Efficient query processing for XML keyword queries based on the IDList index

Junfeng Zhou · Zhifeng Bao · Wei Wang · Jinjia Zhao · Xiaofeng Meng

Received: 12 September 2012 / Revised: 24 March 2013 / Accepted: 29 March 2013
© Springer-Verlag Berlin Heidelberg 2013

Abstract  Keyword search over XML data has attracted a lot of research efforts in the last decade, where one of the fundamental research problems is how to efficiently answer a given keyword query w.r.t. a certain query semantics. We found that the key factor resulting in the inefficiency for existing methods is that they all heavily suffer from the common-ancestor-repetition problem. In this paper, we propose a novel form of inverted list, namely the IDList; the IDList for keyword $k$ consists of ordered nodes that directly or indirectly contain $k$. We then show that finding keyword query results based on the smallest lowest common ancestor and exclusive lowest common ancestor semantics can be reduced to ordered set intersection problem, which has been heavily optimized due to its application in areas such as information retrieval and database systems. We propose several algorithms that exploit set intersection in different directions and with or without using additional indexes. We further propose several algorithms that are based on hash search to simplify the operation of finding common nodes from all involved IDLists. We have conducted an extensive set of experiments using many state-of-the-art algorithms and several large-scale datasets. The results demonstrate that our proposed methods outperform existing methods by up to two orders of magnitude in many cases.

Keywords  XML · Keyword Query Processing · LCA · SLCA · ELCA

1 Introduction

Keyword search on XML data has received much attention in the literature recently [1,4,7,12,14,16–18,22,25–27,31]. Finding efficient query processing method for keyword search on XML data is a fundamental problem in this area, as many applications demand fast query execution speed for multiple users simultaneously [15].

Typically, an XML document can be modeled as a node-labeled tree $T$. For a given keyword query $Q$, researchers have proposed different search semantics [7,12,14,16,26] to define meaningful results based on the notion of LCA, which is the lowest common ancestor of a set of nodes, of which each one directly contains at least one distinct query keyword of $Q$. Among these LCA-based semantics, the most widely adopted ones are arguably Smallest Lowest Common Ancestor (SLCA) [4,17,18,22,25,26] and Exclusive Lowest Common Ancestor (ELCA) [4,12,13,27,31]. An SLCA node $v$ of $Q$ on $T$ satisfies that $v$ is an LCA node of $Q$ on $T$, and no other LCA node of $Q$ can be $v$’s descendant. ELCA semantics is slightly more complex: A node $v$ is an ELCA node if (1) $v$ is an LCA node of $Q$ on $T$, and no other LCA node of $Q$ can be $v$’s descendant. ELCA semantics is slightly more complex: A node $v$ is an ELCA node if (1) $v$ is an LCA node of $Q$ on $T$, and no other LCA node of $Q$ can be $v$’s descendant. ELCA semantics is slightly more complex: A node $v$ is an ELCA node if (1) $v$ is an LCA node of $Q$ on $T$, and no other LCA node of $Q$ can be $v$’s descendant. ELCA semantics is slightly more complex: A node $v$ is an ELCA node if (1) $v$ is an LCA node of $Q$ on $T$, and no other LCA node of $Q$ can be $v$’s descendant.
To facilitate SLCA/ELCA computation, the basic idea of existing methods is firstly choosing a set of nodes according to their positional relationship, then computing their LCA nodes followed by an appropriate pruning according to specific constraints of the corresponding semantics. As the Dewey label [23] of a node $v$ consists of a sequence of components that implicitly contain all ancestor nodes on the path from the document root to $v$, Dewey labeling scheme is a natural choice of the state-of-the-art algorithms [12,17,22,26,27,31] for result computation, where the two basic operations are

\( (OP1) \) testing the document order of two nodes, 
\( (OP2) \) computing the LCA of two nodes.

However, as each Dewey label consists of a set of components that collectively represent a node, the cost of $OP1$ and $OP2$ operations is both linear to the height of the XML tree. Moreover, as each component of a Dewey label corresponds to a node in the XML tree, either $OP1$ or $OP2$ operation is equal to visiting all common ancestors of the two involved nodes once. In practice, as a node $v$ could be a common ancestor of multiple nodes, frequently executing $OP1$ and $OP2$ operations will result in all nodes on the path from the root to $v$ be repeatedly visited, which we call as common-ancestor-repetition (CAR). The CAR problem is inherent in algorithms [1,12,17,18,22,26,27,31] that are based on the $OP1$ and $OP2$ operations, which is inefficient yet difficult to make optimization.

To address the CAR problem, we propose a suite of novel and efficient algorithms for answering SLCA/ELCA queries that depart from existing Dewey labeling based approaches. Specifically, we make the following contributions.

1. We assign each node a unique ID which is compatible with the document order, based on which we propose a new kind of inverted index, namely IDList. For each keyword $k$, the corresponding IDList consists of all distinct IDs of nodes that directly or indirectly contain $k$. Compared with existing inverted lists of Dewey labels, each node is recorded only once in an IDList, and all necessary information for answering a given keyword query is maintained without any loss.

2. We show that SLCA/ELCA computation can be cast into a variant of the set intersection problem [8–10,21,24]. Then, we propose a family of efficient algorithms based on the set intersection operation to compute the set of qualified results. We also design several optimization techniques that exploit the positional relationships between nodes in the XML tree to accelerate the computation. Another salient feature of our methods is that they are actually very simple to be implemented and can leverage any one of existing fast set intersection algorithms.

3. To further improve the overall performance, we consider the existence of additional hash indexes [25,31] on IDLists and propose several algorithms to accelerate SLCA/ELCA computation.

4. We conducted an extensive set of performance studies to compare our proposed algorithms with 13 state-of-the-art algorithms. Our experimental results show that our methods outperform existing approaches by up to two orders of magnitude in many cases.

As an extension of [30], we have several major updates:

1. We propose several algorithms that are based on hash search to accelerate SLCA/ELCA computation in Sect. 7.

2. We make a comparison between our methods and existing methods and discuss the extension of our methods to other LCA-based semantics in Sect. 8. (3) We conduct more performance study to make comparison with the state-of-the-art algorithms in Sect. 9. (4) We give an in-depth analysis to existing methods on set intersection (Sect. 2.3) and SLCA/ELCA computation (Sect. 2.4).

The rest of the paper is organized as follows. In Sect. 2, we introduce preliminaries and related work. In Sect. 3, we introduce the CA tree and its properties. The inverted list and its properties are introduced in Sect. 4. The algorithms for SLCA/ELCA computation are discussed in Sects. 5 and 6, respectively. In Sect. 7, we introduce algorithms for SLCA/ELCA computation based on hash search. In Sect. 8, we present an theoretical analysis and comparison of our and existing algorithms and discuss extensions to other LCA-based semantics. In Sect. 9, we present experimental results and then conclude our work in Sect. 10.

2 Preliminaries and related work

2.1 Data model

We model an XML document as a labeled ordered tree, where nodes represent elements or attributes, while edges represent direct nesting relationship between nodes. We say a node $v$ directly contains a keyword $k$ (also say $v$ is a keyword node) if $k$ appears in the node name, attribute name, or text value of $v$. We say $v$ contains $k$ if $k$ appears at least once in the subtree rooted at $v$. Figure 1 is a sample XML document.

The positional relationships between two nodes include Document Order ($≺_d$), Equivalence ($\equiv$), AD (ancestor - descendant, $≺_a$), PC (parent-child, $≺_p$), Ancestor-or-self ($\leq_a$), and Sibling relationship. Given two nodes $u$ and $v$: $u ≺_d v$ means that $u$ is located before $v$ in document order; $u ≺_a v$ means that $u$ is an ancestor node of $v$; $u ≺_p v$
2.2 Query semantics

Given a query \( Q = \{k_1, k_2, \ldots, k_m\} \) and an XML document \( D \), we use \( L^D_i \) to denote the inverted list of \( k_i \) that consists of labels of Dewey or one of its variants, such as JDewey [4] or IDDewey [25,31]. All labels of \( L^D_i \) are sorted by document order. Let \( LCA(v_1, v_2, \ldots, v_m) \) be the lowest common ancestor (LCA) of nodes \( v_1, v_2, \ldots, v_m \), the LCAs of \( Q \) on \( D \) are defined as \( LCA(Q) = \{v | v = LCA(v_1, v_2, \ldots, v_m), v_i \in L^D_i (1 \leq i \leq m)\} \). For example, the LCA nodes for \( Q = \{k_1, k_2\} \) on the XML document in Fig. 1 are nodes with IDs 1, 3, 8 and 15.

Based on the LCA semantics, the most widely adopted variants are SLCA [22,26] and ELCA [12,27,31]. SLCA defines a subset of \( LCA(Q) \), of which no one is the ancestor of any other LCA, which can be formally defined as follows.

Definition 1 (SLCA) Given a keyword query \( Q = \{k_1, k_2, \ldots, k_m\} \) and an XML document \( D \), the set of SLCAs of \( Q \) on \( D \) are \( SLCA(Q) = \{v | v \in LCA(Q) \text{ and } \forall v' \in LCA(Q), v \sim_a v'\} \).

The definition of ELCA is a bit more complex: A node \( v \) is an ELCA node if the subtree rooted at \( v \) contains at least one occurrence of all query keywords, after excluding the occurrences of the keywords in each subtree rooted at a descendant LCA node of \( v \), as shown by Definition 2.

Definition 2 (ELCA) Given a keyword query \( Q = \{k_1, k_2, \ldots, k_m\} \) and an XML document \( D \), the set of ELCAs of \( Q \) on \( D \) are \( ELCA(Q) = \{v | v \in L^D_i, v \in LCA(v_1, \ldots, v_m) \wedge \forall i \in [1, m], \exists x \in LCA(Q) \wedge child(v, v_i) \preceq d x \} \), where \( child(v, v_i) \) is the child of \( v \) on the path from \( v \) to \( v_i \).

Example 1 Consider the query \( Q = \{k_1, k_2\} \) and the sample XML document \( D \) in Fig. 1. Although nodes 1 and 8 are LCAs, they are ancestors of node 15, and hence not SLCAs. The set of SLCAs of \( Q \) on \( D \) are nodes with IDs 3, 8, and 15. As nodes 3 and 15 do not contain other descendant LCA nodes, they are ELCA nodes. For node 8, after removing the subtree rooted at node 15, the subtree rooted at node 8 still contains all query keywords, and thus, node 8 is a qualified ELCA result. Similarly, node 1 is not an ELCA node. Therefore, the set of ELCA nodes for \( Q \) on \( D \) are those with IDs 3, 8, and 15.

Obviously, SLCA semantics requires the resulting LCAs to be lowest, such that each query result is a tightest XML fragments containing all the keywords. As a comparison, ELCA semantics may take some LCA nodes that are not SLCA nodes as meaningful results, that is, ELCA attempts to capture more meaningful results.

