Multiway SLCA-based Keyword Search in XML Data

Chong Sun
School of Computing
National University of Singapore
sunchong@soe.ucsc.edu

Chee-Yong Chan
School of Computing
National University of Singapore
chancy@comp.nus.edu.sg

Amit K. Goenka
School of Computing
National University of Singapore
amitkuma@comp.nus.edu.sg

ABSTRACT

Keyword search for smallest lowest common ancestors (SLCAs) in XML data has recently been proposed as a meaningful way to identify interesting data nodes in XML data where their subtrees contain an input set of keywords. In this paper, we generalize this useful search paradigm to support keyword search beyond the traditional AND semantics to include both AND and OR boolean operators as well. We first analyze properties of the LCA computation and propose improved algorithms to solve the traditional keyword search problem (with only AND semantics). We then extend our approach to handle general keyword search involving combinations of AND and OR boolean operators. The effectiveness of our new algorithms is demonstrated with a comprehensive experimental performance study.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing, texture databases*

General Terms

Algorithms

Keywords

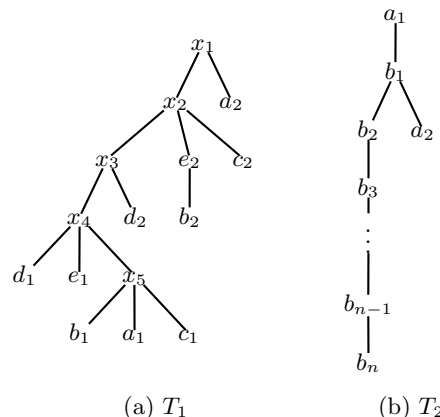keyword search query, smallest lowest common ancestor, XML

1. INTRODUCTION

Keyword search is a convenient and widely-used approach to retrieve information from both unstructured and structured data [1, 4, 7, 10, 11]. Its appeal stems from the fact that keyword queries can be easily posed without requiring to use a query language and knowing the schema or structure of the data being searched. For XML data, where the data is viewed as a hierarchically-structured rooted tree, a natural keyword search semantics is to return all the nodes in XML tree that contain all the keywords in their subtrees. However, this simple search semantics can result in returning too many data nodes, many of which are only remotely linked to the nodes containing the keywords.

A recent direction to improve the effectiveness of keyword search in XML data is based on the notion of *smallest lowest*

Figure 1: **Example XML Trees** T_1 **and** T_2

common ancestor (SLCA) semantics [14]. A keyword search using the SLCA semantics returns nodes in the XML data that satisfy the following two conditions: (1) the subtrees rooted at the nodes contain all the keywords, and (2) the nodes do not have any proper descendant node that satisfies condition (1). The set of returned data nodes are referred to as the *SLCAs* of the keyword search query. Another recent work on keyword search based on the *meaningful LCA (MLCA)* semantics also shares the similar principle as SLCA [12].

The following example illustrates the difference between the SLCA-based keyword search and the conventional LCA-based keyword search.

Example 1.1 Consider the XML tree T_1 shown in Figure 1(a), where the keyword nodes are annotated with subscripts for ease of reference. Consider a keyword search using the keywords $\{a, b, c, d, e\}$ on T_1. If the search is based on the conventional LCA semantics, then the result is given by $\{x_1, x_2, x_3, x_4\}$ as x_1 is the LCA of $\{a_2, b_2, c_2, d_2, e_2\}$, x_2 is the LCA of $\{a_1, b_1, c_2, d_1, e_1\}$, x_3 is the LCA of $\{a_1, b_1, c_1, d_2, e_1\}$, and x_4 is the LCA of $\{a_1, b_1, c_1, d_1, e_1\}$. However, if the search is based on the SLCA semantics, then the result is given by $\{x_4\}$. Observe that each of x_1, x_2, and x_3 is not a SLCA because it has a descendant node x_4 that is a SLCA. □

The state-of-the-art algorithms for keyword search using SLCA semantics are the *Scan Eager (SE)* and *Indexed Lookup Eager (ILE)* algorithms [14], which were shown to be more efficient than stack-based algorithms [8, 12]. The

ILE algorithm is the algorithm of choice when the keyword search involves at least one low frequency keyword, while the SE algorithm performs better when the frequencies of the keywords in the query do not vary significantly. We classify both these algorithms as *binary-SLCA approach (BS)* as they are both based on the same principle of computing the SLCAs for a query with k keywords in terms of a sequence of $k-1$ intermediate SLCA computations, where each SLCA computation takes a pair of data node lists as inputs and outputs another data node list. Specifically, consider a search query with k keywords $w_1, \cdots, w_k$. Let S_i denote the list of XML data nodes that are labeled with keyword k_i, $i \in [1, k]$; and let L_i denote the SLCAs for a query with the first i keywords, $i \in [1, k]$. The binary-SLCA algorithms compute the SLCAs for $w_1, \cdots, w_k$ by computing the sequence $L_2, L_3, \cdots, L_k$, where each L_i is computed by finding the SLCAs of L_{i-1} and S_i (with $L_1 = S_1$). An important observation exploited in the binary-SLCA algorithms is that the result size is bounded by $\min\{|S_1|, \cdots, |S_k|\}$; therefore, by choosing the keyword with the lowest frequency as k_1 (i.e., $|S_1| \leq |S_i|$ for $i \in [1, k]$), the algorithms can guarantee that each $|L_i| \leq |S_1|$, for $i \in [2, k]$.
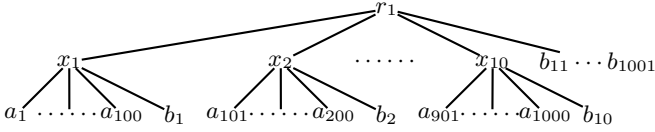

Figure 2: Example XML Tree T_3

However, a drawback of the binary-SLCA approach is that by computing the SLCAs in terms of a series of intermediate SLCA computations, it can often incur many unnecessary SLCA intermediate computations even when the result size is small as the following example illustrates.

Example 1.2 Consider the XML tree T_3 in Figure 2. The SLCAs for the keywords $\{a, b\}$ in T_3 are $\{x_1, x_2, \cdots, x_{10}\}$. Since $|S_a| < |S_b|$, the BS approach will enumerate each of the "a" nodes in S_a to compute a potential SLCA with it. Clearly, this approach results in many redundant computations; for example, the SLCA of a_i and b_1 gives the same result x_1 for $i \in [1, 100]$. In fact, the BS approach will incur a total of 1000 SLCA computations to produce a result of size 10. □

Our first contribution in this paper (Section 3) is the proposal of a novel approach for processing SLCA-based keyword search queries called *multiway-SLCA approach (MS)*. In contrast to the BS approach, our MS approach computes each potential SLCA by taking one data node from each keyword list S_i in a single step instead of breaking the SLCA computation into a series of intermediate binary SLCA computations. Conceptually, each potential SLCA computed by the BS approach can be thought of as being driven by some node from S_1 (i.e., the keyword list with the lowest frequency); on the other hand, our MS approach picks an "anchor" node from among the k keyword data lists to drive the multiway SLCA computation. By doing so, our approach is able to optimize the selection of the anchor node (not necessarily from S_1) to maximize the skipping of redundant

computations. The following example provides an idea of the skipping optimization of our MS approach.

