since the keys contain both keywords and Dewey numbers. Given a keyword w and a Dewey number p, it takes a single range scan operation [11] to find the right and left match of p in the keyword list of w. Since B+ tree implementations usually buffer top level nodes of the B+ tree in memory, we assume the number of disk accesses for finding a match in a keyword list does not include the accesses to the non-leaf nodes in the B+ tree and is $O(1)$.

The number of disk accesses of the Indexed Lookup Eager is $O(k|S_1|)$ because for each node x_1 in S_1 the IL algorithm needs to find a left and a right match in each one of the other $k-1$ keyword lists. Notice that the number of disk accesses of IL cannot be more than $\sum_{i=1}^{k} B_i$ where B_i is the number of blocks of the keyword list S_i. This is because the IL algorithm accesses all keyword lists strictly in order.

In the implementation of the Scan Eager algorithm and the $Stack$ algorithm, the keys in the B+ tree are simply keywords. The data associated with each key w is the list of Dewey numbers of the nodes directly containing the keyword w (See Figure 4). All keyword lists are clustered. The number of disk accesses of Scan Eager or Stack is $O(\sum_{i=1}^{k} B_i)$ where $B_i = \frac{|S_i|}{N_i}$. N_i is the average number of nodes in a disk block of S_i and $|S_i|$ is the number of nodes in the keyword list S_i. N_i depends on the page size W, the depth of the XML tree d, and the maximum out-degrees of nodes at each level. N_i is at least $\frac{W}{\lceil (\sum_{i=1}^{d} \lceil \log_2 o_i \rceil)/8 \rceil}$ where o_i is the maximum out-degree of nodes at level i. In our experiments, using the DBLP dataset, N_i on average is around 200.

The full size keyword lists are not needed to compute SLCAs according to the following property
$slca(S_1, \ldots, S_k) = slca(core(S_1), \ldots, core(S_k))$
where $core(S) = removeAncestor(S)$. $core(S_i)$ is called the core-keyword list of the keyword w_i. To turn a keyword list S into a core-keyword list, the brute-force algorithm compares each node to every other node.

Given any two nodes v_1, v_2 such that $pre(v_1) < pre(v_2)$, if v_1 is not an ancestor of v_2, then for any v such that $pre(v) > pre(v_2)$, v_1 cannot be an ancestor of v. IndexBuilder (Figure 6) uses an algorithm that produces all core-keyword lists in one pass of parsing an input XML document based on the above fact. The description of the algorithm is omitted to save space.

5. THE ALL LOWEST COMMON ANCESTOR PROBLEM (ALCA)

We can use the Indexed Lookup Eager algorithm to derive an efficient algorithm to find all LCAs, that is, LCAs for each combination of nodes in S_1 through S_k. Because an LCA is either an ancestor of a SLCA or is a SLCA itself, we can find all LCAs by walking up in the tree beginning from SLCAs. We solve the ALCA problem by first finding the list L of all SLCAs and then for each

[4]Of course the Dewey number of the node $0.2.0.0.0$ would be changed accordingly.

[5]We could have $LT(1) = 0$. However, the root is conveniently represented by 0.

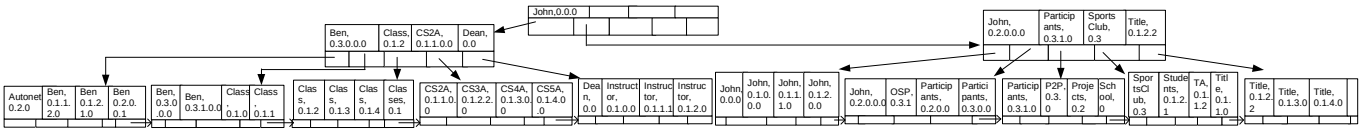


Figure 5: B+ tree from the data of Figure 1 for Indexed Lookup Eager Algorithm

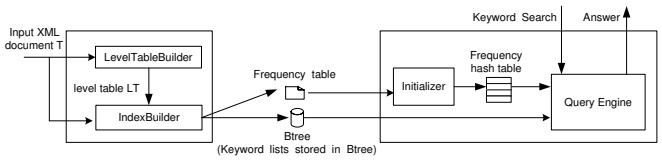


Figure 6: XKSearch Architecture

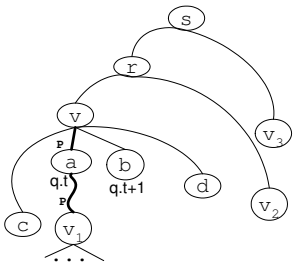


Figure 7: Finding all LCAs

ancestor v of each node v_1 in L check whether v is an LCA, as explained next.