2.3 Set intersection methods

Finding common elements from several sorted lists is a central operation in information retrieval engines, search engines, and databases. Given \( m \) lists \( L_1, L_2, \ldots, L_m \) (without loss of generality, assume that \( |L_i| \leq |L_2| \leq \cdots \leq |L_m| \)), to efficiently find the common components that appear in all these lists, we need to repeatedly pick an element \( e \) from a list \( L_i \) and use it as the eliminator to probe all other lists. If \( e \) appears in all the \( m \) lists, it is output as a result. The overall performance of such set intersection algorithm is dominated by two orthogonal factors:

1. the number of probe operations, which is in turn heavily impacted by the probe order, that is, which list should be probed first,
2. the cost of each probe operation, that is, how to efficiently find the matched element\(^1\) for eliminator \( e \), which is further affected by two orthogonal factors:

1. which search method is used (e.g., binary, galloping [3], or interpolation search [2]),
2. how large the search interval is.

\(^1\) The matched element in \( L_i \) to eliminator \( e \) is the minimum element that is equal to or greater than \( e \), if all lists are sorted in ascending order.
For probe order, SvS [9] computes the final results by processing two lists each time from the shortest to the longest. The Adaptive algorithm [8] computes the intersection by repeatedly cycling through the set of lists in a round-robin fashion. However, it does not always exhibit the best performance due to the overhead of adaptivity [9]. To address this problem, the Small Adaptive algorithm [9] was proposed by adopting galloping search [3], limiting the form of adaptivity, and changing the join order according the number of unprocessed components in each list. The Quantile-based algorithm [24] deduces in advance a good join order by first partitioning all lists into sub-partitions, thus avoids the overhead of adaptivity in choosing the join order. The Probabilistic Intersection algorithm [21] uses a probabilistic probing policy to determine which list to examine next based on historical data called “average jump.” [10] uses partitioning and hashing to quickly eliminate parts of the sets that do not have overlap and hence is suitable for cases where the result’s size is small.

### 2.3.2 Search method

To complete a probe operation using eliminator $e$ on a search interval $L[a, b]$, where $a$ and $b$ are the position values of the first and the last elements in $L$, existing methods usually adopt either one of the three commonly used search methods: binary, galloping, and interpolation search.

Binary search always checks whether the element $L[m]$, in the middle position of $L[a, b]$ is equal to $e$, that is, $m = \frac{a + b}{2}$. If $L[m] = e$, it stops the probe operation and returns $L[m]$ as the matched element. If $L[m] < e$, it recursively finds the matched element of $e$ from $L[m + 1, b]$; otherwise, it completes this task according to the search interval $L[a, m - 1]$.

Galloping search [3] firstly finds a smaller search interval $L[a', b']$ from $L[a, b]$ such that $L[a'] \leq e \leq L[b']$ and then finds the matched element from $L[a', b']$ using binary search method. To get this search interval, it repeatedly tests a search interval $L[a_i, b_i]$ that consists of $2^i (i = 1, 2, 3, \ldots)$ elements until $L[a_i, b_i]$ satisfies $L[a_i] \leq e \leq L[b_i]$. Let $L[a_{i-1}, b_{i-1}]$ be the $(i-1)$th search interval, the relationship between two continuous search intervals can be stated as follows: $a_{i-1} - b_{i-1} = 1$ and $b_i - a_i = 2^i - 1$. The starting position can be from either left or right of $L[a, b]$.

Interpolation search [2] uses Formula 1 to find the position of the median element. Obviously, it can locate the matched component of $e$ very quickly when components of $L[a, b]$ are uniformly distributed; otherwise, it may not be as efficient as binary and galloping search because of its higher cost in computing the position of the median element.

$$m = \left\lfloor \frac{e - L[a]}{L[b] - L[a]} \right\rfloor \times (b - a) + a \quad (1)$$

### 2.3.3 Search interval

Assume that $n$ elements have been processed for $L$, then the length of the search interval is $|L| - n$. Obviously, the larger the search interval is, the more the number of comparison operations, that is, the number of search steps is needed for binary search to complete a probe operation. For galloping search, the number of comparison operations is determined by the distance from the current element to the matched one. The longer the distance, the more comparison operations are needed. For interpolation search, the number of comparison operations is greatly affected by the distribution of the underlying elements. Even though, for both galloping and interpolation search, the larger the search interval is, the more the number of comparison operations is needed on average to complete a probe operation.

### 2.4 Algorithms for SLCA and ELCA computation

Many LCA-based query semantics [7,12,14,16,22,26,27,31] have been proposed to define query results for an XML keyword query. We refer readers to the recent tutorials for a complete coverage [5,6,19], among which the two widely adopted semantics are SLCA [22,26] and ELCA [12,27,31].

To facilitate SLCA/ELCA computation, existing methods usually assign each node $v$ a Dewey label [1,12,17,18,22,26,27], or one of its variants, such as JDewey [4] and IDDewey [25,31] (Dewey labels consisting of node IDs that are compatible with the document order), based on which inverted lists are built for all keywords for fast result computation. However, as each component $c$ of $v$’s Dewey/JDewey/IDDewey label corresponds to a node $w(w \preceq_a v)$ in the XML tree, and as $w$ could be a common ancestor of multiple nodes, $c$ may appear in Dewey/JDewey/IDDewey labels of many other nodes. Further, as all Dewey/JDewey/IDDewey labels in an inverted list are maintained separately, for a given keyword query $Q$, processing different Dewey/JDewey/IDDewey labels could result in accessing $c$ multiple times, which equals visiting $w$ multiple times, and is called as common-ancestor-repetition (CAR). In this section, we would like to make a deep analysis to various algorithms and illustrate how each of them suffers from the CAR problem.

### 2.4.1 Analysis on algorithms for SLCA computation

For SLCA computation, existing methods can be classified into three categories:
(1) algorithms that are based on inverted lists of Dewey labels, such as Stack [26], IL [26], and IMS [22],
(2) algorithm that is based on inverted lists of JDewey labels, such as JDewey [4],
(3) algorithm that is based on hash table and inverted lists of IDDewey labels, such as HS [25].

Among the first kind of algorithms, Stack processes all Dewey labels in document order by using a stack to merge Dewey labels on the fly and simultaneously computing qualified SLCA nodes. IL computes the SLCA results by processing two lists each time from the shortest to the longest. IMS computes each potential SLCA by taking one node from each list in each iteration. Compared with Stack, IL and IMS are more flexible to utilize the positional relationship to prune useless keyword nodes. The difference between IL and IMS lies in that in each iteration, IL always uses a node of the shorter list to probe the longer list, while IMS always uses a maximum node to probe other lists. IL is simpler than IMS in each iteration, while usually suffers from more iterations than IMS. IMS needs the least number of iterations, but suffers from the highest cost in each iteration. As discussed in Sect. 1, the two basic operations for these methods are $(OP_1)$ testing the document order of two nodes and $(OP_2)$ computing the LCA of two nodes; therefore, they heavily suffer from the CAR problem.

The second kind of algorithm, that is, JDewey [4], performs set intersection operation on all lists of each tree depth from the leaf to the root. For all lists of each level, after finding the set of common nodes, it needs to recursively delete all ancestor nodes in all lists of higher levels. Even though the JDewey algorithm does not depend on $OP_1$ and $OP_2$ operation, it still suffers from the CAR problem, since a node could be a parent of many other CA nodes, and the deletion operation needs to process these CA nodes separately.

Different from the above algorithms, the third kind of algorithm, that is, HS [25], takes the shortest list $L^D_1$ as the working list and sequentially processes all IDDewey labels of $L^D_1$. In each iteration, it picks from $L^D_1$ an IDDewey label $l$ and checks whether the nodes represented by IDs of $l$ contain all keywords of the given query in their subtree. By maintaining a hash mapping between each pair of node and keyword, the checking of whether a node contains a certain keyword in its subtree becomes a probe operation on the hash table. However, HS still suffers from the CAR problem, since it processes each IDDewey label separately, without noticing that some IDs repeatedly appear in many different IDDewey labels in the same inverted IDDewey label list.

Table 1 shows the comparison of these algorithms, from which we know that IL [26], IMS [22], IS [27] and JDewey [4] are better than Stack [26], and DIL [12] according to their time complexity; and HS [25] and HC [31] are better than IL, IMS, IS, and JDewey for the same reason. As discussed above, the common problem that results in their inefficiency is that they all heavily suffer from the CAR problem.

### 2.4.2 Analysis on algorithms for ELCA computation

Similar to algorithms for SLCA computation, the algorithms for ELCA computation can also be classified into three categories according to which kind of inverted index is used.

Among the first kind of algorithms, DIL [12] works in the same way as Stack. IS [27] takes the shortest list $L^D_1$ as the working list. In each iteration, it picks from the shortest list $L^D_1$ an IDDewey label $l$ and uses it to probe other lists to get a candidate ELCA node. The two basic operations of DIL and IS are $OP_1$ and $OP_2$ operations. The second kind of algorithm, that is, JDewey, computes all ELCA nodes in the same way as that of SLCA computation. The third kind of algorithm, that is, HC [31], works in the similar way as HS does. Therefore, for the same reason as discussed in Sect. 2.4.1, they all suffer from the CAR problem.

Table 1 shows the comparison of these algorithms, from which we know that IL [26], IMS [22], IS [27] and JDewey [4] are better than Stack [26], and DIL [12] according to their time complexity; and HS [25] and HC [31] are better than IL, IMS, IS, and JDewey for the same reason. As discussed above, the common problem that results in their inefficiency is that they all heavily suffer from the CAR problem.

### 3 From CA to SLCA/ELCA

An important concept underlying our methods is common ancestor, which forms a superset of SLCA/ELCA results.

**Definition 3 (Common Ancestor (CA))** Given a keyword query $Q$ and an XML document $D$, node $v$ is a common ancestor of $Q$ on $D$ if the subtree rooted at $v$ contains each keyword of $Q$ at least once.

For example, for query $Q = \{k_1, k_2\}$, the CA nodes of $Q$ on the XML document in Fig. 1 are nodes 1, 2, 3, 8, and 15. Obviously, we have the following lemma.
Lemma 1 For a given query \( Q \) and an XML document \( D \), 
\( \text{SLCA}(Q) \subseteq \text{ELCA}(Q) \subseteq \text{LCA}(Q) \subseteq \text{CA}(Q) \).

Definition 4 (CA Tree) The CA tree of a keyword query \( Q \) on an XML document \( D \) is defined as \( T = \{V_T, E_T\} \), where 
\( V_T \) is the set of CA nodes of \( Q \) on \( D \), that is, \( CA(Q) = V_T \), 
\( E_T \) is the set of parent-child edges between two nodes of \( V_T \).

Figure 2 shows the CA tree of \( Q = \{ k_1, k_2 \} \) on the XML document of Fig. 1. Each italic number beside a node \( v \) denotes \( v \)'s ID, and \( < N_1, N_2 > \) is the vector associated with \( v \) denoting the number of nodes that directly contain \( "k_1" \) and \( "k_2" \) in the subtree rooted at \( v \).