Example 1.3 Consider the processing of the SLCA-based keyword search with keywords $\{a, b\}$ on T_3 in Figure 2 using our MS approach. MS will first consider the first data nodes in all the keyword lists and selects the node that occurs the latest in T_3 as the anchor node. Thus, between $a_1 \in S_a$ and $b_1 \in S_b$, b_1 will be selected as the anchor node (the property behind this optimization will be explained later in the paper). Next, using b_1 as anchor, our approach will select the closest data nodes from the other keyword lists to compute a potential SLCA. Thus, the first SLCA is computed for the set $\{b_1, a_{100}\}$. After the first potential SLCA is computed, our approach will consider the first nodes in the keyword lists that occur after b_1 for the next computation (i.e., nodes a_{101} and b_2). The next anchor node selected is b_2, and the next SLCA computation involves b_2 and a_{200}. Clearly, the MS approach is able to skip many unnecessary computations. □

In addition to introducing the notion of anchor nodes that we alluded to for minimizing redundant computations, we also develop several optimizations to further maximize the skipping of data nodes in the keyword list without compromising correctnesses of query results.

Our second contribution in this paper (Section 4) is the generalization of the SLCA-based approach to handle more general keyword search queries beyond the implicit AND-semantics to support any combinations of AND and OR semantics. This enables more flexible and expressive keyword search queries such as "(a OR b) AND c AND (d or E)" to be specified. We extend our MS approach to evaluate general keyword search queries involving both AND and OR operators.

Finally, our third contribution in this paper (Section 5) is a comprehensive experimental performance evaluation which demonstrates that our proposed multiway-SLCA approach outperforms the previous binary-SLCA approach for both traditional keyword search queries as well as generalized keyword search queries.

2. PRELIMINARIES

Let $K = \{w_1, \cdots, w_k\}$ denote an input set of k keywords, where each keyword w_i is associated with a set S_i of nodes in an XML document T (sorted in document order[1]). A set of nodes $S = \{v_1, \cdots, v_k\}$ is defined to be a *match* for K if $|S| = |K|$ and each $v_i \in S_i$ for $i \in [1, k]$. We use S_i to denote the data node list (sorted in document order) associated with the keyword w_i. For simplicity and without loss of generality, we assume w_1 to be the lowest frequency keyword among the keywords in K; i.e., $|S_1| \leq |S_i|$ for $i \in [1, k]$.

A node v in T is a *lowest common ancestor* (or LCA) for K if v is the lowest common ancestor node of some match S. Moreover, v is also a *smallest lowest common ancestor* (or SLCA) for K if each descendant of v in T is not a LCA for K.

Given two nodes v and w in a document tree T, $v \prec_p w$ denotes that v precedes w (or w succeeds v) in document order in T; and $v \preceq_p w$ denotes that $v \prec_p w$ or $v = w$. More generally, given two matches $V = \{v_1, \cdots, v_k\}$ and

---

[1]Document order corresponds to a preorder traversal of document nodes.

$W = \{w_1, \cdots, w_k\}$, where $v_i, w_i \in S_i$, $\forall\, i \in [1, k]$, we say that V *precedes* W (or W *succeeds* V), denoted by $V \prec_p W$, if they satisfy both the following properties: (1) $v_i \preceq_p w_i$ for each $i \in [1, k]$; and (2) $V \neq W$.

We use $v \prec_a w$ to denote that v is a proper ancestor of w in T, and $v \preceq_a w$ to denote that $v = w$ or $v \prec_a w$.

Consider a node v and a set of nodes S. The function $first(S)$ returns the "first" node $v' \in S$ such that $v' \preceq_p v_i$ for each $v_i \in S$. Similarly, the function $last(S)$ returns the "last" node $v' \in S$ such that $v_i \preceq_p v'$ for each $v_i \in S$. Both functions return null if any of its input argument values is null.

The function $out(v, S)$ returns the "first" node $v' \in S$ such that $v \prec_p v'$ and v' is not a descendant of v or equal to v; i.e., $out(v, S) = first(\{v' \in S \mid v \prec_p v',\ v \npreceq_a v'\})$. The function returns null if no such node exists or if v is null.

The function $next(v, S)$ returns the first node in S that succeeds v if it exists; otherwise, it returns null. The function $pred(v, S)$ returns the *predecessor* of v in S, that is, the last node in S that precedes v if it exists; otherwise, it returns null.

The function $closest(v, S)$ computes the *closest node* in S to v as follows:

$$closest(v, S) = \begin{cases} pred(v, S) & \text{if } lca(v, next(v, S)) \prec_a \\ & \quad lca(v, pred(v, S)), \\ next(v, S) & \text{otherwise.} \end{cases}$$

However, $closest(v, S)$ returns null if both $pred(v, S)$ and $next(v, S)$ are null; and it returns the non-null value if exactly one of $pred(v, S)$ and $next(v, S)$ is null.

The function $lca(S)$ computes the *lowest common ancestor* (or LCA) of the set of nodes S and returns null if any of its arguments is null.

For notational convenience, we assume that the root node of the data tree T has a virtual parent node, denoted by d_{root}, such that d_{root} is a proper ancestor node of every node in T.

The following example illustrates our definitions.

Example 2.1 Consider the XML document tree T_1 shown in Figure 1(a) and the keyword search query $K = \{a, b, c\}$. Note that $\{a_1, b_1, c_2\}$ is a match for K; but neither $\{a_1, c_2\}$ nor $\{x_1, b_2, c_1\}$ is a match for K. We have $e_1 \prec_p b_2$ but $e_2 \nprec_p x_4$. Moreover, $\{a_1, b_1, c_1\} \prec_p \{a_2, b_2, c_2\}$. We have $x_2 \preceq_a e_1$ and $x_3 \npreceq_a c_2$. If $S = \{d_1, c_1, d_2\}$, then $first(S) = d_1$, $last(S) = d_2$, $out(x_4, S) = d_2$, $next(x_4, S) = c_1$, $next(c_1, S) = d_2$, $next(e_2, S) = null$, $pred(a_1, S) = d_1$, $pred(x_4, S) = null$, $closest(b_1, S) = c_1$, $closest(a_2, S) = d_2$, and $lca(S) = x_3$. □

3. OUR APPROACH

In this section, we present our new approach of processing SLCA-based keyword search queries called the *multiway-SLCA approach (MS)*.

As alluded in the introduction, the key motivation behind our MS approach is to avoid the unnecessary overhead of the BS approach where SLCAs are computed in terms of intermediate SLCA computations by enumerating each data node in the lowest-frequency keyword list. As Example 1.2 demonstrates, by rigidly driving the SLCA computations from the lowest-frequency keyword list can result in many redundant computations particularly when the size of the results is small.

We first introduce the notion of *anchor nodes* in Sec-

tion 3.1 which is a central idea in our MS approach. Section 3.2 then presents several important properties about anchored matches. We present our first MS-based algorithm called *basic multiway-SLCA (BMS)* in Section 3.3, followed by our second improved MS-based algorithm, called *incremental multiway-SLCA (IMS)*, in Section 3.4.

3.1 Anchor Nodes

A match $S = \{v_1, \cdots, v_k\}$ is said to be *anchored* by a node $v_a \in S$ if for each $v_i \in S - \{v_a\}$, $v_i = closest(v_a, S_i)$. We refer to v_a as the *anchor node* of S.

The concept of anchor nodes is important to our multiway-SLCA approach as it enables us to restrict matches to those that are anchored by some nodes as the following result demonstrates.

LEMMA 3.1. *If $lca(S)$ is an SLCA and $v \in S$, then $lca(S) = lca(S')$, where S' is the set of nodes anchored by v.*

PROOF. The proof is established by contradiction. Suppose that $lca(S) \neq lca(S')$. Since $lca(S)$ is an SLCA and $v \in S' \cap S$, this implies that $lca(S')$ is a proper ancestor of $lca(S)$. It follows that there must exists some S_i such that $lca(S) \npreceq_a closest(v, S_i)$. However, since $S \cap S_i \neq \emptyset$, we have a contradiction. □

Thus, our MS approach only considers anchored sets for computing potential SLCAs. The optimization potential of the above result was illustrated earlier in Example 1.3. Recall that b_1 is the selected anchor node for the SLCA computation with $a_{100} = closest(b_1, S_a)$. By choosing b_1 as the anchor node (instead of using a_1 as in the BS approach), for the first SLCA computation, it follows from Lemma 3.1 that it is unnecessary to compute SLCAs for matches that include any $a_i, i \in [1, 99]$ because such matches would necessarily include b_1 and Lemma 3.1 states that it is not possible to generate new SLCAs with such matches.