Let v_1 be a SLCA. Consider any node v that is an ancestor of v_1 (See Figure 7). If the subtree rooted at v contains a node v' with a keyword, say w_1, that is not under node v_1 and is not an ancestor of v_1, then v is an LCA. To determine whether v contains such a node v' we use at most two lookups. The nodes under v but not under v_1 are divided into two parts by the path P from v to v_1. Let node c be the right match node of v in S_1 (the keyword list of w_1). If node c is not under v_1 and v_1 is not under c, then c is in the left part of P (otherwise c would be under v_1 because v_1 is a SLCA) which means v is an LCA. Next, let a be the child of v on the path from v to v_1. If the Dewey id of v is q then the Dewey id of a is $q.t$, where t is the ordinal number of a among its siblings. The node b, which is the immediate right sibling of a, has Dewey number $q.(t+1)$ and is called the uncle node of v_1 under v. Let d be the right match node of b in S_1. If $v \prec d$, then d is in the right side of P, which makes v an LCA. The existence of any node containing a keyword from $w_2, \ldots, w_k$ under v but not under v_1 can be checked similarly. The subroutine $checkLCA$ in Algorithm 3 is based on the above observations.

Since a node v might be an ancestor node of multiple SLCAs, we want to avoid repeatedly checking whether v is an LCA. Instead of maintaining some data structures to record whether a node has been identified as an LCA or not, we use an approach that only needs to keep three nodes in memory. Let v_1, v_2, v_3 be the first three nodes (see Figure 7) in the list L of SLCAs produced by the Eager algorithm, and let $r = lca(v_1, v_2)$. For each node v in the path from v_1 to r, we check whether v is an LCA or not. Then we check each node in the path from v_2 to $s = lca(v_2, v_3)$ [6], and so on. Algorithm 3 is based on this approach and guarantees that each of the ancestor nodes of all SLCAs is checked exactly once. Notice that in Algorithm 3 we do not need to produce all SLCAs first. Algorithm 3 pipelines the delivery of LCAs since Algorithm IL pipelines the delivery of SLCAs.

The number of disk accesses of Algorithm 3 is $O(kd|S_1|)$. The main memory complexity of Algorithm 3 is $O(|S_1|kd^2 \log|S|)$. Finding all SLCAs costs $O(|S_1|kd \log|S|)$. Checking whether the ancestors of the $SLCAs$ are LCAs or not costs $O(|S_1|kd^2 \log|S|)$ since we need to check $O(|S_1|d)$ nodes and checking each node costs $O(dk \log|S|)$.

ALGORITHM 3 (COMPUTING ALL LCAS).

```
findLCA(List L) {      //L is the list of SLCAs
  v2 = v1 = removeHead(L);
  while L has more nodes {
    v1 = v2;
    v2 = removeHead(L);
    current-lca=lca(v1,v2);
    for each ancestor node v of v1 until current-lca
    //not including current-lca
      if (checkLCA(v,v1)==true) output v.
  } }
boolean checkLCA( v, v1) {
  for i = 1 to k {
    x= rm(v, Si)
    if ( x ⋠ v1 && v1 ⋠ x ) return true; }
  for i = 1 to k {
    u= the uncle node of v1 under v;
    y = rm(u, Si);
    if (v ≺ y) return true; }
  return false;
}
```

6. EXPERIMENTS

We have run XKSearch on the DBLP data[7]. We filter out citation and other information only related to the DBLP website and group first by journal/conference names, then by years. The experiments have been done on a 1.2GHz laptop with 512MB of RAM. An online demo, which enables keyword search in the same grouped 83MB DBLP data used in the experiments is provided at `http://www.db.ucsd.edu/projects/xksearch`.