Lemma 2 For a given CA tree \( T \) of \( Q \) on \( D \), let \( V_T^{leaf} \) be the set of leaf nodes of \( T \), then \( \text{SLCA}(Q) = V_T^{leaf} \).

For instance, the matched SLCA nodes for \( Q = \{ k_1, k_2 \} \) are the leaf nodes of its CA tree with IDs 3 and 15.

According to the definition of ELCA, we have the following lemma, which is similar to [31].

Lemma 3 For a given keyword query \( Q = \{ k_1, k_2, \ldots, k_m \} \) and its CA tree \( T \) on an XML document \( D \), assume that for each node \( v \) in \( V_T \), \( S^\text{child}_{v} = \{ v_1, v_2, \ldots, v_l \} \) is the set of child nodes of \( v \) in \( T \), \( v \) is associated with a vector \( N = \{ N_1, N_2, \ldots, N_m \} \), where \( N_i (i \in [1, m]) \) denotes, among all the nodes in the subtree rooted at \( v \), the number of nodes that directly contain \( k_i \), then \( v \) is an ELCA node if \( \forall i \in [1, m] \) such that \( v.N_i = \sum_{j=1}^{i} v_j.N_i \).

Proof As each \( v_i \in S^\text{child}_{v} \) is a CA node, if \( v_i \) contains only one child CA node \( v_{i_r} \), and all other child nodes of \( v_i \) in the XML document do not contain any keyword of \( Q \), then \( v_i \) is not an LCA node. Therefore, if \( v_i \) is not an LCA node, \( v_i \) contains the same set of keyword nodes w.r.t. keywords of \( Q \). If \( v_i \) is not an LCA node either, we can recursively get the closest descendant LCA node of \( v_i \), such that both nodes contain the same number of keyword nodes of \( Q \). Let \( T_e \) be the subtree rooted at \( v \), according to Definition 2, the operation of removing all keyword nodes of \( Q \) from all subtrees rooted at \( v \)'s descendant LCA nodes equals removing keywords from all subtrees rooted at \( v \)'s children. Thus, if \( \exists i \in [1, m] \), such that \( v.N_i = \sum_{j=1}^{i} v_j.N_i \), it means that after removing all subtrees rooted at \( v \)'s child CA nodes (or descendant LCA nodes), \( T_e \) still contains each keyword of \( Q \) at least once, that is, \( v \) is an ELCA node. \( \Box \)

4 IDList and its properties

In this section, we introduce our new indexing data structure, IDList, designed specifically for efficient keyword queries.

We assign each node an ID that will result in the same order as the document order among the nodes. One simply coding scheme is to assign each node its order in a depth-first traversal of the document tree. These IDs are shown as underline numbers beside each node in Fig. 1.

Given a keyword \( k_i \), its corresponding inverted list \( L_i \) consists of sorted entries; each entry corresponds to a node \( v \), such that the subtree rooted at \( v \) contains \( k_i \). More specifically, each entry in \( L_i \) consists of two or three numeric values:

- “ID” is the node ID of a node \( v \),
- “PIDPos” is the array subscript of the entry containing \( v \)'s parent ID in \( L_i \),
- and optionally “\( N_{\text{Desc}} \)” denotes the number of nodes that directly contain \( k_i \) in the subtree rooted at \( v \); this field is only required for ELCA computation.

Entries in \( L_i \) are sorted in ascending order according to their ID values.

Figure 3 shows the two IDLists of \( k_1 \) and \( k_2 \), respectively, where we draw each entry vertical, and hence, it becomes a column in the table. Take \( L_1 \) as an example and let “Pos” denote the array subscript. At Pos 0, the values of ID, PIDPos, and \( N_{\text{Desc}} \) are 1, −1, and 5, where “1” means that the first node in \( L_1 \) has ID=1, “-1” means that node 1
has no parent node, and “5” means that the subtree rooted at node 1 contains five nodes that directly contain “k1”.

We have the following properties with respect to our IDList index.

Property 1 For a given keyword query \( Q = \{k_1, \ldots, k_m\} \) and its CA tree \( T \) on an XML document \( D \), let \( R(L_1, \ldots, L_m) \) be the result set of set intersection operation on the \( m \) IDLists, then \( CA(Q) = R(L_1, \ldots, L_m) = V_T \).

In addition, if the set intersection is performed in ascending order, then results of \( R \) are output in document order, equivalent to visiting \( T \) in document order.

This property enables us to reduce CA computation into a sorted set intersection problem, which can be executed extremely efficiently. Together with Lemmas 1, 2, and 3, we can obtain efficient algorithms on SLCA/ELCA computation.

Property 2 For any two nodes \( u \) and \( v \) of \( L_i \), \( u \prec_p v \) if \( u.ID = L_i[v.PIDPos].ID \).

Notations and Discussions. For simplicity, we do not differentiate a node, its ID, and the corresponding entry in an IDList if there is no ambiguity. For example, when we say node 3, it denotes the node with ID 3 in Fig. 1, it also denotes the entry in \( L_1 \) and \( L_2 \) at Pos 2. Each IDList \( L_i \) is associated with a cursor \( C_i \) pointing to some entry of \( L_i \). Henceforth, \( C_i \) will refer to the entry that \( C_i \) points to. Function \( fwdAdvance(C_i) \) moves \( C_i \) to the next entry, if any. We use \( pos(C_i) \) to denote the “Pos” value of \( C_i \) in \( L_i \), and \( L_i[P] \) to denote an entry of \( L_i \) at Pos \( x \). We have the assertion that if \( pos(C_i) = x \) then \( L_i[P] = C_i \).

It is obvious that our IDList index can be constructed in one pass of the XML document, and almost in the same way as constructing an ordinary inverted index with Dewey codes. Since our coding scheme is essential, the same as range codes [29], efficient methods to update the codes upon modification to the XML data exist (e.g., [11]). Off-the-shelf techniques to update the inverted index [20] can be applied to update our IDList index accordingly.

5 SLCA computation

5.1 Forward solution

Our first algorithm on SLCA computation, that is, FwdSLCA, computes all CA nodes in document order according to Property 1, filters on-the-fly using Property 2 and Lemma 2, and outputs leaf nodes of the CA tree as SLCA answers.

5.1.1 The FwdSLCA Algorithm

The pseudocode is shown in Algorithm 1. It finds a CA node \( v \) by intersecting \( m \) IDLists (line 3) in each iteration (line 2–9). Since it processes nodes in document order, it cannot determine if a CA node is really an SLCA result until it obtains the next CA node. Hence, it buffers the last CA node \( u \). In line 4, if no CA node exists in the remaining entries (when \( flag = false \)), we break out of the loop (line 8), and output \( u \) as a result (line 10); otherwise, when \( flag = true \) in line 4, we check whether \( u \) is the parent of \( v \) by comparing their IDs in line 5 (note that, parent returned in line 3 is the parent node of \( v \)). Since all CA nodes are identified in document order, if \( u \neq_p v \), it means \( u \in V_T^{leaf} \). According to Lemma 2, \( u \) is an SLCA node, then we output it as an SLCA result in line 5. In line 6, we buffer the current CA node \( v \) as a candidate SLCA node in \( u \), then advance each cursor (line 7) before the next iteration.

The function fwdGetCA is called to find a CA node by intersecting \( m \) IDLists, where \( j \) is used to denote the IDList to be probed, \( n \) is the number of occurrences of \( C_k \) in all IDLists. fwdGetCA always uses the cursor with the maximum ID value as the eliminator (line 2, and \( C_k \) is the eliminator) and uses the static probing order from the shortest IDList to the longest (line 1 and 3–13). The probe operation will continue to the remaining IDLists if entries with same ID are found (line 9–10); otherwise, since we have found an entry larger than \( C_k \), \( C_k \) will be reset, we restart the probe from \( L_1 \) immediately (line 12). Binary search (fwdBinSearch) is used to perform the probe—finding a matched node for \( C_i \), though other kinds of search can also be used. Function fwdEof checks whether we have exhausted an IDList by checking the cursor positions.

Example 2 For query \( Q = \{k_1, k_2\} \) and its two IDLists in Fig. 3, FwdSLCA will find all CA nodes of \( Q \), that is, 1, 2, 3, 8, and 15, in document order. Each time it finds a CA node, it will check whether the previous CA node is a leaf CA node, that is, SLCA node, by testing their parent-child relationship. Consider nodes 3 and 8, node 8’s Pos values is 4. Since \( 3 \neq 1 \), \( L_i[4].PIDPos.ID = L_i[0].ID = 1 \), node 3 is a leaf node of \( Q \)’s CA tree, thus node 3 is an SLCA node. Note that node 15 is the last CA node visited in document order, it must be a leaf node of the CA tree, and therefore an SLCA node.

5.1.2 Analysis of the FwdSLCA Algorithm

For a given keyword query \( Q \) of \( m \) keywords, the cost of line 7 in Algorithm 1 is \( O(m) \). Since the cost of fwdBinSearch is \( O(\log |L_m|) \), and the maximum number of iterations is bounded by the size of the smallest IDList, that is, \( |L_1| \);
The main problem with Algorithm 1 is that we may potentially compute many CA nodes that are eventually found not to be SLCA nodes, because there are other CA nodes that are identified later and reside in their subtrees. We give an extreme example below.

Algorithm 1: FwdSLCA(Q)

1. $Q = \{k_1, \ldots, k_m\}, 0 < |L_1| \leq |L_2| \leq \ldots \leq |L_m|$
2. $u \leftarrow \{-1, -1\}$
3. while ($\neg$ fwdEof()) do
4.   if ($flag = TRUE$) then
5.     if (parent.ID $\neq u$.ID) then output u.ID as an answer
6.     $u \leftarrow v$
7.     foreach ($i \in [1, m]$) do fwdAdvance($C_i$)
8.   else break
9. endwhile
10. output u.ID as an answer

Function fwdGetCA():
1. $j \leftarrow 1, n \leftarrow 1$
2. $k \leftarrow \text{argmax}_i(C_i.ID)$
3. while ($n < m$) do
4.   if ($j = k$) then
5.     $j \leftarrow j + 1$
6. endwhile
7. fwdBinSearch($L_j, C_j$)
8. if ($\text{pos}(C_j) \geq |L_j|$) then return \{FALSE, NULL, NULL\}
9. if ($C_i.ID = C_j.ID$) then
10.   $j \leftarrow j + 1, n \leftarrow n + 1$
11. else
12.   $k \leftarrow j, j \leftarrow 1, n \leftarrow 1$
13. endwhile
14. return \{TRUE, $L_j$[1..$\text{PIDPos}$], $C_j$\}

Function fwdEof():
1. if (i, such that $\text{pos}(C_i) = |L_i|$) then return TRUE
2. else return FALSE

Procedure fwdBinSearch($L_j, u$):
1. $s \leftarrow \text{pos}(C_j); c \leftarrow |L_j|$
2. while ($s < c$) do
3.   $mid \leftarrow \lfloor (s + c) / 2 \rfloor$
4.   if ($L_j[mid]$.ID $= u$.ID) then $C_j \leftarrow L_j[mid];$ break
5. else if ($L_j[mid]$.ID $< u$.ID) then $s \leftarrow mid + 1$
6. else $c \leftarrow mid - 1$
7. endwhile
8. if ($s \geq c$) then $C_j \leftarrow L_j[s]$

therefore, the worst-case time complexity of FwdSLCA is $O(m \times |L_1| \log (|L_m|))$.