Note in our MS approach, an anchor node can be chosen from any of keyword data lists (i.e., $S_1, \cdots, S_k$). For notational convenience, when we denote an anchor node as v_m, we also mean that v_m is selected from S_m, $m \in [1, k]$.

3.2 Properties

In this section, we present several important properties that form the basis of the optimizations in our MS-based algorithms.

The following result states that if the LCAs of two matches S and S' are both distinct SLCAs, then the two matches must necessarily be disjoint.

LEMMA 3.2. *If $lca(S)$ and $lca(S')$ are distinct SLCAs, then $S \cap S' = \emptyset$.*

PROOF. Suppose $lca(S)$ and $lca(S')$ are two distinct SLCAs. If both S and S' contains some common node, then $lca(S)$ and $lca(S')$ must be related in one of three possibilities: (a) $lca(S) = lca(S')$; (b) $lca(S)$ is an ancestor of $lca(S')$; or (c) $lca(S)$ is a descendant of $lca(S')$. Case (a) contradicts the fact that $lca(S)$ and $lca(S')$ are distinct SLCAs. Cases (b) and (c) imply that either $lca(S)$ or $lca(S')$ is not a SLCA, contradicting the fact that $lca(S)$ and $lca(S')$ are two distinct SLCAs. Thus, it follows by contradiction that $S \cap S' = \emptyset$. □

LEMMA 3.3. *Let V and W be two matches such that $V \prec_p W$. If $lca(W)$ is not a descendant of $lca(V)$, then for any*

match X where $W \prec_p X$, $lca(X)$ is also not a descendant of $lca(V)$.

PROOF. Let $V \prec_p W$ and $lca(W)$ is not a descendant of $lca(V)$, then either (a) all the nodes in the subtree rooted at $lca(V)$ precede all the nodes in the subtree rooted at $lca(W)$, or (b) $lca(W)$ is an ancestor of $lca(V)$.

For case (a), if $lca(X)$ is a descendant of $lca(V)$, then all the nodes in the subtree rooted at $lca(X)$ must precede all the nodes in the subtree rooted at $lca(W)$ contradicting that X succeeds W. Thus, $lca(X)$ cannot be a descendant of $lca(V)$ for case (a). For case (b), W must contain some node w such that w is not a descendant of $lca(V)$ and w succeeds V; if not, $lca(W)$ cannot be an ancestor of $lca(V)$. Therefore, if $lca(X)$ is a descendant of $lca(V)$, then this implies that $w \in W$ succeeds X, contradicting the fact that X succeeds W. Thus, $lca(X)$ cannot be a descendant of $lca(V)$ for case (b) as well. □

Lemma 3.3 is a useful generalization of Lemma 2 in [14] that is exploited in our algorithms to determine whether a computed LCA is guaranteed to be a SLCA. Specifically, if $V \prec_p W$ and $lca(W)$ is not a descendant of $lca(V)$, then one can conclude that $lca(W)$ is an SLCA.

LEMMA 3.4. *Consider two matches S and S', where $S \prec_p S'$, and S is anchored by some node v. If S' contains some node u where $u \preceq_p v$, then $lca(S')$ is either equal to $lca(S)$ or an ancestor of $lca(S)$.*

PROOF. Let $v \in S_i$, $i \in [1, k]$. Let $S_i \cap S' = \{v'\}$. Since $S \prec_p S'$, we must have $v \preceq_p v'$. Since $\{u, v'\} \subseteq S'$ and $u \preceq_p v \preceq_p v'$, v must be a descendant of $lca(S')$ which implies that $lca(S')$ is either equal to $lca(S)$, a descendant of $lca(S)$, or an ancestor of $lca(S)$. However, if $lca(S')$ is a descendant of $lca(S)$, it would contradict the fact that S is anchored by v. Therefore, $lca(S')$ is either equal to $lca(S)$ or an ancestor of $lca(S)$. □

Lemma 3.4 provides a useful property to optimize the selection of the next match to be considered. Specifically, if we have considered a match S that is anchored by a node v_a, then we can skip matches that contain any node $v \preceq_p v_a$ in S.

LEMMA 3.5. *Let S and S' be two matches. If S' contains two nodes, where one is a descendant of $lca(S)$, while the other is not, then $lca(S')$ is either equal to $lca(S)$ or an ancestor of $lca(S)$.*

PROOF. Let $v, w \in S'$, where v is a descendant of $lca(S)$, and w is not a descendant of $lca(S)$. Since v is a descendant of both $lca(S)$ and $lca(S')$, either $lca(S)$ and $lca(S')$ are equal or one is an ancestor of the other. However, since w is not a descendant of $lca(S)$, $lca(S')$ cannot be a descendant of $lca(S)$; and the claim follows. □

Lemma 3.5 provides another useful property to optimize the selection of the next match to be considered. Specifically, if we have considered a match S and $lca(S)$ has been confirmed to be an SLCA, then we can skip matches S' that contains some node that is a descendant of $lca(S)$ as well as another node that is a not a descendant of $lca(S)$.

LEMMA 3.6. *Let S be a set of nodes. Then $lca(S) = lca(first(S), last(S))$.*

PROOF. Since $\{first(S), last(S)\} \subseteq S$, therefore, $lca(S)$ is either equal to or an ancestor of $lca(first(S), last(S))$. Clearly, for each $v \in S$ where $first(S) \prec_p v \prec_p last(S)$, v is a descendant of $lca(first(S), last(S))$. It follows that $lca(S) = lca(first(S), last(S))$. □

Lemma 3.6 states that the LCA of a set of nodes S is equivalent to the LCA of the two extreme nodes (i.e., the first and last nodes) in S. This property enables the LCA computation for a set of nodes S to be improved significantly as it suffices to compute the LCA of S in terms of only its first and last nodes.

3.3 Basic Multiway-SLCA Algorithm (BMS)

In this section, we present our first Multiway-SLCA-based algorithm called *Basic Multiway-SLCA (BMS)* for computing SLCAs for a set of keywords $\{w_1, \cdots, w_k\}$. The details are given in Algorithm 1 which takes k keyword data lists $S_1, \cdots, S_k$ as input and returns the SLCAs as a collection of nodes. Each S_i is the list of data nodes associated with keyword w_i.

The algorithm computes the SLCAs iteratively. At each iteration, an anchor node v_m is judiciously selected to compute the match anchored by v_m and its LCA (denoted by α). If α is potentially an SLCA, it is maintained in an intermediate SLCA result list given by $\alpha_1, \cdots, \alpha_n$, $n \geq 1$, where all the LCAs α_i in the list are definite SLCAs except for the most recently computed candidate α_n. To minimize the computation of LCAs that are not SLCAs, it is important to optimize the anchor node selected at each iteration.