The demo runs as a Java Servlet using Apache Jakarta Tomcat server. The Xalan engine is used to translate XML results to HTML.

We evaluate the Scan Eager, Indexed Lookup Eager and Stack algorithms discussed in Section 4 for the SLCA semantics by varying the number and frequencies of keywords both on hot cache (Figures 8, 9 and 10) and on cold cache (Figures 11, 12 and 13).

A program randomly chose forty queries for each experiment. The response time of each experiment on hot cache in Figures 8, 9

[6] s could be a descendent of r.

[7] `http://www.informatik.uni-trier.de/~ley/db`

	# of disk accesses	main memory operations			main memory complexity
		$\#lca$ operations	$\#descendant$ operations	# Dewey number comparisons	
IL	$O(k\|S_1\|)$	$O(k\|S_1\|)$	$O(k\|S_1\|)$	$O(k\|S_1\| \log \|S\|)$	$O(kd\|S_1\| \log \|S\|)$
Scan	$O(T)$	$O(k\|S_1\|)$	$O(k\|S_1\|)$	$O(k\|S\|)$	$O(kd\|S\|)$
Stack	$O(T)$	$O(k\|S\|)$	–	$O(k\|S\|)$	$O(kd\|S\|)$

Table 1: Complexity Analysis for Indexed Lookup Eager, Scan Eager and Stack where $|S_1|$ ($|S|$) is the minimum (maximum) size of keyword lists S_1 through S_k, T is the total number of blocks of all keyword lists on disk and d is the maximum depth of the tree.

and 10 is the average of the corresponding forty queries after five executions. The response time of each experiment on cold cache in Figures 11, 12 and 13 is the average of the corresponding forty queries each of which was run just once after a machine reboot.

In Figure 8 each query q contains two keywords. The smaller frequency is shown in the caption while the bigger frequency is variable. For example, each query of Figure 8(c) in the "Frequency of the large list=1000" category contains two keywords where one of them has frequency of 100 while the other keyword has frequency of 1000. As can be seen from Figure 8, the performance of the Scan Eager and Stack algorithms degrades linearly when the size of the large keyword list increases, while the run time for IL algorithm is essentially constant and its performance is often several orders of magnitude better than Scan Eager and Stack. In all experiments Scan Eager performs a little better than Stack for the reasons explained in Section 3.3.

In Figure 9, each query q contains a keyword of "small" frequency (10 in Figure 9(a), 100 in Figure 9(b), 1000 in Figure 9(c), 10000 in Figure 9(d)) and all other keywords of q have frequency of 100000. For example, each query of Figure 9(b) in the $\#Keywords = 5$ category contains five keywords where one of them has frequency of 100 and the other four have frequency of 100000. The performance of the Scan Eager and the Stack algorithms is essentially independent of $|S_1|$.

Keywords in Figure 10 have the frequencies shown in the caption. The Scan Eager and the Stack algorithms perform a little better than the Indexed Lookup Eager algorithm in most experiments since the Indexed Lookup Eager performs best when the frequencies of keyword lists vary greatly, while all keyword lists in Figure 10 have the same size and the cost of index lookups is more likely greater than the cost of a single scan.

We repeated the experiments in Figures 8, 9 and 10 with cold cache and the results are reported in Figures 11, 12 and 13 respectively. We see similar relationships among the Scan Eager, Indexed Lookup Eager and Stack algorithms. However the differences between the performance of algorithms is not as significant as those in the hot cache experiments. The reason is that most keyword lists do not take many pages. Hence making a random access on the list is effectively equivalent to fetching the complete list. Notice that disk access time dominates any main memory cost as can be seen from the significant response time increases from the hot cache experiments to the cold cache experiments.