We also note that any set intersection algorithm can be used in Algorithm 1 for CA computation, and any of the search methods (e.g., binary, galloping [3], or interpolation search [2]) can be used to implement function fwdGetCA. Therefore, our method is not only easy to implement, but also can enjoy the benefits of latest advances in set intersection algorithms (e.g., [10, 28]).

5.2 Backward solution

The main problem with Algorithm 1 is that we may potentially compute many CA nodes that are eventually found not to be SLCA nodes, because there are other CA nodes that are identified later and reside in their subtrees. We give an extreme example below.

Example 3 Consider a query $Q = \{k_1, k_2, \ldots, k_m\}$ and the XML document $D$ in Fig. 4a. The CA tree of $Q$ on $D$ is $T$ in Fig. 4b. According to Algorithm 1, it will call fwdBinSearch $m - 1$ times to find a CA node. However, according to Lemma 2, only node 5 is an SLCA node. As shown in Fig. 4c, Algorithm 1 needs to find each CA node from all IDLists, which is unnecessary but can hardly be avoided, as we cannot tell whether the current CA node is an SLCA node or not unless we obtain the next CA node.

While JDewey [4] solves this inherent inefficiency by computing SLCA results in a bottom-up manner, it needs to memorize all the CA nodes at a particular level of the XML data tree in each iteration, hence requiring large amount of memory. Instead, we propose the novel idea to compute SLCA results by the reverse document order. The basic idea of the resulting backward solution is whenever we found a CA node from set intersection performed backwards, we proactively remove all its ancestor nodes from the shortest inverted IDList. This method will have the salient feature that every time a CA node is found by backward set intersection, it is definitely an SLCA node.

The straight-forward way to implement the above idea requires additional data structures. For example, one can use an auxiliary array whose size equals the size of the shortest IDList; for each CA node returned from an iteration, we iteratively use the PIDPos to find all its ancestors and mark them as invalid nodes in the auxiliary array. Another implementation choice is to maintain a hash table to “remember” all ancestor nodes of SLCA nodes discovered so far.

Instead, we devise a solution without using additional data structure and hence will be both space and time efficient. The idea is to skip the parent node of the current CA node in each iteration, that is, we only need to remember one additional node as compared with JDewey. The correctness is ensured by Lemma 4 below.

Lemma 4 Consider the case where all CA nodes are computed in the reverse document order. Let $v'$ be the CA node returned from the previous iteration, $u'$ be the parent node of $v'$, $v$ be the CA node of the current iteration, then $v$ is an SLCA node if $v \neq u'$.
Case I: \( u \prec_d u' \). Since \( u' \) is the parent of \( v' \), so \( u' \) must be a CA node. Therefore \( u' \) rather than \( v \) will be the current CA node, and hence a contradiction.

Case II: \( u' \prec_d v \prec_d v' \). Since \( v \) is not an SLCA node, according to Lemma 2, \( v \) is not a leaf node of the CA tree. Then \( v \) must have a descendant leaf node \( v_d \) in the CA tree, which means \( u' \prec_d v \prec_d v_d \prec_d v' \). So \( v_d \) rather than \( v \) will be the current CA node, and hence a contradiction.

By summarizing the above two cases, Lemma 4 holds.

The pseudocode of our backward solution is shown in Algorithm 2, where we only list additional or different lines for the auxiliary functions if they are different from their counterparts in Algorithm 1. In each iteration (line 2–10), it finds a CA node \( v \). If no CA node is found, we break the loop (line 5); otherwise, we immediately output the current CA node as an SLCA node (line 4). In line 6, all cursors are moved to the previous entries in all lists. For two consecutive entries \( u \) and \( v (u.ID < v.ID) \) in an IDList, we can skip \( u \) if \( u \) is the parent of \( v \) according to the CA semantics and Lemma 4. In line 7–9, we perform such iterative skipping only on the smallest IDList \( L_1 \) until its current and previous nodes have no parent-child relationship.

In Algorithm 2, function \( \text{bwdGetCA} \) is called in each iteration to find an \( SLCA \) node from the \( m \) IDLists, which in turn calls \( \text{bwdBinSearch} \) to locate the matched node for \( C_k \). Note that in line 4.2 and 4.3 of \( \text{bwdBinSearch} \), the current CA node will be skipped if its ID is equal to that of the parent of the previous CA node.

Example 4 Consider Example 3 again. Our BwdSLCA Algorithm returns node 5 in the first iteration, which is output as an SLCA node immediately. Then by iteratively pruning the adjacent parent node (first node 4, then 3, 2, and 1), cursor \( C_1 \) will be moved beyond the first entry in \( L_1 \); hence, the algorithm stops without computing any other CA node. As shown by the blue solid arrows in Fig. 4b.

Time Complexity: The worst-case time complexity of our BwdSLCA algorithm is the same as that of FwdSLCA, that is, \( O(m \cdot |L_1| \cdot \log(|L_m|)) \).

5.3 Optimized backward solution

We can further optimize the performance of the backward solution by judiciously shrinking the search interval based on the structural relationship between nodes in an IDList.

### Algorithm 2: BwdSLCA(\( Q \))

```
1. parent ← {−1, −1} 
2. while (~bwdEof()) do 
3.   { flag, parent, v } ← bwdGetCA(parent)
4.   if (flag = TRUE) then output output v.ID as an answer 
5.       else break 
6.   foreach (i ∈ [1, m]) do bwdAdvance(C_i) 
7.   while (parent.ID = C_i.ID) do 
8.     parent ← L_{C_i.PIDPos}; bwdAdvance(C_i) 
9. end while 
10. end while 

Function bwdGetCA(parent) 
/*Only differences from fwdGetCA are shown here*/
1. \( z \) ← argmin \( C \), such that \( pos(C) = −1 \) then return TRUE 
2. else return FALSE 

Function bwdEof() 
/*Only differences from fwdEof are shown here*/
1. \( m \) ← 0; \( e \) ← pos(C)
2. if \( L_{C.m}.ID = u.ID \) then 
3.   if \( L_{C.m}.ID = parent.ID \) then 
4.     parent ← L_{C.m.PIDPos}; \( m \leftarrow m \) \( e \leftarrow m \) 
5.     else \( C_j ← L_{C.m} \)
6.   end if 
7.   if \( s > e \) then \( C_j ← L[v] \)
8.   return parent 
```

5.3.1 Reducing the search interval

Consider one iteration in the BwdSLCA algorithm where \( v \) is the current CA node and \( u \) is the parent node of \( v \). According to line 2 of function \( \text{bwdGetCA} \), the eliminator, \( C_k \), represents the node with the minimum ID value among the set of entries that precedes \( v \) in all IDLists. Hence, we have \( u \prec_d C_k \prec_d v \). Now the search interval for each probe operation can be reduced from \([0, pos(C)]\) to \([pos(u), pos(v)]\) in line 1 of function \( \text{bwdBinSearch} \). In fact, we can further reduce the search interval based on the following lemma.

Lemma 5 Consider the backward solution. Let

- \( v \) be the CA node returned from the current iteration,
- \( u \) be the parent node of \( v \),
- \( y \) be the first node that precedes \( v \) in \( L_i \),
- \( C_k \) be the eliminator used in the next iteration (hence satisfying \( u \prec_d C_k \prec_d y \prec_d v \)),
- \( w \) be the common ancestor of \( C_k \) and \( y \), and
- \( z \) be the child node of \( w \) (if it exists) on the path from \( w \) to \( y \).

Then we have \( u \prec_d w \prec_d C_k \prec_d y \prec_d v \), where the parenthesis part holds only if \( z \) exists.

Proof Since each probe operation involves two lists, we prove this result using \( L_1 \) and \( L_2 \). After processing \( v \), \( C_2 \) points to \( y \) and \( C_1 \) points to \( x \). Assume that \( x \prec_d y \), according to BwdSLCA, \( C_1 \), that is, \( x \), will be chosen in the next
iteration as the eliminator to find from $L_2$ the maximum node that is equal to or less than $x$. As shown in Fig. 5, there are three possible positional relationships between $x$ and $y$.

**Case A** $x = y$ (Fig. 5a). It consists of two sub-cases:

(A.1) $x = y = u$. In this case, all nodes between $u$ and $v$ in $L_2$ must be as $[u, v]$.

(A.2) $u <_d x$. In this case, the nodes between $u$ and $v$ in $L_2$ must be as $[u, ..., y, v]$, where $u <_d y <_d v$.

Therefore in Case A, $z$ does not exist and $C_1 = x = w = y$, thus we have $u \leq_d w \leq_d C_1 \leq_d y <_d v$.

**Case B** $x <_d y$ (Fig. 5b). In this case, all nodes between $u$ and $v$ is as $[u, ..., z, ..., y, v]$, where $z$ is the child node on the path from $x$ to $y$ and $u \leq_d x <_d z \leq_d y <_d v$. Since $C_1 = x = w$, we have $u \leq_d w \leq_d C_1 <_d z \leq_d y <_d v$.

**Case C** $x <_d y$ and $x \not<_d y$ (Fig. 5c). In this case, the nodes between $u$ and $v$ must be as $[u, ..., w, ..., z, ..., y, v]$, where $u <_d w <_d z <_d y <_d v$, $x$ satisfies that $w <_d x <_d z$. Since $C_1 = x$, we have $u \leq_d w \leq_d C_1 <_d z \leq_d y <_d v$.

By summarizing all the above three cases, we have $u \leq_d w \leq_d C_1 <_d z <_d y <_d v$. For $m$ IDLists, if the minimum node after processing $v$ is $C_k (k \in [1, m])$, then we have the same result, that is, $u \leq_d w \leq_d C_k <_d z <_d y <_d v$. \hfill \Box

According to Lemma 5, for each eliminator $C_k$, we set the search interval as $[\text{pos}(u), \text{pos}(z)]$, where $z = y$ if $C_k = y$. Obviously, this interval is not larger than $[\text{pos}(u), \text{pos}(v)]$.