Initially, step 2 initializes the first candidate SLCA α_1 to be d_{root} (the virtual root node of data tree); if α_1 remains as d_{root} at the end of the algorithm (step 22), then it means that the SLCA result list is empty. The first anchor node v_m is selected in step 1. Instead of choosing the first node $v_1 \in S_1$ as the anchor (as is done in the BS approach), BMS selects the first node $v_m \in S_m$, $m \in [1, k]$ that is the "furthest" node among all the first nodes in $S_1, \cdots, S_k$. In doing so, all the nodes in S_1 that precede $u_1 = closest(v_m, S_1)$ are skipped from consideration as anchor nodes. The correctness of this optimization stems from the fact that for each $v \in S_1$ that precedes u_1, $closest(v, S_m)$ must be v_m; therefore, by Lemma 3.1, no SLCAs will be missed out by using v_m as the first anchor node.

Steps 4 to 9 further optimize the selection of the anchor node to ensure that the total number of candidate SLCAs computed is no more than $|S_1|$ (elaborated in Section 3.5). Specifically, if the selected anchor node v_m precedes $closest(v_m, S_1)$, then by Lemma 3.1, no SLCAs will be omitted by replacing the anchor node v_m with $closest(v_m, S_1)$. The usefulness of this optimization is illustrated in Example 3.2.

After an anchor node v_m has been chosen, step 10 computes the match anchored by v_m, and step 11 computes the LCA α of this match in terms of only its first and last nodes (based on Lemma 3.6). Steps 12 to 16 check whether the newly computed LCA α can be a candidate SLCA; and if so, whether α can be used to eliminate the previous candidate SLCA α_n. Steps 17 to 20 optimize the selection of the next anchor node by choosing the furthest possible node that maximizes the number of skipped nodes: step 17 is based on Lemma 3.4 while steps 18 to 20 are based on Lemma 3.5.

Example 3.1 Consider computing SLCAs for the set of

Algorithm 1 Basic Multiway-SLCA $(S_1, \cdots, S_k)$
1: let $v_m = last(\{first(S_i) \mid i \in [1, k]\})$, where $v_m \in S_m$
2: initialize $n = 1$; $\alpha_1 = d_{root}$
3: **while** $(v_m \neq$ null$)$ **do**
4:    **if** $(m \neq 1)$ **then**
5:      $v_1 = closest(v_m, S_1)$
6:      **if** $(v_m \prec_p v_1)$ **then**
7:        $v_m = v_1$
8:      **end if**
9:    **end if**
10:   $v_i = closest(v_m, S_i)$ for each $i \in [1, k], i \neq m$
11:   $\alpha = lca(first(v_1, \cdots, v_k), last(v_1, \cdots, v_k))$
12:   **if** $(\alpha_n \preceq_a \alpha)$ **then**
13:     $\alpha_n = \alpha$
14:   **else if** $(\alpha \npreceq_a \alpha_n)$ **then**
15:     $n = n + 1$; $\alpha_n = \alpha$
16:   **end if**
17:   $v_m = last(\{next(v_m, S_i) \mid i \in [1, k], v_i \preceq_p v_m\})$
18:   **if** $(v_m \neq null)$ **and** $(\alpha_n \npreceq_a v_m)$ **then**
19:     $v_m = last(\{v_m\} \cup \{out(\alpha_n, S_i) \mid i \in [1, k], i \neq m\})$
20:   **end if**
21: **end while**
22: **if** $(\alpha_1 = d_{root})$ **then return** $\emptyset$ **else return** $\{\alpha_1, \cdots, \alpha_n\}$

Algorithm 2 Incremental Multiway-SLCA $(S_1, \cdots, S_k)$
1: let $v_m = last(\{first(S_i) \mid i \in [1, k]\})$, where $v_m \in S_m$
2: initialize $n = 1$; $\alpha_1 = d_{root}$
3: **while** $(v_m \neq$ null$)$ **do**
4:    **if** $(m \neq 1)$ **then**
5:      $v_1 = closest(v_m, S_1)$
6:      **if** $(v_m \prec_p v_1)$ **then**
7:        $v_m = v_1$
8:      **end if**
9:    **end if**
10:   $P = \{pred(v_m, S_i) \mid i \in [1, k], i \neq m\} \cup \{v_m\}$
11:   $N = \{next(v_m, S_i) \mid i \in [1, k], \; next(v_m, S_i) \neq null\}$
12:   initialize $r_{max} = last(N)$; $r = v_m$
13:   **repeat**
14:     remove ℓ from P, where $\ell = first(P)$
15:     $\alpha = lca(\ell, r)$
16:     $r = last(r, v)$ where $v \in N$ s.t. $v = next(v_m, S_j), \ell \in S_j$
17:   **until** $(r = null)$ or $(\alpha \npreceq_a r)$ or $(r = r_{max})$
18:   **if** $(r = null)$ or $(\alpha \npreceq_a r)$ **then**
19:     **if** $(\alpha_n \preceq_a \alpha)$ **then**
20:       $\alpha_n = \alpha$
21:     **else if** $(\alpha \npreceq_a \alpha_n)$ **then**
22:       $n = n + 1$; $\alpha_n = \alpha$
23:     **end if**
24:     $v_m = last(r, out(\alpha_n, S_1), \cdots, out(\alpha_n, S_k))$
25:   **else**
26:     $v_m = r$
27:   **end if**
28: **end while**
29: **if** $(\alpha_1 = d_{root})$ **then return** $\emptyset$ **else return** $\{\alpha_1, \cdots, \alpha_n\}$

---

keywords $\{a, b, c, d, e\}$ on the data tree T_1 in Figure 1(a) using the BMS algorithm. Since each keyword has the same frequency, let S_1 be the list of data nodes for keyword "a". The first anchor node selected is c_1, and the first candidate SLCA computed is $\alpha_1 = lca(d_1, e_1, b_1, a_1, c_1) = x_4$. The next anchor node selected is a_2, and the second candidate SLCA computed is $\alpha = lca(d_2, e_2, b_2, c_2, a_2) = x_1$. However, since $x_1 \preceq_a x_4$, x_1 is not a SLCA. The next anchor node selected has a *null* value (due to $next(a_2, S_1) = null$), and the algorithm terminates with x_4 as the only SLCA. $\quad\square$

The next example illustrates the importance of the optimization performed by steps 4 to 9.

Example 3.2 Consider computing SLCAs for the set of keywords $\{a, b\}$ on the datatree T_2 in Figure 1(b). Here, S_1 refers to the list of nodes for keyword "a". Using a non-optimized BMS algorithm that excludes steps 4 to 9, the first anchor node is b_1 and the candidate SLCA computed is $lca(b_1, a_2) = b_1$. Similarly, the subsequent sequence of anchor nodes selected is b_2, b_3, $\cdots$, b_n, and the candidate SLCA computed for each of these anchor nodes b_i is $lca(b_i, a_2) = b_1$. Clearly, the non-optimized BMS incurs many redundant SLCA computations that involve the same a_2 node. In general, the non-optimized BMS performs poorly when many anchor nodes share the same closest node (w.r.t. some keyword) that succeeds the anchor nodes. The BMS algorithm avoids this problem by bounding the number of computed SLCAs to be no more than $|S_1|$ using steps 4 to 9. In this case, the first anchor node selected is optimized to a_2 and the first candidate SLCA computed is $lca(a_2, b_1) = b_1$. The next anchor node selected has a *null* value (since $next(a_2, S_1) = null$) and the algorithm terminates without any redundant SLCA computations. Thus, the number of candidate SLCA computations is reduced from n to just one. $\quad\square$

3.4 Incremental Multiway-SLCA Algorithm (IMS)

In this section, we present our second Multiway-SLCA-based algorithm called *Incremental Multiway-SLCA (IMS)*,

whose details are shown in Algorithm 2. IMS is an optimized variant of BMS that reduces the number of LCA computations.

In BMS (Algorithm 1), each computation of α incurs at least $2k - 1$ LCA computations: each of the $k - 1$ calls to *closest* function in step 10 requires two LCA computations, and step 11 adds another LCA computation.