We implemented XKSearchB that stores Dewey numbers without using a level table as discussed in Section 4. Experiments show that the size of the keyword lists and the time to construct them are proportional to the size of the input XML documents. On average, the size of indexes constructed by XKSearch is 65% of XKSearchB; the construction time of XKSearch is 55% of XKSearchB; the query response time of XKSearch for hot cache is 70% of XKSearchB for the queries in Figures 8, 9 and 10.

7. RELATED WORK

LCA computation and proximity search are the two areas most related to this work. Computing the LCA of two nodes has been extensively studied and constant time algorithms are known for main memory data structures [20, 26]. These algorithms were designed without the concern of minimizing disk accesses. For example, to compute the LCA of two nodes in [20], two lookups may be needed even after we adapt data structures for disk access minimization. Computing the LCA of nodes using the Dewey numbers does not need any disk access, which is the reason we use Dewey numbers.

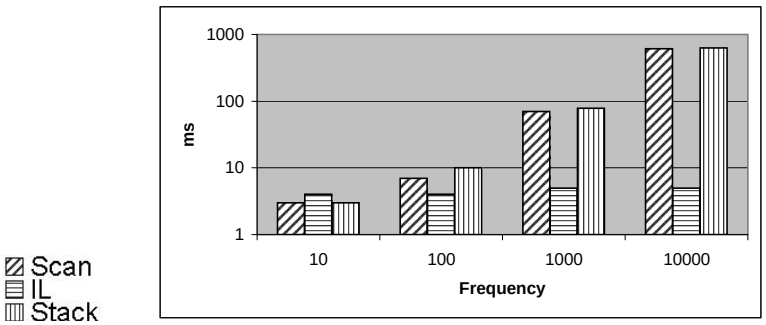
Works on integrating keyword search into XML query languages [9, 10, 21, 24, 25] augment a structured query language with keyword search primitive operators. [21] proposes a *meet* operator, which operates on multiple sets where all nodes in the same set are required to have the same schema, which is a special case of the all LCAs problem. The meet operator of two nodes v_1 and v_2 is implemented efficiently using joins on relations, where the number of joins is the number of edges on the path from v_1 to their LCA. This technique is good for relational database implementations. The XXL search engine [25] extends an SQL-like syntax with ranking and ontological knowledge for similarity metrics.

In BANKS [5] and Proximity Search [12], a database is viewed as a graph of objects with edges representing relationships between the objects. Proximity Search [12] enables proximity searches by a pair of queries. One example is "$Find$ Movie $Near$ Travolta Cage". BANKS uses heuristics to approximate the Steiner tree problem. Discover [15], DBXplorer [1] and XKeyword [16] perform keyword search on relational databases, modeled as graphs. XKSearch delivers much higher efficiency than the above systems, which perform keyword search on arbitrary graphs, by being tuned for SLCA keyword search on trees.

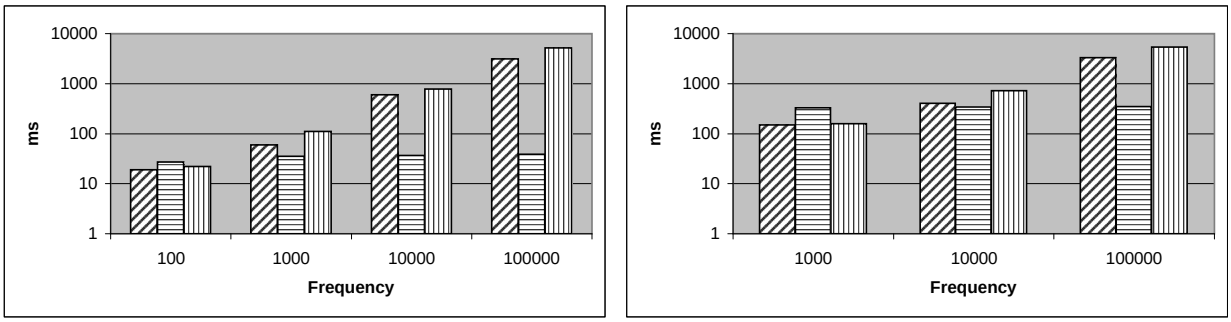
XRANK[13] extends Web-like keyword search to XML. Results are ranked by a Page-Rank [6] hyperlink metric extended to XML. The ranking techniques are orthogonal to the retrieval and hence can easily be incorporated in our work. The keyword search algorithm in XRANK that is relevant to our problem is adapted as the Stack algorithm in the paper, which we have described and compared with the Indexed Lookup Eager and Scan Eager algorithms. Notice that XRANK's query result $xrank(S_1, \ldots, S_k)$ has the following relationship to the semantics used in the paper, $slca(S_1, \ldots, S_k) \subseteq xrank(S_1, \ldots, S_k) \subseteq lca(S_1, \ldots, S_k)$. A recent work XSEarch [8] supports extended keyword search in XML documents and focuses on the semantics and the ranking of the results. It extends information-retrieval techniques (`tfidf`) to rank the results.