5.3.2 The BwdSLCA$^+$ Algorithm

The optimized backward algorithm, that is, BwdSLCA$^+$, is the same as BwdSLCA except that in function bwdGetCA, bwdBinSearch is replaced by bwdBinSearch$^+$ (shown in Algorithm 3). The difference between them lies in line 1 marked with rectangle in bwdBinSearch, which is replaced by setInterval in bwdBinSearch$^+$. As shown in setInterval, line 1 corresponds to case (A.2) of Lemma 5 and Fig. 5a. Line 2–7 will iteratively compute the search interval, which correspond to cases (B) and (C) of Lemma 5. Note that case (A.1) of Lemma 5 is processed in line 7–9 of Algorithm 2.

![Diagram](image)

**Fig. 5** The positional relationships between nodes in an XML document, where each solid (dashed) line from $u$ to $v$ means $u \prec_p v$ ($u \preceq_p v$).

Algorithm 3: BwdSLCA$^+$ ($Q$)

/* The same as BwdSLCA except using bwdBinSearch$^+$ instead of bwdBinSearch$^+$/

**Function bwdBinSearch$^+$** ($L_j, u, \text{parent}$)

/* Only differences from bwdBinSearch are shown here */

1. $\{x, e\} \leftarrow \text{setInterval} (L_j, u)$

**Function setInterval** ($L_j, u$)

1. If $(C_j.ID = u.ID)$ then $(x \leftarrow \text{pos}(C_j); e \leftarrow x; \text{return} \{x, e\})$
2. $e \leftarrow \text{pos}(C_j)$
3. $x \leftarrow \text{pos}(L_j[C_j, \text{PIDPos}])$
4. While $(L_j[v] > u.ID)$ do
5. $(e \leftarrow x; x \leftarrow \text{pos}(L_j[v, \text{PIDPos}]))$
6. endwhile
7. return $\{x, e\}$

**Example 5** For query $Q = \{k_1, k_2\}$ and its two IDLists in Fig. 3, $C_1$ and $C_2$ point to nodes 16 and 20 initially. In the first iteration, BwdSLCA$^+$ will firstly use $C_1 = 16$ to probe $L_2$.

Since the parent of node 20 is node 15 and $15 \leq C_1 \leq 20$, we find the matched element of $C_1$ from the search interval $L_2[8, 12]$ (Fig. 6a), rather than $L_2[0, 12]$. After that, the SLCA node 15 is found, and after outputting node 15, in line 6, $C_1$ and $C_2$ point to 14 and 9, respectively. In the second iteration, $C_2 = 9$ is chosen as the eliminator to probe $L_1$ (Fig. 6b). Similarly, as the parent of node 14 is node 12 and $C_2 < 12$, we find the parent of node 12, which is node 10. In the same way, we further find the parent of node 10, which is node 8. Since $8 \leq C_2 \leq 10$, we set the search interval as $L_1[4, 5]$, rather than $L_1[0, 9]$, and find the matched element, that is, node 8. As node 8 is the parent of node 15, we directly move $C_1$ to node 4. As $C_1 = 4 < C_2 = 9$, in the third iteration, we use $C_1$ as the eliminator to probe $L_2$ (Fig. 6c). Similarly, we set the search interval as $L_2[0, 6]$, rather than $L_2[0, 7]$, to find the matched element of $C_1$, that is, node 3, then output node 3 as an SLCA result. After that, as nodes 1 and 2 are ancestors of node 3, and no other elements are located between them, our method directly skips the two nodes, then terminates the processing.

**Time Complexity:** The worst-case time complexity of our BwdSLCA$^+$ algorithm is the same as that of BwdSLCA, that is, $O(m \cdot |L_1| \cdot \log(|L_m|))$.

6 ELCA computation

6.1 Forward solution

As $ELCA(Q) \subseteq CA(Q)$, we can still use the fwdCA function of Algorithm 1 to find all CA nodes first. According to Lemma 3, it is not possible for us to know whether a CA node $v$ is an ELCA node until we have processed all its descendant CA nodes.

To facilitate identifying all ELCA nodes, a stack $S$ is used to check whether the popped element is an ELCA node. Besides the ID value, each element $e$ of $S$ is associated
Reducing the search intervals for different probe operations

with two vectors, one is \( N = \langle N_1, N_2, \ldots, N_m \rangle \) denoting the \( N_{D\text{ searches}} \) values of all keywords in \( Q \), the other is \( n = \langle n_1, n_2, \ldots, n_m \rangle \) used to online add the \( N \) value of \( e \)'s child nodes in the CA tree. According to Lemma 3, when \( e \) is popped from \( S \), if \( 2i \in [1, m] \), such that \( e.N_i = e.n_i \), then \( e \) is an ELCA node. The main procedure can be stated as: whenever finding a CA node \( v \), we firstly pop from \( S \) all CA nodes that are not parent of \( v \), then push \( v \) to \( S \). For each popped node \( w \), we firstly add \( w \)'s \( N \) vector to the \( n \) vector of the top element of \( S \), then check \( w \)'s satisfiability according to Lemma 3. As shown in Algorithm 4, in each iteration (line 1–12), fwdGetCA is called to find a CA node in line 2, if \( \text{flag} = \text{FALSE} \), it means that all CA nodes are found, then we stop the processing (line 10) and pop out each element of \( S \) to check their satisfiability (line 13–17). If \( S \) is empty or the top element of \( S \) is the parent node of the current CA node, we directly push it into \( S \) (line 9); otherwise, if \( S \) is not empty and the top element of \( S \) is not the parent of the current CA node \( v \), we need to pop out from \( S \) all elements that are not parent of \( v \) (line 5). For each element \( w \) popped out from \( S \), we firstly modify \( n \)'s value of the top element of \( S \) in line 6, then check whether \( w \) is an ELCA by calling function isELCA, and output \( w.ID \) as an ELCA node if isELCA returns \( \text{TRUE} \) (line 7). Note that when each element is pushed into stack \( S \), all its \( n_t \) values are initialized to 0.

**Example 6** For query \( Q = \{k_1, k_2\} \) and its IDLists in Fig. 3, as shown in Fig. 7a–c, since node 1 is the parent of node 2, which in turn is the parent of node 3, the three CA nodes, that is, nodes 1, 2, and 3, are pushed into stack \( S \). The next CA node returned from fwdGetCA is node 8. Since nodes 3 and 2 are not the parent of node 8, they are popped out from \( S \) one by one; after that, we modify the \( n \) value of the top element of \( S \) by adding up the \( N \) value of popped elements, as shown in Fig. 7d, e; then, we check the satisfiability of nodes 3 and 2 according to Lemma 3, and output node 3 as an ELCA node. Finally, nodes 8 and 15 are pushed into \( S \) (Fig. 7f, g). After that, all elements are popped out from \( S \) and the algorithm stops. After each element is popped out from \( S \), we check its satisfiability and add its \( N \) vector to \( n \) vector of the top element in \( S \). Note that, nodes 1 and 2 are not ELCA nodes, since each \( N_t \) of nodes 1 and 2 is equal to their \( n_t \) (see Fig. 7d, i). Therefore, the set of ELCA results are nodes 3, 8, and 15.

**Time Complexity:** In the worst case, FwdELCA processes at most \(|L_1| \) nodes. For each node, the cost of checking each node in other lists is \( O(m - 1) \cdot \log |L_m|) \). Each element is pushed into and popped out from the stack at most once, the cost of processing it in line 4 to line 9 is \( O(m) \). Therefore,
the worst-case time complexity of FwdELCA is the same as that of FwdSLCA, that is, $O(m \cdot |L_1| \cdot \log(|L_m|))$.

6.2 Backward solution

BwdELCA computes all CA nodes in the reverse document order and tries to utilize the information of CA nodes that are located after the current one to make optimization. In this way, when a CA node $v$ is processed, all its descendant CA nodes must have been processed already; thus, we know its $n$ value. Since any non-leaf CA node $v$ could be an ELCA node, we need to know its $N$ value to determine whether it is an ELCA node. Therefore, for each CA node, we need to locate its position in each IDList to fetch its $N_{Desc}$ value, which is different from the BwdSLCA algorithm that can avoid probing other lists when meeting a non-leaf CA node.

The core operation for ELCA computation in reverse document order is how to transfer $N$’s value of a CA node to its parent node, since the parent $u'$ of the previous CA node $v'$ may not satisfy $u' \lesssim_p v'$, where $v$ is the current CA node. Fortunately, Lemma 6 can be used to simplify the checking.

**Lemma 6** Assume that all CA nodes are computed in reverse document order. Let $v'$ be the CA node returned from the previous iteration, $u'$ be the parent of $v'$, and $v$ be the CA node of the current iteration, then $u' \lesssim_d v \prec_d v'$.

**Proof** As shown in Fig. 8, there are three kinds of positional relationships for $u'$, $v'$, $u$, and $v$: Case 1 (Fig. 8a), $u'$ has no other descendant CA nodes except $v'$, thus the next CA node must be $u'$ itself, that is, $u' = v \prec_d v'$. Case 2 (Fig. 8b), $v$ is a leaf node of the CA tree and is $u'$’s first child node before $v'$, thus $u' \prec_p v \prec_d v'$. Case 3 (Fig. 8c), $v$ is a leaf node of the CA tree and is $u'$’s last descendant node before $v'$. By summarizing the above cases, we have $u' \lesssim_d v \prec_d v'$. □

According to Lemma 6, in the BwdELCA algorithm, we use a stack $S$ to online buffer CA nodes that are not met, but at least one of their child CA nodes have been processed.

The main idea of the BwdELCA algorithm is: whenever finding a CA node $v$, push its parent $u$, rather than itself, into the stack according to the positional relationships between $v$, $u$, and $u'$, where $u'$ is the parent of the previous CA node.

As shown in Algorithm 5, in each iteration (line 1–25), bwdGetCA is called to find a CA node $v$ in line 2. If $flag =$ FALSE, it means that all CA nodes have been founded, then we stop the processing (line 23); otherwise, if $S$ is empty (line 4–7), it means that $v$ is the first CA node found in reverse document order, then it must be a leaf node of the CA tree and therefore is an ELCA node; thus, we output it in line 5. If $v$ is the root node (line 6), we stop the processing; otherwise, the parent node of $v$ is pushed into $S$ in line 7. If $S$ is not empty, we process $v$ according to its position. If $v$ equals the top element of $S$ (line 9, Fig. 8a), it means that all descendant CA nodes of $v$ has been processed, then we need to pop $v$ from $S$ (line 10), and check whether it is an ELCA node (line 11). After that, we process $v$ in three cases:

1. If $S$ is empty and $v$ is the root node, we stop the processing in line 13; otherwise, push $v$’s parent into $S$ in line 14.
2. If the top element of $S$ equals $v$’s parent, we add up the $N$ value of $v$ to the top element of $S$ in line 15.
3. If the parent of $v$ is a descendant of the top element of $S$, we directly push $v$’s parent into $S$ in line 16.