To avoid the large number of LCA computations incurred by an explicit computation of M, the IMS algorithm determines $first(M)$ and $last(M)$ without actually computing M. In the following, we analyze the properties of $first(M)$ and $last(M)$, and explain how this can be achieved. By definition of the match M anchored by v_m, M must satisfy the following three conditions:

1. $M \subseteq \{v_m\} \cup P \cup N$, where $P = \{pred(v_m, S_i) \mid i \in [1, k], i \neq m, pred(v_m, S_i) \neq null\}$ and $N = \{next(v_m, S_i) \mid i \in [1, k], i \neq m, next(v_m, S_i) \neq null\}$;

2. $M \cap S_i \neq \emptyset \; \forall \; i \in [1, k]$; and

3. $v_m \in M$.

Since M must contain v_m and every node in P precedes v_m, it follows that $first(M) \in P \cup \{v_m\}$. Furthermore, $last(M)$ can be determined once $first(M)$ is known. Let $P' \subseteq P$ denote the subset of nodes in P that precedes $first(M)$ (i.e., $P' = \{v \in P \mid v \prec_p first(M)\}$); and let $N' \subseteq N$ denote the subset of nodes in N that corresponds to P' that succeeds v_m (i.e., $N' = \{next(v_m, S_i) \mid i \in [1, k], \; pred(v_m, S_i) \in P'\}$). Since $P' \cap M = \emptyset$, in order for M to satisfy condition (2), it is necessary that $M \supseteq N'$. Moreover, since $|P'| = |N'|$ and $|M| = k$, we must have $M = (P - P') \cup \{v_m\} \cup N'$ and $last(M) = last(N')$.

Since there are $|P| + 1$ possible values for $first(M)$, let $M_1, M_2, \cdots, M_{|P|+1}$ denote the sequence of matches where for each $i \in [1, |P|+1]$, we have (1) $v_m \in M_i$; (2) $first(M_i) \in$

$P \cup \{v_m\}$; and (3) $first(M_1) \prec_p \ first(M_2) \ \prec_p \cdots \ \prec_p$ $first(M_{|P|+1})$. In other words,

$$first(M_i) \ = \ \begin{cases} first(P \ \cup \ \{v_m\}) & \text{if } i = 1, \\ first((P \ \cup \ \{v_m\})- & \text{otherwise.} \\ \bigcup_{j=1}^{i-1} first(M_j)) \end{cases} \quad (1)$$

Based on the preceding analysis of $first(M)$ and $last(M)$, $last(M_i)$ can be computed incrementally as follows:

$$last(M_i) \ = \ \begin{cases} v_m & \text{if } i = 1, \\ last(M_{i-1} \ \cup & \text{otherwise.} \\ \{next(v_m, S_x)\}) \end{cases} \quad (2)$$

where $first(M_{i-1}) \in S_x$.
    It follows that

$$first(M_1) \prec_p \cdots \prec_p first(M_{|P|+1}) \prec_p$$
$$last(M_1) \preceq_p \cdots \preceq_p last(M_{|P|+1}). \quad (3)$$

    Clearly, $M = M_j$ for some $j \in [1, |P|+1]$, where $lca(M_i) \preceq_a$ $lca(M_j)$ for $i \in [1, |P| + 1]$. We can characterize M_j by the following two properties:

(P1) for $i \in [1, j)$, $lca(M_i) \preceq_a last(M_{i+1})$; and

(P2) if $j < |P| + 1$, then $lca(M_j) \not\preceq_a last(M_{j+1})$.

Property (P1) implies that $lca(M_i) \preceq_a lca(M_{i+1})$ for $i \in [1, j)$. Specifically, since $first(M_i) \prec_p first(M_{i+1}) \prec_p last(M_i)$ (by Equation (3)), it follows that $lca(M_i) \preceq_a first(M_{i+1})$; combining this with property (P1), we have $lca(M_i) \preceq_a lca(M_{i+1})$.
    Property (P2) implies that $lca(M_i) \preceq_a lca(M_j)$ for $i \in (j, |P| + 1]$. To see this, note that $lca(M_j) \preceq_a first(M_i)$ for $i \in (j, |P| + 1]$ (by Equation (3)). Furthermore, since $lca(M_j) \not\preceq_a last(M_{j+1})$ (property (P2)) and $last(M_{j+1}) \preceq_p last(M_i)$ for $i \in (j+1, |P| + 1]$ (by Equation (3)), it follows that $lca(M_j) \not\preceq_a last(M_i)$ for $i \in (j, |P| + 1]$. Thus, for $i \in (j, |P| + 1]$, we have $M_j \prec_p M_i$, $lca(M_j) \preceq_a first(M_i)$ and $lca(M_j) \not\preceq_a last(M_i)$; it follows from Lemma 3.5 that $lca(M_i) \preceq_a lca(M_j)$.
    The IMS algorithm (Algorithm 2) shares many similarities with the BMS algorithm in the previous section. The key difference lies in steps 10 to 17 which determines $lca(M)$ for a match M anchored by a node v_m without actually computing M. The repeat loop enumerates a sequence of matches $M_1, M_2, \cdots$ to compute $first(M)$ and $last(M)$ and hence $lca(M)$. In the ith iteration of the repeat loop (steps 13 to 17), $first(M_i)$ is computed in step 14 as ℓ, and α_i is computed in step 15 (with r representing $last(M_i)$). Step 16 determines $last(M_{i+1})$ for the next iteration. The search for M is terminated when any one of the three conditions in step 17 is met. Firstly, if $r = null$, then it means that $next(v_m, S_j) = null$ and there are no further matches in the data; therefore, $\alpha = lca(M)$ and the next anchor node is correctly set to $null$ by step 24. Secondly, if $\alpha \not\preceq_a r$, then $\alpha = lca(M)$ and Lemma 3.5 is applied to optimize the selection of the next anchor node in step 24. Finally, if $r = last(N)$, then it means that all the matches M_i subsequently enumerated within the repeat loop must have $last(M_i) = last(N)$ as well; therefore, M must correspond to the very last match in the enumeration. To quickly skip to this last match without continuing with the enumeration, step 26 applies Lemma 3.1 to update the next anchor node to be r.

Note that the number of LCA computations incurred by IMS for each candidate SLCA computation is at least one (one iteration of repeat loop) and at most $k + 1$ ($k - 1$ iterations of repeat loop and one call to *closest* function). In contrast, BMS requires between $2k - 1$ and $2k + 1$ LCA computations.

Example 3.3 Consider again computing SLCAs for the set of keywords $\{a, b, c, d, e\}$ on the data tree T_1 in Figure 1(a), where S_1 is associated with keyword "a". Using the IMS algorithm, the first anchor node selected is c_1, $P = \{d_1, e_1, b_1, a_1, c_1\}$, and $N = \{d_2, e_2, b_2, c_2, a_2\}$. In the first iteration of the repeat loop, $\alpha = lca(d_1, c_1) = x_4$. r is then updated to d_2 and the repeat loop terminates since $x_4 \not\preceq_a d_2$. Therefore, the first candidate SLCA computed is $\alpha_1 = x_4$. The next anchor node selected is a_2, $P = \{d_2, e_2, b_2, c_2, a_2\}$, and $N = \{\}$. In the first iteration of the repeat loop, $\alpha = lca(d_2, a_2) = x_1$. r is then updated to $null$ and the repeat loop terminates. Therefore, since $\alpha \preceq_a \alpha_1$, α is definitely not a SLCA. The next anchor node has a $null$ value and the algorithm terminates with x_4 as the only SLCA. The number of LCA computations incurred by the IMS algorithm is only two. In contrast, the BMS algorithm incurs 18 LCA computations (Example 3.1). □

3.5 Analysis

    In this section, we analyze the time complexity of our new algorithms. We begin by establishing an upper bound on the number of candidate SLCAs computed by each of the BMS and IMS algorithms.