[14] provides an optimized version of the LCA-finding stack algorithm. Most important, the algorithm of [14] returns the set of LCAs along with efficiently (for performance and presentation purposes) summarized explanations on why each node is an LCA.

[18] proposes a simple, novel search technique called Schema-Free XQuery to enable users to query an XML document using XQuery without requiring full knowledge of the document schema. In particular, [18] introduces a function named *mlcas* to XQuery

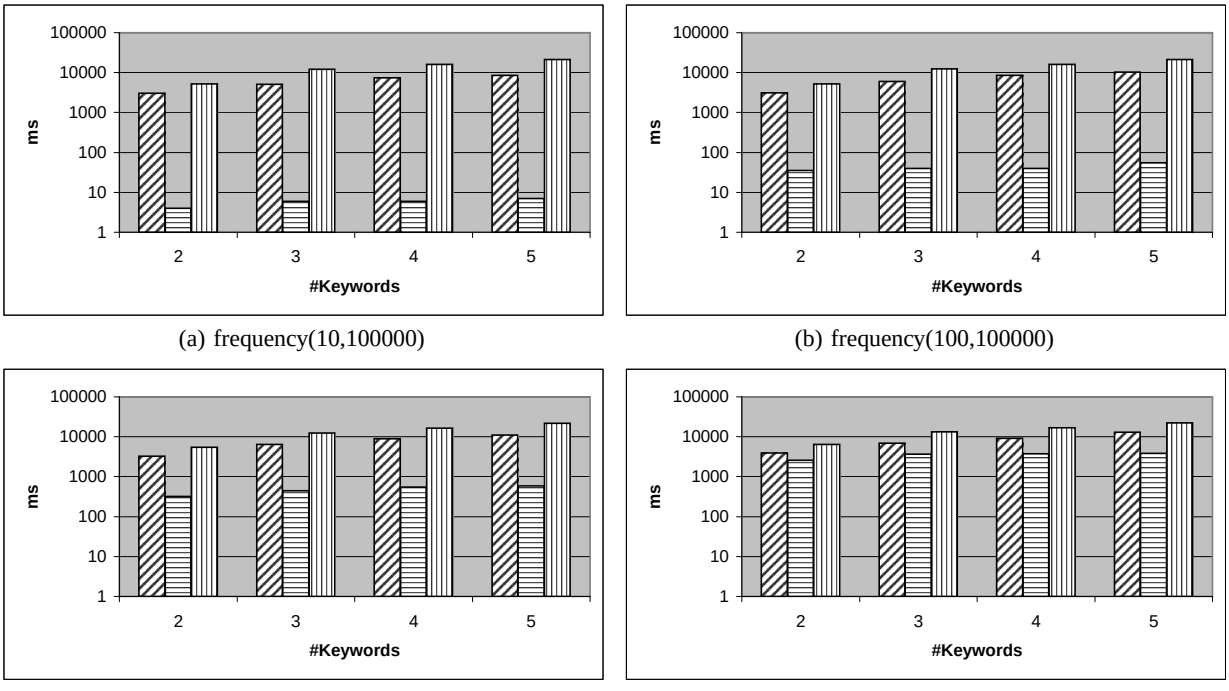


(a) Legend (b) small frequency=10