If $v$ does not equal the top element of $S$ (line 18-20), it means that $v$ is a leaf node of the CA tree, then it must be an ELCA node and we output it in line 18. After that, if $v$ is a child node of the top element of $S$, which corresponds to the case of Fig. 8b, we add up the $N$ value of $v$ to the top element of $S$ in line 19; otherwise (the case of Fig. 8c), we directly push $v$’s parent into $S$ in line 20. Note that, in the BwdELCA algorithm, we use the optimization technique proposed in Sect. 5.3 to accelerate the locating of a node in other lists as shown in function bwdBinSearch.

**Example 7** For query $Q = \{k_1, k_2\}$ and its two IDLists in Fig. 3, the processing is shown in Fig. 9a-d. The first CA node is 15, which corresponds to Fig. 8a; thus, 15 is output as an ELCA node, and its parent, that is, node 8, is pushed into stack $S$ (Fig. 9a). The next CA node is node 8, which corresponds to Fig. 8a; thus, node 8 is popped out from $S$. Since node 8 is an ELCA node, it is output as an ELCA node, then its parent, that is, node 1, is pushed into $S$ (Fig. 9b). The third CA node is 3, which corresponds to Fig. 8c, thus node 3 is a leaf node of the CA tree, and it is output as an ELCA node, then its parent, that is, node 2, is pushed into $S$ (Fig. 9c). The next CA node is 2, which corresponds to Fig. 8a, thus node 2 is popped from the stack. Since node 2 is not an ELCA node, we directly add its $N$ value to node 1. The last
Algorithm 5: BwdELCA(Q)

```plaintext
1 while (~ bwdEnd(Li)) do
2    { flag, parent, v } ← bwdGetCA();
3    if (flag = TRUE) then
4        if (isEmpty(S)) then
5            output v.ID as an answer
6        if (parent.ID = −1) then break
7        pushParent(S, parent, v)
8        else
9            if (v.ID = top(S).ID) then
10               w ← pop(S)
11               if (isELCA(w)) then output w.ID as an answer
12               else
13                   if (parent.ID = −1) then break
14                   else pushParent(S, parent, v)
15                   else if (parent.ID = top(S).ID) then modTop(S, v)
16                   else pushParent(S, parent, v)
17        else
18            output v.ID as an answer
19            if (parent.ID = top(S).ID) then modTop(S, v)
20        else pushParent(S, parent, v)
21    endif
22    endforeach (i ∈ [1..m]) do bwdAdvance(Ci)
23 endwhile

Function pushParent(S, parent, v)
1    foreach (i ∈ [1..m]) do parent.ni ← v.Ni
2    push(S, parent)
```

7 Combining IDList with hash search

All algorithms in Sects. 5 and 6 rely on the probe operation (implemented by binary search) to align the cursors of inverted lists, hence leading to a $O(m \cdot |L_1| \cdot \log(|L_m|))$ term in the time complexity. To further improve the overall performance, we consider the existence of additional hash indexes [31] on inverted lists such that each probe operation takes $O(1)$ time.

Specifically, we maintain two kinds of hash tables. As shown in Fig. 10, for each keyword $k_i$, the first records the number of IDs in its IDList, which is used to choose the shortest IDList. For each IDList $L_i$, the second hash table records, for each ID $id_v \in L_i$, the number of descendant nodes that directly contain $k_i$ in the subtree rooted at $v$. Obviously, the second hash table maintains the same information as that in $N_{desc}$ row of our IDList and can be used to check the existence of a keyword below a node, and check the frequency of a keyword appearing in a subtree. Note that, the hash tables used in our methods are similar to that of [31], except that in [31], $H_{Freq}$ records, for each keyword $k_i$, the length of its inverted list of ID Dewey labels, that is, $|L_i^D|$.

According to Fig. 3, we know that the length of $L_1$ is 12, which can be denoted as $12 = H_{Freq}[k_1]$ according to Fig. 10a. According to Fig. 3, $k_1$ appears four times in the subtree rooted at node 8, which can be denoted as $4 = H_{k_1}[8]$ (Fig. 10b). Similarly, node 4 does not contain $k_2$ in its subtree, which can be denoted as $4 \notin H_{k_2}$ (Fig. 10c).

7.1 The forward solution on SLCA computation

According to Property 1, $CA(Q) \subseteq L_1$. Thus, we can get all CA results based on $L_1$ by transforming each probe operation on other lists into that on hash tables. The main idea is: take the shortest list $L_1$ as the working list, sequentially check whether each node of $L_1$ is a CA node in document order, and output the previous CA node if it is an SLCA result.

7.1.1 Processing useless nodes

Although a naive implementation based on the above idea does not suffer from the CAR problem, the number of hash probe operations is linear to the length of $L_1$. In practice, when the number of CA nodes is much less than the length of $L_1$, most hash probe operations are wasted on non-CA nodes. As discussed in Sect. 5 and 6, we can skip many useless nodes by comparing the positional relationship of nodes in different lists. When the number of involved lists is 1, that is, $L_1$, no other lists can be used to facilitate the skipping operation. The only way is utilizing the positional relationship of nodes in $L_1$ to skip useless nodes. Fortunately, we have the following observation.
Observation 1: For a given keyword query \( Q \) and an XML node \( v \), if \( v \) is not a CA node of \( Q \), then anyone of \( v \)'s descendant nodes cannot be a CA node.

According to the above observation, whenever finding a node \( u \) in \( L_1 \) that is not a CA node, for all descendant nodes of \( u \) in \( L_1 \), we do not need to waste hash probe operations on them. Then a question is: how to determine whether \( u \) is an ancestor node of \( v \) located after \( u \) in \( L_1 \)? Obviously, this can be determined by the "PIDPos" of \( v \) and that of its ancestor nodes until we reach \( u \) from \( v (u \prec_a v) \), or otherwise, reach a node that is located before \( u \) (\( u \not\prec_a v \)). Even though it is feasible, the cost of checking ancestor-descendant (AD) relationship is \( O(d) \). Instead, we check the AD relationship of \( u \) and \( v \) with cost \( O(1) \) based on the following lemma.

Lemma 7 Given a query \( Q = \{k_1, k_2, \ldots, k_m\} \) and one of its IDList \( L_i (i \in [1, m]) \), assume that \( y \) is the first node after \( u \) in \( L_i \), such that \( y.PIDPos \leq u.PIDPos \), then each node \( v \) between \( u \) and \( y \) is a descendant of \( u \).

Proof Since \( u \prec_d v \prec_d y \), we know \( y \not\prec_a v \). There are three kinds of relationships for nodes between \( u \) and \( y \).

Case 1 (Figure 11a, \( x \prec_p u \), \( y \) is the next sibling node of \( u \), that is \( y.PIDPos = u.PIDPos = p_x \)). If \( u \not\prec_a v \), then there must exist one of \( v \)'s ancestor node \( v' \) that is the next sibling node of \( u \), that is, \( u \prec_d v' \prec_d y \) and \( u.PIDPos = v'.PIDPos = y.PIDPos = p_x \). Therefore, \( y \) is not the first node after \( u \), such that \( y.PIDPos \leq u.PIDPos \), which contradicts the assumption.

Case 2 (Figure 11b, \( z \prec_p x \), \( u \) is the last child node of \( x \), \( y \) is the next sibling node of \( x \), \( y.PIDPos = p_z \prec u.PIDPos = p_x \)). If \( u \not\prec_a v \), there must exist one of \( v \)'s ancestor node \( v' \) that is the next sibling node of \( x \), then \( u.PIDPos = p_x \prec v'.PIDPos = y.PIDPos = p_z \). Therefore, \( y \) is not the first node after \( u \), such that \( y.PIDPos \leq u.PIDPos \), which contradicts the assumption.

Case 3 (Figure 11c, \( q \prec_a z \not\prec_q \prec_p y \), \( x \) is the last child node of \( z \), \( u \) is the last child node of \( x \), \( y \) is the next sibling node of one of \( u \)'s ancestors, \( y.PIDPos = p_q \prec u.PIDPos = p_z \)). If \( u \not\prec_a v \), then there must exist one of \( v \)'s ancestor node \( v' \), such that \( q \prec_a v' \) and \( v' \) is the next sibling node of one of \( u \)'s ancestor nodes. Therefore, \( u.PIDPos = p_x \succ v'.PIDPos \geq y.PIDPos = p_q \), and \( y \) is not the first node after \( u \), such that \( y.PIDPos \leq u.PIDPos \), which contradicts the assumption.

By summarizing the above cases, \( u \prec_a v \).

As the cost of each hash probe operation is much more expensive than that of a comparison operation between two integers, whenever finding a non-CA node \( u \) from \( L_1 \), we do not need to waste hash probe operations on each next node \( v \) until \( u.PIDPos \leq v.PIDPos \).

7.1.2 The FwdSLCA-HS Algorithm

The FwdSLCA-HS algorithm tries to transform as many hash probe operations on useless nodes into less number of comparison operations as possible. As shown in algorithm 6, \( u \) denotes the previous CA node, \( parPos \) denotes, after the current CA node, the position value of the parent of first non-CA node. In each iteration (line 3–17), it picks a node \( C_1 \) and records the position value of \( C_1 \)'s parent in line 4, and then checks whether \( C_1 \) is a CA node by calling Function isCA(\( C_1, CID \)) in line 5. If \( C_1 \) is a CA node, our algorithm further checks whether \( u \) is the parent of \( C_1 \). If \( u \) is not the parent of \( C_1 \), it means that \( u \) is a leaf CA node of the CA tree. According to Lemma 2, we output \( u \) as an SLCA result in line 7. In line 9, we buffer the current CA node \( C_1 \) as a candidate SLCA node in \( u \), then advance \( C_1 \) (line 10) before the next iteration. If \( C_1 \) is not a CA node, then all nodes in the subtree rooted at \( C_1 \) are not CA nodes. According to Lemma 7,
the descendant nodes of \( C_1 \) are processed without hash probe operations in line 12–15.

**Example 8** For query \( Q = \{ k_1, k_2 \} \), our method takes \( L_1 \) as the working list since \(|L_1| = 12 < |L_2| = 13 \). According to Algorithm 6, it will first find three CA nodes, that is, nodes 1, 2, and 3, then find a non-CA node, that is, node 4. As node 4 does not have other non-CA nodes in \( L_1 \), the next iteration will find node 8 as a CA node. Since node 3 is not the parent of node 8, we output node 3 as an SLCA node. After that, our method processes node 10, which is not a CA node. According to Lemma 7, our method records the position value of its parent with sequentially processes each node after node 10 by comparing the position value of its parent with \( \text{par Pos} \). If \( \text{par Pos} < C_1.PIDPos \), we directly move \( C_1 \) to the next node. In such a way, nodes 11, 12, 13, and 14 are processed without hash probe operations, because they are descendant nodes of node 10. The next CA node is node 15, which is also the last CA node; thus, our method outputs it as an SLCA result.