    LEMMA 3.7. *The number of candidate SLCAs computed by each of the BMS and IMS algorithms is no more than* $|S_1|$.

    PROOF. We prove the claim for BMS; the proof for IMS follows similarly. The upper bound is established by showing that for any two candidate SLCAs computed by BMS, their corresponding matches do not contain the same node from S_1. By step 1, the first anchor node selected either succeeds or is equal to $first(S_1)$. Let v_m be the anchor node used to compute a candidate SLCA $lca(M)$. By the optimization in steps 4 to 9, $v_m \preceq_p closest(v_m, S_1)$. Moreover, by step 17, if $M \cap S_1 = \{v_1\}$, then the next anchor node selected either succeeds or is equal to $next(v_1, S_1)$. Thus, since no two matches computed by BMS share the same node from S_1, the claim is established for BMS. □

    We now consider the costs of the various functions. Let d denote the height of the XML data tree, and let S denote the data node list with the highest frequency; i.e. $|S_i| \leq |S|$ for $i \in [1, k]$. As in [14], we assume that each data node is stored together with its Dewey label which enables the lca function to be computed efficiently in $O(d)$ time. The cost of $first(v_1, \cdots, v_k)$ and $last(v_1, \cdots, v_k)$ is each $O(k)$. The cost of $pred(v, S_i)$, $next(v, S_i)$, $out(v, S_i)$, and $closest(v, S_i)$ is each $O(d \log(|S_i|))$ based on a binary search of S_i and comparing nodes using their Dewey labels.
    For the BMS algorithm, the cost to compute a candidate SLCA is $O(kd \log(|S|))$ (due to step 10). Since there are at most $|S_1|$ candidate SLCAs (by Lemma 3.7), the time complexity of BMS algorithm is $O(kd|S_1| \log(|S|))$. For the IMS algorithm, the cost to compute a candidate SLCA is also $O(kd \log(|S|))$ (due to steps 10 and 11); thus the time complexity of IMS algorithm is also $O(kd|S_1| \log(|S|))$. However,

---

Algorithm 3 Simple AND-OR-SLCA (Q)

1: let $Q = C_1 \vee C_2 \vee \cdots \vee C_k$, where each
    $C_i = w_{i,1} \wedge w_{i,2} \wedge \cdots \wedge w_{i,n_i}$, $n_i \geq 1$
2: initialize R to be empty
3: **for** i = 1 to k **do**
4:    let T = result of evaluating C_i using some AND-algorithm
5:    **for** each node $v \in T$ **do**
6:      initialize toDeleteV = false
7:      **for** each node $v' \in R$ **do**
8:        **if** $(v \preceq_a v')$ **then**
9:          toDeleteV = true
10:          exit inner for-loop
11:        **else if** $(v' \preceq_a v)$ **then**
12:          remove v' from R
13:        **end if**
14:      **end for**
15:      **if** (toDeleteV) **then**
16:        delete v from T
17:      **end if**
18:    **end for**
19:    $R = R \cup T$
20: **end for**
21: **return** R

---

our experimental results show that IMS outperforms BMS as the number of candidate SLCAs computed by IMS is less than that by BMS (by up to a factor of 30).

In comparison, the time complexities of SE and ILE are, respectively, $O(kd|S|)$ and $O(|S_1|kd\log(|S|) + |S_1|^2)$ [14].

4. AND-OR KEYWORD SEARCH

In this section, we examine how to process more general SLCA-based keyword search queries that go beyond the AND-semantics in conventional queries to support any combination of AND and OR boolean operators. We consider AND-OR keyword search queries of the form: $Q = (Q) \mid Q$ and $Q \mid Q$ or $Q \mid w$, where w denotes some keyword.

In the following, we consider two approaches to process SLCA-based AND-OR keyword search queries. The first approach is a straightforward application of the algorithms presented in the preceding section for processing conventional SLCA-based AND-keyword search queries by expressing the general AND-OR queries in disjunctive normal form (DNF). The second approach is an extension of our Multiway-SLCA approach (MS).

4.1 Simple AND-OR Algorithm (SA)

A straightforward approach to process a general AND-OR query Q is to rewrite Q in DNF and evaluate it in two stages: first, evaluate each disjunct in Q using an existing AND-query evaluation algorithm; next, the results of the individual evaluations are combined by eliminating intermediate SLCAs that are ancestor nodes of some other intermediate SLCAs. The algorithm, referred to as *Simple AND-OR (SA)* is shown in Algorithm 3.

4.2 AND-OR Multiway-SLCA (AOMS)

In this section, we show how our Multiway-SLCA approach for processing conventional AND-keyword search queries can be easily generalized to process AND-OR keyword search queries. The extended algorithm, called *AND-OR Multiway-SLCA (AOMS)*, is shown in Algorithm 4.

Our approach requires a general AND-OR query Q to be expressed in conjunctive normal form (CNF), $C_1 \wedge \cdots \wedge C_n$,

---

Algorithm 4 AND-OR Multiway-SLCA

1: initialize $x = 1$; $\alpha_1 = d_{root}$
2: let $v_m = last(\{first(C_i) \mid i \in [1,n]\})$, where $v_m = first(C_m)$
3: **while** $(v_m \neq$ null$)$ **do**
4:    **if** $(m \neq 1)$ **then**
5:      $v_1 = closest(v_m, C_1)$
6:      **if** $(v_m \prec_p v_1)$ **then**
7:        $v_m = v_1$
8:      **end if**
9:    **end if**
10:   $P = \{pred(v_m, C_i) \mid i \in [1,n], i \neq m\} \cup \{v_m\}$
11:   $N = \{next(v_m, C_i) \mid i \in [1,n], \; next(v_m, C_i) \neq null\}$
12:   initialize $r_{max} = last(N)$; $r = v_m$
13:   **repeat**
14:      remove ℓ from P, where $\ell = first(P)$
15:      $\alpha = lca(\ell, r)$
16:      $r = last(r, v)$ where $v \in N$ s.t. $v = next(v_m, C_i), \ell \in S_{i,j}$
17:   **until** $(r = null)$ or $(\alpha \npreceq_a r)$ or $(r = r_{max})$
18:   **if** $(r = null)$ or $(\alpha \npreceq_a r)$ **then**
19:      **if** $(\alpha_x \preceq_a \alpha)$ **then**
20:        $\alpha_x = \alpha$
21:      **else if** $(\alpha \npreceq_a \alpha_x)$ **then**
22:        $x = x + 1$; $\alpha_x = \alpha$
23:      **end if**
24:      $v_m = last(r, out(\alpha_x, C_1), \cdots, out(\alpha_x, C_n))$
25:   **else**
26:      $v_m = r$
27:   **end if**
28: **end while**
29: **if** $(\alpha_1 = d_{root})$ **then return** $\emptyset$ **else return** $\{\alpha_1, \cdots, \alpha_x\}$

---

where each conjunct $C_i = w_{i,1} \vee \cdots \vee w_{i,n_i}$. is a disjunction of n_i keywords, $n_i \geq 1$. We use $S_{i,j}$ to denote the list of data nodes associated with each keyword $w_{i,j}$, $i \in [1,n], j \in [1,n_i]$.

Note that AOMS algorithm is almost equivalent to IMS algorithm except that the six functions $first$, $last$, $pred$, $next$, $closest$, and out now involve a conjunct C_i instead of a keyword list S_i. These mildly generalized versions of the definitions from Section 2 are extended as follows:

- $first(C_i) = first(\{first(S_{i,j}) \mid j \in [1,n_i], first(S_{i,j}) \neq null\})$.