(c) small frequency=100 (d) small frequency=1000

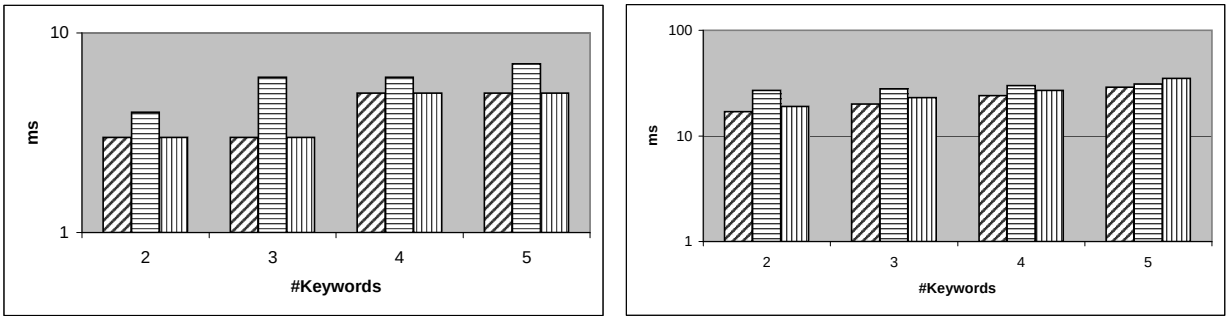
Figure 8: #Keywords=2, keeping the small frequency constant, varying the frequency of the large keyword list (hot cache)



(a) frequency(10,100000) (b) frequency(100,100000)

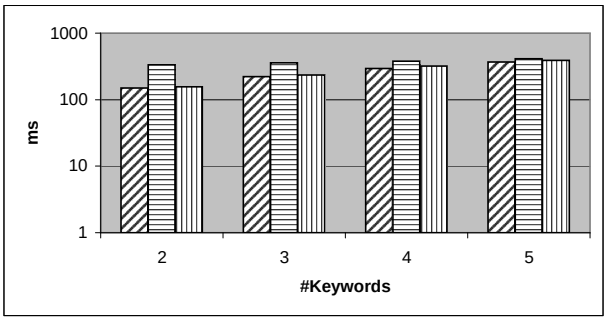
(c) frequency(1000,100000) (d) frequency(10000,100000)

Figure 9: Varying the number of keywords, keeping frequencies constant (hot cache)

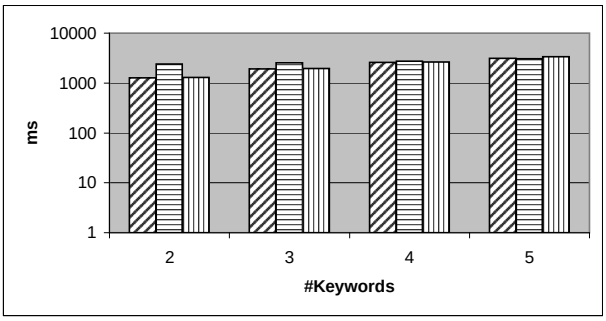


(a) small frequency(10)

(b) medium frequency(100)

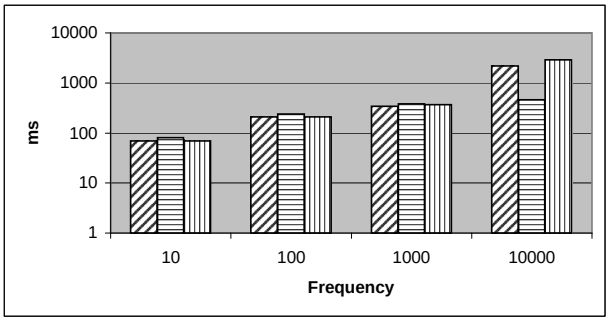


(c) high frequency(1000)