### 7.2 The backward solution on SLCA computation

Compared with FwdSLCA, even though FwdSLCA-HS removes the log \( |L_m| \) factor from its time complexity, it still needs to afford the cost of probing the hash table \( m - 1 \) times for each CA node. Similar to the BwdSLCA algorithm, if we compute all CA nodes in reverse document order, we can avoid the cost of probing the hash table for non-leaf CA nodes. For all non-CA nodes, we try to skip as many useless nodes as possible based on Lemma 8.

**Lemma 8** Given query \( Q \), assume that node \( u \) is the parent of node \( v \). If both \( u \) and \( v \) are non-CA nodes, then any node \( w \) satisfying \( u \prec_d w \prec_d v \) cannot be a CA node of \( Q \).

---

**Algorithm 6: FwdSLCA-HS (Q)**

```plaintext
\text{FwdSLCA-HS}(Q) \triangleright Q = \{k_1, \ldots, k_m\}, 0 < |L_1| \leq |L_2| \leq \ldots \leq |L_m| */
1 \text{ u ← } \{-1, -1\} /*u denotes the previous CA node */
2 \text{ par Pos ← -1}
3 \text{ while } (\text{not eof}(L_1)) \text{ do}
4 \text{ par Pos ← C_1.PIDPos}
5 \text{ if } (\text{isCA}(C_1.ID)) \text{ then}
6 \text{ output } u.ID \text{ as an answer}
7 \text{ endwhile}
8 \text{ u ← C_1)
9 \text{ fwpAdvance}(C_1)
10 \text{ else}
11 \text{ fwpAdvance}(C_1)
12 \text{ while } (\text{par Pos} < C_1.PIDPos) \text{ do}
13 \text{ fwpAdvance}(C_1)
14 \text{ endwhile}
15 \text{ output } u.ID \text{ as an answer}
16 \text{ Function isCA(id_1)}
17 \quad \text{1 if } (\exists k \in Q, \text{ such that } id_1 \neq H_k) \text{ then return } FALSE
18 \quad \text{2 return TRUE}
```

**Algorithm 7: BwdSLCA-HS (Q)**

```plaintext
\text{BwdSLCA-HS}(Q) \triangleright Q = \{k_1, \ldots, k_m\}, |L_1| \leq |L_2| \leq \ldots \leq |L_m| */
1 \text{ u ← } \{-1, -1\} /*u denotes a non-leaf CA node */
2 \text{ prePos ← L_1; cur Pos ← L_1)
3 \text{ while } (\text{not eof}(L_1)) \text{ do}
4 \text{ if } (u \neq C_1 \land \text{cur Pos} \neq \text{pos}(C_1) \land \neg \text{isCA}(C_1.ID)) \text{ then}
5 \text{ prePos ← pos}(C_1)
6 \text{ C_1 ← L_1[C_1.PIDPos]
7 \text{ else}
8 \text{ cur Pos ← pos}(C_1)
9 \text{ if } (\text{cur Pos} = \text{prePos} - 1) \text{ then}
10 \text{ if } (u \neq C_1) \text{ then}
11 \text{ output } C_1.ID \text{ as an answer}
12 \text{ endif}
13 \text{ u ← L_1[C_1.PIDPos]
14 \text{ prePos ← cur Pos}
15 \text{ C_1 ← L_1[par Pos - 1]
16 \text{ else}
17 \text{ C_1 ← L_1[par Pos - 1]
18 \text{ endif}
19 \text{ endif}
20 \text{ endwhile}
```

We omit the proof of Lemma 8 as its correctness is obvious. According to Lemma 8, when processing all nodes of \( L_1 \) in reverse document order, we can directly skip all nodes between a node \( v \) and its parent \( u \), if \( u \) is not a CA node. The main idea is: compute all CA nodes in reverse document order, and output all leaf CA nodes as SLCA results.

As shown in Algorithm 7, \( u \) denotes a non-leaf CA node, which is the parent of the CA node \( v \) identified most recently. \( \text{cur Pos} \) always points to \( v \), and \( \text{prePos} \) always points to one of \( v \)’s child node that is processed just before \( v \). Initially, \( u \) points to a virtual node before the root node (line 1), and both \( \text{cur Pos} \) and \( \text{prePos} \) point to a virtual node after the last node of \( L_1 \) (line 2).

In each iteration (line 4–19), we process a node according to the following cases:

**Case 1** \( C_1 \) points to a non-CA node (line 4 returns TRUE). In this case, we directly make \( \text{prePos} \) point to \( C_1 \) (line 5), and move \( C_1 \) to its parent node (line 6).

**Case 2** \( C_1 \) points to a non-leaf CA node \( u = C_1 \). In this case, \( C_1 \notin SLCA(Q) \), and checking whether \( C_1 \) is a CA node is unnecessary. We process \( C_1 \) in two sub-cases:

**Case 2.1** \( \text{cur Pos} = \text{prePos} - 1 \) (line 9), it means that all descendant nodes of \( C_1 \) have been processed. We simply move \( u \) to its parent (line 13), and move \( C_1 \) forwardly to the node before it (line 15).

**Case 2.2** \( \text{cur Pos} \neq \text{prePos} - 1 \), it means that some nodes between \( C_1 \) and \( L_1[\text{prePos}] \) have not been processed; thus, we move \( C_1 \) to the node before \( \text{prePos} \) (line 17).

**Case 3** \( C_1 \) points to the CA node identified most recently (pointed by \( \text{cur Pos} \); \( \text{cur Pos} = \text{pos}(C_1) \)). In this case, we process \( C_1 \) in two sub-cases:
Case 3.1 \( \text{cur Pos} \neq \text{prePos} - 1 \) (line 9 returns FALSE), it means that some nodes between \( C_1 \) and \( L_1[\text{prePos}] \) have not been processed. We just move \( C_1 \) to the node before \( \text{prePos} \) (line 17).

Case 3.2 \( \text{cur Pos} = \text{prePos} - 1 \) (line 9). In this case, if \( u \neq C_1 \) (line 10 returns TRUE), it means that \( C_1 \) is a leaf CA node, thus we directly output \( C_1.ID \) as an SLCA result; otherwise, we move \( u \) to its parent node (line 13), and move \( C_1 \) forwardly to the node before it (line 15).

Case 4 \( C_1 \in \text{CA}(Q) \land u \neq C_1, \) and \( C_1 \) is not equal to a previously identified CA node (\( \text{cur Pos} \neq \text{pos}(C_1) \)). We firstly set \( \text{cur Pos} = \text{pos}(C_1) \) in line 8, then check whether all descendant nodes of \( C_1 \) have been processed (line 9). If there are still some descendant nodes of \( C_1 \) have not been processed (line 9 returns FALSE), we simply move \( C_1 \) forwardly to the node before \( \text{prePos} \) (line 17); otherwise (line 9 returns TRUE), we simply output \( C_1.ID \) as an SLCA result, then move \( u \) to its parent node (line 13), and move \( C_1 \) forwardly to the node before it (line 15).

Example 9 For query \( Q = \{k_1, k_2\} \), BwdSLCA-HS takes \( L_1 \) as the working list (see Fig. 3) and processes all nodes in reverse document order. Initially, \( u = (-1, -1), \) \( \text{prePos} = \text{cur Pos} = 12, \) \( C_1 \) points to node 16. In the first iteration (Fig. 12a, Case 1), as node 16 is not a CA node, it moves \( C_1 \) to its parent node, that is, node 15 (line 6). In the second iteration (Fig. 12b, Case 4), it finds that node 15 is a CA node. Since both line 9 and line 10 return TRUE, it outputs node 15 as an SLCA result (line 11), then \( u \) points to node 8, \( \text{prePos} = 10, \) and \( C_1 \) points to node 14. The next three iterations (Fig. 12c–e, Case 1) are similar to the first iteration. After that, \( C_1 \) points to node 8 and \( \text{prePos} \) points to node 10. In the 6th iteration (Fig. 12f, Case 2.1), since \( u = C_1, \) it makes \( u \) point to node 1, the parent of node 8 (line 13), and it makes \( \text{prePos} \) point to node 8 (line 14) and \( C_1 \) point to node 4 (line 15). In the 7th iteration (Fig. 12g, Case 1), since node 4 is not a CA node, \( \text{prePos} \) points to node 4 and \( C_1 \) points to node 3. As node 3 is a CA node and \( u \neq C_1, \) in the 8th iteration (Fig. 12h, Case 4), it outputs node 3 as an SLCA result, and makes \( u \) point to node 2, the parent of node 3 (line 13), \( \text{prePos} \) point to node 3 (line 14) and \( C_1 \) point to node 2 (line 15). The next two iterations (Fig. 12i, j, Case 2.1) are processed in the same way as the 6th iteration (Fig. 12f). After that, the processing stops. The SLCA results are node 15 and node 3.

7.3 The hybrid solution on SLCA computation

Obviously, FwdSLCA-HS is eager in transforming hash probe operations on non-CA nodes into comparison operations as many as possible, while it suffers from redundant hash probe operations on non-leaf CA nodes. Besides, it needs to process all nodes that are not involved with hash probe operations one by one. On the contrary, BwdSLCA-HS avoids processing all non-leaf CA nodes using hash probe operations; however, it may suffer from much more hash probe operations than FwdSLCA-HS on non-CA nodes, even though in some cases, it can skip many useless nodes.

Example 10 For query \( Q = \{k_1, k_2\} \), assume that the XML tree consisting of nodes in \( L_1 \) is the one shown in Fig. 13, where node 1 to 3 are CA nodes, others are non-CA nodes. Obviously, if we use FwdSLCA-HS, only five nodes are processed with hash probe operations, that is, nodes 1, 2,
3, 4, and 8, and nodes 5, 6, 7, 9, 10, 11, and 12 are processed without hash probe operations. Even though nodes 1 and 2 are not SLCA nodes, they are still processed by hash probe operations. On the contrary, if we use BwdSLCA-HS, four nodes are skipped without being processed, that is, nodes 6, 9, 10, and 11, and the two CA nodes, that is, nodes 1 and 2, are processed without hash probe operations. However, it still needs to process nodes 3, 4, 5, 7, 8, and 12 by hash probe operations, which is not as efficient as FwdSLCA-HS.

To take advantage of the benefits of both FwdSLCA-HS and BwdSLCA-HS, while at the same time, avoid their weaknesses, we propose the hybrid algorithm, namely HybSLCA-HS, which processes nodes of L1 in reverse document order, so as to skip as many useless nodes as possible, while at the same time, processes some nodes in document order, so as to avoid costly hash probe operations.

Let v \in L1 be the node that has been just processed, u is the parent of v, HybSLCA-HS processes all nodes between u and v in three cases:

**Case 1** v \in CA(Q). If there is no other nodes between u and v, we directly move to the node before u; otherwise, choose the node before v for the next iteration.

**Case 2** v \notin CA(Q) \land u \in CA(Q). We process all nodes between u and v using FwdSLCA-HS.

**Case 3** v \notin CA(Q) \land u \notin CA(Q). We directly skip all nodes between u and v as BwdSLCA-HS does.

We omit the detailed description of HybSLCA-HS due to limited space. Example 11 illustrates the difference of HybSLCA-HS when processing Q = \{k1, k2\} on the XML document in Fig. 13.