- $last(C_i) = last(\{last(S_{i,j}) \mid j \in [1,n_i], last(S_{i,j}) \neq null\})$.

- $pred(v, C_i) = last(\{pred(v, S_{i,j}) \mid j \in [1,n_i], pred(v, S_{i,j}) \neq null\})$

- $next(v, C_i) = first(\{next(v, S_{i,j}) \mid j \in [1,n_i], next(v, S_{i,j}) \neq null\})$

- $closest(v, C_i) = pred(v, C_i)$ if $lca(v, next(v, C_i)) \prec_a lca(v, pred(v, C_i))$; otherwise, $closest(v, C_i) = next(v, C_i)$.

- $out(v, C_i) = first(\{out(v, S_{i,j}) \mid j \in [1,n_i], out(v, S_{i,j}) \neq null\})$

For simplicity and without loss of generality, we assume that $|C_1| \leq |C_i|$ for $i \in [1,n]$ where $|C_i| = \sum_{j=1}^{n_i} |S_{i,j}|$.

5. EXPERIMENTAL STUDY

To verify the effectiveness of our proposed algorithms, we conducted extensive experiments to compare their performance against existing approaches for evaluating both AND as well as AND-OR keyword search queries.
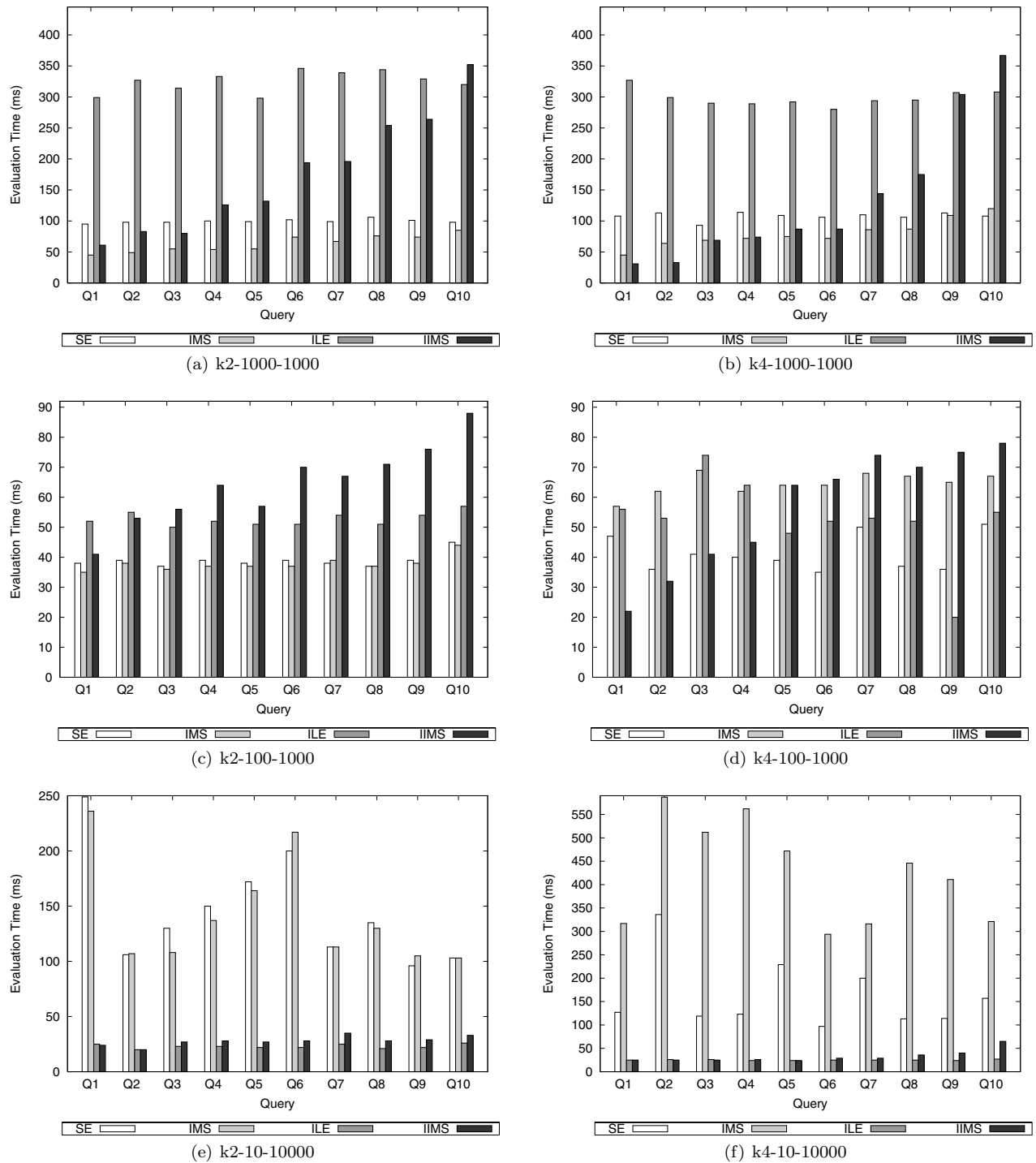
(a) k2-1000-1000

(b) k4-1000-1000

(c) k2-100-1000

(d) k4-100-1000

(e) k2-10-10000

(f) k4-10-10000

Figure 3: Comparison for AND queries

5.1 Experimental Setup

Similar to what was done in [14], where there is both a non-indexed version of the algorithm (i.e., SE) as well as an indexed version of the algorithm (i.e., ILE), we also created two versions for each of our proposed algorithms. We use IBMS, IIMS, and IAOMS, respectively, to refer to the indexed versions of BMS, IMS, and AOMS. As in [14], in the non-indexed algorithms, all the data nodes are organized using a B-tree where the B-tree keys are the keywords of the data nodes and the B-tree data associated with each B-tree key is a list of Dewey labels of the data nodes having that key as keyword. In the indexed algorithms, all the data nodes are organized using a B-tree where each B-tree key is a composite key consisting of a keyword (as primary key) and a Dewey number (as secondary key). No data values are associated with this composite-key organization.

For the simple approach of evaluating AND-OR queries (Section 4.1), we have six variants of the algorithm denoted by SA-SE, SA-BMS, SA-IMS, SA-ILE, SA-IBMS, and SA-IIMS; where SA-X denotes the variant using algorithm X to evaluate the AND-subqueries of the AND-OR query.

The evaluation of the algorithms was carried out by using different classes of queries. Each class of AND queries is denoted by `kN-L-H`, where N, L, and H are three positive integer values: N represents the number of keywords, and L and H, with $L \leq H$, represent two keyword frequencies such that one of the N keywords has the low frequency L while each of the remaining $N - 1$ keywords has the high frequency H; thus, $L = |S_1|$. Each class of AND-OR queries (in CNF) is denoted by `cM-kN-L-H`, where M represents the number of conjuncts in a query, N represents the number of keywords in each conjunct, L represents the frequency of each keyword in one conjunct, and H (with $L \leq H$) represents the frequency of each keyword in the remaining $M - 1$ conjuncts. For each class of queries, a set of 10 random queries were generated and each query was executed six times and its average execution time over the last five runs was computed. All the experiments were conducted using a DBLP dataset [6] with two million data nodes.

Our implementation used BerkeleyDB (Java Edition) [3] to store the keyword data lists similar to what was done in [14]. The BerkeleyDB database was configured using a page size of 8KB and a cache size of 1GB. All the experiments were conducted on a 3GHz dual-core desktop PC with 1GB of RAM.