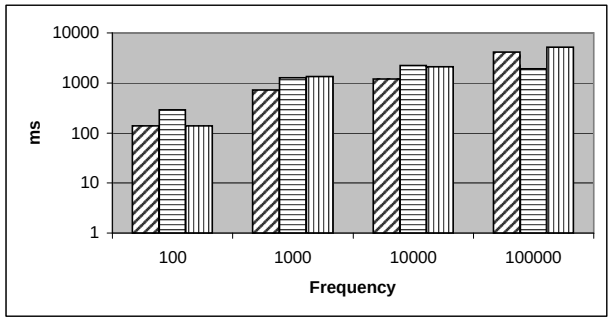


(d) large frequency(10000)

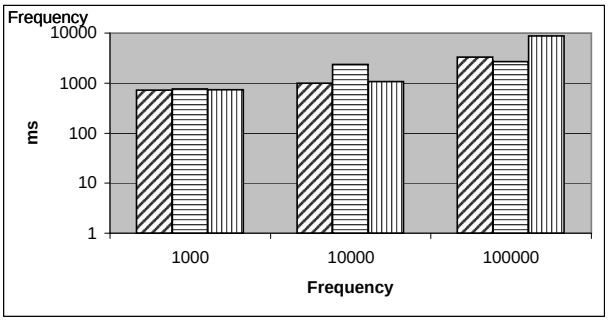
Figure 10: Varying the number of keywords, all keyword lists having the same size (hot cache)



(a) small frequency=10

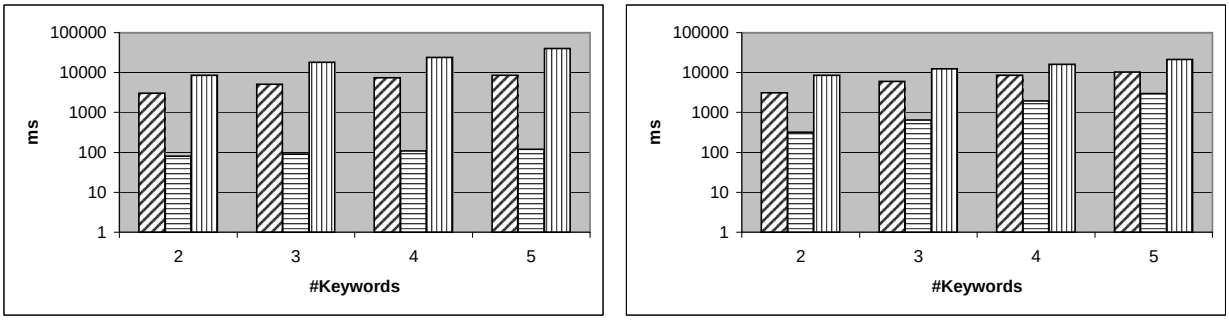


(b) small frequency=100



(c) small frequency=1000

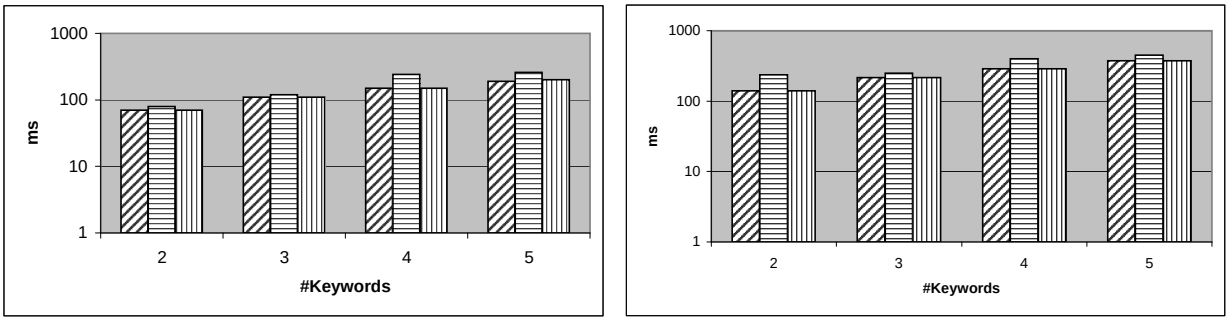
Figure 11: #Keywords=2, keeping the small frequency constant, varying the frequency of the large keyword list (cold cache)



(a) frequency(10,100000) (b) frequency(100,100000)

(c) frequency(1000,100000) (d) frequency(10000,100000)

Figure 12: Varying the number of keywords, keeping frequencies constant (cold cache)



(a) small frequency(10) (b) medium frequency(100)

(c) high frequency(1000) (d) large frequency(10000)

Figure 13: Varying the number of keywords, all keyword lists having the same size (cold cache)

based on the concept of *MLCA* (Meaningful Lowest Common Ancestor) where the set of MLCAs of k sets $S_1, \ldots, S_k$ is the same as $slca(S_1, \ldots, S_k)$. The complexity of the stack based algorithm proposed in [18] to compute the set of MLCAs of $S_1, \ldots, S_k$, which is similar to the sort merge stack algorithm in XRANK, is $O(kd|S|)$, the same as that of the Scan and Stack algorithms. Similar to [14], [18] returns the set of MLCAs with explanations on why each node is an MLCA.

A popular numbering scheme is to use a pair of numbers consisting of preorder and postorder numbers [2, 17]. Given two nodes and their pairs of numbers, it can be determined whether one of them is an ancestor of the other in constant time. However, this type of scheme is inefficient in finding the LCA of two nodes.

Tree query patterns for querying XML trees have received great attention and efficient approaches are known [3, 7, 22]. These approaches are not applicable for the keyword searching problem because given a list of keywords, the number of tree patterns from the keywords is exponential in the size of the schema of the input document and the number of keywords.

Finally there are research prototypes and commercial products that allow keyword searches on a collection of XML documents and return a list of (ranked) XML documents that contain the keywords [19, 27].

8. CONCLUSIONS

The XKSearch system inputs a list of keywords and returns the set of Smallest Lowest Common Ancestor nodes, i.e., the list of nodes that are roots of trees that contain the keywords and contain no node that is also the root of a tree that contains the keywords.

For each keyword the system maintains a list of nodes that contain the keyword, in the form of a tree sorted by the id's of the nodes. The key property of SLCA search 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 v 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 and leads to the Indexed Lookup Eager algorithm, whose main memory complexity is $O(|S_1|kd\log|S|)$ where d is the maximum depth of the tree, k is the number of keywords in the query, and $|S_1|$ ($|S|$) is the minimum (maximum) size of keyword lists S_1 through S_k. Assuming a B-tree disk-based structure, where the non-leaf nodes of the B-tree are cached in main memory the number of disk accesses needed is $O(k|S_1|)$.

The analytical results, as well as the experimental evaluation, show that the Indexed Lookup Eager algorithm outperforms, often by orders of magnitude, other algorithms when the keywords have different frequencies. We provide the Scan Eager algorithm as the best variant for the case where the keywords have similar frequencies. The experimental evaluation compares the Indexed Lookup Eager, Scan Eager and Stack (described in [13]) algorithms.

The XKSearch system is implemented, using the BerkeleyDB [4] B-tree indices and a demo of it on DBLP data is available at `http://www.db.ucsd.edu/projects/xksearch`.

9. REFERENCES

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

[2] V. Aguilera et al. Querying XML documents in XYleme. In *SIGIR Workshop on XML and Information Retrieval*, 2000.

[3] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. In *EDBT*, 2002.

[4] BerkeleyDB. http://www.sleepycat.com/.

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

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[7] Z. Chen, H. Jagadish, F. Korn, and N. Koudas. Counting twig matches in a tree. In *ICDE*, 2001.

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

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

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

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

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

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

[14] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword Proximity Search in XML Trees. Available at http://www.db.ucsd.edu/publications/treeproximity.pdf.

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

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

[17] Q. Li and B. Moon. Indexing and Querying XML data for regular path expressions. In *VLDB*, 2001.

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

[19] J. Naughton et al. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24(2):27–33, 2001.

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

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

[22] D. Srivastava et al. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[23] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, 2002.

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

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

[26] Z. Wen. New algorithms for the LCA problem and the binary tree reconstruction problem. *Information Processing. Lett,51(1): 11-16*, 1994.

[27] XYZFind. http://www.searchtools.com/tools/xyzfind.html.