5.2 AND Keyword Search Queries

Figure 3 shows the comparison of the two binary-SLCA algorithms (SE and ILE) against our multiway-SLCA algorithms IMS and IIMS. To avoid cluttering the graphs, we have omitted the BMS and IBMS algorithms as they were outperformed by the IMS and IIMS algorithms (by up to an order of magnitude), respectively. Compared to the BMS variants, the IMS variants not only incur fewer number of candidate SLCA computations but they are also more efficient in SLCA computations.

Each graph in Figure 3 shows the performance comparison for a different query class. For each query class, the ten random queries shown (Q1 to Q10) are ordered in non-descending order of the number of candidate SLCA computations incurred by the IMS algorithm. Figures 3(a) and 3(b) show the results for the case where the low and high frequencies are the same. Comparing IMS and IIMS, we

observe that IIMS generally performs better than IMS only when the number of candidate SLCA computations is small. For the binary-SLCA algorithms, our results are consistent with [14] with SE outperforming ILE. Overall, IMS generally performs the best for both `k2-1000-1000` and `k4-1000-1000` with IIMS performing better than IMS for some cases.

Figures 3(c) and 3(d) show the results for the case where the low and high frequencies differ by a factor of 10. In this case, the non-indexed methods generally perform better than the indexed methods. For `k2-100-1000`, IMS has an edge over SE. For `k4-100-1000`, IIMS performs the best when the number of its candidate SLCA computations is small; otherwise, SE generally has the best performance. Figures 3(e) and 3(f) show the results for the case where the low and high frequencies differ by a factor of 100. In this case, the indexed methods perform better than the non-indexed methods. IIMS and ILE are comparable when the number of candidate SLCA computations by IIMS is small; otherwise, ILE gives better performance.

5.3 AND-OR Keyword Search Queries

Figure 4 compares the performance of algorithms SA-SE, SA-IMS, AOMS, SA-ILE, SA-IIMS, and IAOMS for AND-OR queries; algorithms SA-BMS and SA-IBMS have been omitted as they are outperformed by the IMS variants. The evaluation times shown in Figure 4 are average evaluation times of ten queries. Figures 4(a) and 4(b) show the results for the case where the low and high frequencies are the same, while Figures 4(c) and 4(d) show the results for the case where the low and high frequencies differ by a factor of 10 and 100, respectively. In all these cases, the non-indexed methods outperform their indexed counterparts, with AOMS giving the best performance.

6. RELATED WORK

Efficient algorithms for computing LCAs on trees have been carefully studied by a number of early work [2, 9]. However, the algorithms designed there are meant for main-memory resident data. Schmidt et al. [13] propose the *meet* operator for querying XML document by computing the LCAs of nodes in XML trees. XRANK [8] proposes a stack-based keyword search algorithm and the results are ranked by a Page-Rank hyperlink metric extended to XML. Their ranking techniques are orthogonal to the retrieval and could be easily incorporated into our work. Another work XSEarch [5], which is an extension of information-retrieval techniques, is mainly focused on the semantics and ranking of query results.

The research work in [14, 12] is the most closely related to our current work, and both work adopt the idea of smallest LCA (SLCA) or Meaningful LCA (MLCA), which are similar ideas. Li et al. [12] propose a novel schema-free way to incorporate keyword search in XQuery. They also develop an efficient stack-based MLCA searching algorithm. XKSearch [14] focuses on finding the smallest LCA of keywords in XML documents, and it proposes several algorithms, which we compared against in this paper.

More recently, there has also been a lot of interest on keyword search in relational database systems [1, 4, 7, 10, 11] where the emphasis is mainly on optimizing join queries to generate tree tuples.

7. CONCLUSIONS

In this paper, we examined the problem of processing SLCA-based keyword search queries for XML data. We have presented a novel approach called *multiway-SLCA approach* that is more efficient than the state-of-the-art *binary-SLCA* approach for evaluating SLCA-based keyword queries. In addition, we have also extended our approach to process more general keyword search queries that go beyond the traditional AND semantics to support any combination of AND and OR boolean operators. Our experimental performance evaluation using real XML datasets demonstrate the efficiency of our new algorithms compared to previous algorithms. As part of our future work, we intend to extend our approach to handle complex keyword search queries with any combination of AND, OR, and NOT operators.

8. REFERENCES

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

[2] M. Bender and M. F.Colton. The LCA problem revisited. In *Latin American Theoretical Informatics*, 2000.

[3] Berkeley DB. http://www.sleepycat.com.

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

[5] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. In *VLDB*, 2003.

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

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

[8] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *SIGMOD*, 2003.

[9] D. Harel and R. E. Tarjan. Fast algorithm for finding nearest common ancestors. In *SIAM Journal on Computing*, 1984.

[10] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *VLDB*, 2002.

[11] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword Proximity Search on XML Graphs. In *ICDE*, 2003.

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

[13] A. Schmidt, M. Kersten, and M. Windhouwer. Querying XML Document Made Easy:Nearest Concept Queries. In *ICDE*, 2001.

[14] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Database. In *SIGMOD*, 2005.
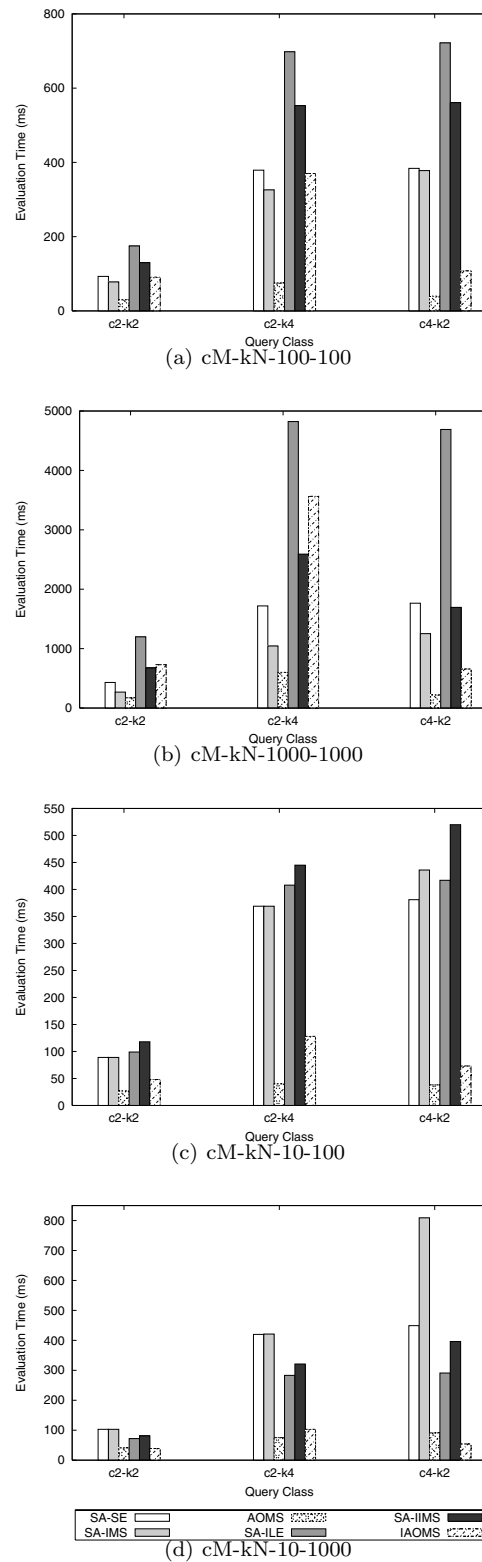
(a) cM-kN-100-100

(b) cM-kN-1000-1000

(c) cM-kN-10-100

(d) cM-kN-10-1000

Figure 4: Comparison for AND-OR queries