**Example 11** Continue Example 10. HybSLCA-HS first processes node 12, which is not a CA node, then HybSLCA-HS checks whether its parent, that is, node 8, is a CA node. As node 8 is not a CA node (Case 3), HybSLCA-HS directly skips nodes 9 to 11. As the parent of node 8 is node 3, which is a CA node (Case 2), we process all nodes between nodes 3 and 8 using FwdSLCA-HS. Therefore, only node 4 is processed using hash probe operations. After that, nodes 2 and 1 are processed without hash probe operations (Case 1).

### 7.4 Comparison of our hash-based algorithms

We firstly classify all nodes of L1 into the following mutually disjoint subsets.

- S1: the set of non-leaf CA nodes in the CA tree.
- S2: the set of leaf CA nodes (SLCA nodes).
- S3: the set of non-CA nodes, of which each one is a child of a CA node.
- S4: the set of non-CA nodes, of which each one is not a child of any CA node, which can be further classified into two subsets.

**S41**: each node v \in S41 is in a subtree t_u, u \in S3 is the rightest node among its sibling nodes.

**S42**: the set of nodes of S4 after excluding S41.

**S43**: each node v \in S43 is in a subtree t_u, u \in S3 is the rightest node among its sibling nodes.

**S44**: the set of nodes of S4 after excluding S43.

Obviously, L1 = S1 \cup S2 \cup S3 \cup S41 \cup S42 \cup S43 \cup S44 and |L1| = |S1| + |S2| + |S3| + |S41| + |S42| + |S43| + |S44|.

**Example 12** Consider the XML tree in Fig. 13, which consists of all nodes in L1 of Q = \{k1, k2\}. According to the above definition, S1 = \{1, 2\}, S2 = \{3\}, S3 = \{4, 8\}, S4 = \{5, 6, 7, 9, 10, 11, 12\}, S41 = \{5, 7, 12\}, S42 = \{12\}, S43 = \{5, 7\}, S44 = \{6, 9, 10, 11\}, S45 = \{9, 10, 11\}, S46 = \{6\}.

Table 2 shows the comparison of our three hash-based algorithms. Obviously, even though FwdSLCA-HS needs to process all nodes, all nodes of S4 are processed without hash probe operations. Thus, when |L1| \gg \sum_{i=1}^{3} |S_i|, the overall performance of FwdSLCA-HS is dominated by the number of nodes of S4; otherwise, its overall performance is dominated by the number of nodes in \bigcup_{j=1}^{3} S_j.

BwdSLCA-HS, on the other hand, does not need to process all nodes of L1. It tries to skip as many useless nodes...
as possible to improve the overall performance. Compared with FwdSLCA-HS, BwdSLCA-HS skips all nodes of $S_1$ from being processed and transforms the hash probe operations of FwdSLCA-HS on nodes of $S_1$ into comparison operations between two integers. However, it also transforms the comparison operations of FwdSLCA-HS on nodes of $S_4$ into costly hash probe operations.

As a comparison, HybSLCA-HS is more adaptive to various cases on average. When $|L_1| \gg |\bigcup_{i=1}^4 S_i|$, compared with BwdSLCA-HS, HybSLCA-HS does not need to afford the cost of processing all nodes of $S''_4$ using hash probe operations; and compared with FwdSLCA-HS, it can be as clever as BwdSLCA-HS to skip all nodes of $S''_4$. When $|S_4| \ll |\bigcup_{i=1}^4 S_i|$, compared with FwdSLCA-HS, HybSLCA-HS does need to process all nodes of $S_1$ using hash probe operations.

Obviously, the performance of the three hash-based algorithms are dominated by two factors: (1) the number of nodes that are processed with hash probe operations and (2) the number of nodes that are processed without hash probe operations. Therefore, the worst-case time complexities of FwdSLCA-HS, BwdSLCA-HS, and HybSLCA-HS are $O(m \cdot |\bigcup_{i=1}^4 S_i| + |S_4|)$, $O(m \cdot |S_2| + |S_3| + |S_4| + |S_1|)$ and $O(m \cdot |S_2| + |S_3| + |S''_4| + |S_1| + |S''_4| + |S'_4|)$, respectively.

Note that, for SLCA computation, if we maintain the parent’s positional value for each ID in the hash table, then the IDList can be largely simplified by just maintaining the ID value for all nodes.

### 7.5 ELCA computation

ELCA computation based on IDList and hash table can be easily implemented based on FwdSLCA-HS, BwdSLCA-HS, and HybSLCA-HS, which are called as FwdELCA-HS, BwdELCA-HS, and HybELCA-HS, respectively. We omit them in this paper and only present their experimental results in Sect. 9. Note that, as each non-leaf CA node $v$ could be a possible ELCA node, we need to know the number of occurrences of each keyword in the subtree rooted at $v$.

### 8 Comparison with existing methods

Compared with existing methods that do not use hash table in Table 1, the time complexity of Stack and DIL is $O(d \cdot m \cdot (|L_i^D|))$, while the time complexity of IL, IS, IMS, and JDewey is $O(d \cdot m \cdot |L_i^D| \cdot \log |L_m^D|)$. Since $|L_i| \leq d \cdot |L_i^D|$ due to the sharing of common ancestors in IDList, our algorithms, that is, FwdSLCA, BwdSLCA, BwdSLCA+$^+$, FwdELCA, and BwdELCA, do not suffer from the CAR problem anymore. Moreover, BwdSLCA, BwdSLCA+$^+$, and BwdELCA can utilize the optimization techniques proposed in Sect. 5 and 6 to accelerate the overall performance.

Compared with existing methods that use hash table in Table 1, that is, HS and HC, according to their time complexities, that is, $O((\log d \cdot m + d) \cdot |L_i^D|)$ and $O(d \cdot m \cdot |L_i^D|)$, the overall performance of our methods is not dominated by the length of $L_i^D$. On the contrary, our algorithms, that is, FwdSLCA-HS, BwdSLCA-HS, HybSLCA-HS, FwdELCA-HS, BwdELCA-HS, and HybELCA-HS, do not suffer from the CAR problem due to the same reason, that is, sharing of common ancestors in IDList. And according to the time complexities of our methods (see Sects. 7.4 and 7.5), the overall performance of our methods is not dominated by the length of $L_1$ anymore.

Note that for a given query $Q$, according to Lemma 1, $LCA(Q) \subseteq CA(A)$. As all our methods can easily identify all CA nodes, for other LCA-based semantics [7,14,16], if additional indexes in their methods that are used to process the specific conditions w.r.t. the corresponding semantics are constructed in advance, our methods can also easily support them with minor adaption. We omit the detailed discussion, since no other technical problems are needed to be solved.

### 9 Experiment

#### 9.1 Experimental setup

All experiments were run on a PC with Pentium(R) Dual-Core E5300 2.6GHz CPU, 2GB memory, 500GB IDE hard disk, and Windows XP Professional OS.

We considered three groups of algorithms:

Table 2 Comparison of our hash-based algorithms, $\triangleright$ corresponds to hash probe operations, $\triangleright\triangleright$ corresponds to comparison operations between integers, $\leftarrow$ means that the corresponding nodes are skipped without being processed.

<table>
<thead>
<tr>
<th>Node set</th>
<th>FwdSLCA-HS</th>
<th>BwdSLCA-HS</th>
<th>HybSLCA-HS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$\triangleright$</td>
<td>$\triangleright\triangleright$</td>
<td>$\triangleright\triangleright$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$\triangleright$</td>
<td>$\triangleright$</td>
<td>$\triangleright$</td>
</tr>
<tr>
<td>$S_3$</td>
<td>$\triangleright$</td>
<td>$\triangleright$</td>
<td>$\triangleright$</td>
</tr>
<tr>
<td>$S_4$</td>
<td>$S''_4$ $\triangleright$</td>
<td>$\triangleright\triangleright$</td>
<td>$\triangleright\triangleright$</td>
</tr>
<tr>
<td></td>
<td>$S''_4$ $\triangleright$</td>
<td>$\triangleright\triangleright$</td>
<td>$\triangleright\triangleright$</td>
</tr>
</tbody>
</table>

Therefore, except FwdELCA-HS which has the same time complexity as FwdSLCA-HS, the worst-case of BwdELCA-HS and HybELCA-HS are $O(m \cdot |S_1 \cup S_2 \cup S_3 \cup S_4|)$ and $O(m \cdot |S_1 \cup S_2 \cup S_3 \cup S''_4| + |S'_4 \cup S''_4|)$, respectively.

Since our current hash table $H_k$, maintains for each ID $id$, the number of occurrence of $k_j$ in the subtree rooted at $v$, the IDList for ELCA computation can be simplified by removing the third row in Fig. 3.
Table 3  Statistics of keywords used in our experiment

<table>
<thead>
<tr>
<th>Keyword</th>
<th>tissue</th>
<th>baboon</th>
<th>necklace</th>
<th>arizona</th>
<th>cabbage</th>
<th>books</th>
<th>shocks</th>
<th>patients</th>
<th>cognition</th>
<th>villages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>344</td>
<td>725</td>
<td>200</td>
<td>451</td>
<td>366</td>
<td>461</td>
<td>596</td>
<td>382</td>
<td>495</td>
<td>829</td>
</tr>
<tr>
<td>N_{LD}</td>
<td>3,252</td>
<td>6,013</td>
<td>1,704</td>
<td>3,255</td>
<td>3,016</td>
<td>3,869</td>
<td>4,823</td>
<td>3,306</td>
<td>4,192</td>
<td>6,981</td>
</tr>
<tr>
<td>L</td>
<td>2,281</td>
<td>4,014</td>
<td>1,196</td>
<td>1,355</td>
<td>2,071</td>
<td>2,674</td>
<td>3,302</td>
<td>2,609</td>
<td>2,857</td>
<td>4,867</td>
</tr>
<tr>
<td>Keyword</td>
<td>male</td>
<td>takano</td>
<td>order</td>
<td>school</td>
<td>check</td>
<td>education</td>
<td>female</td>
<td>province</td>
<td>privacy</td>
<td>gender</td>
</tr>
<tr>
<td></td>
<td>18,441</td>
<td>17,129</td>
<td>16,797</td>
<td>23,561</td>
<td>36,304</td>
<td>35,257</td>
<td>19,902</td>
<td>33,520</td>
<td>31,232</td>
<td>34,065</td>
</tr>
<tr>
<td>N_{LD}</td>
<td>113,081</td>
<td>100,338</td>
<td>116,425</td>
<td>145,894</td>
<td>213,449</td>
<td>187,843</td>
<td>113,081</td>
<td>173,900</td>
<td>139,455</td>
<td>177,450</td>
</tr>
<tr>
<td>L</td>
<td>62,162</td>
<td>53,324</td>
<td>68,049</td>
<td>91,138</td>
<td>106,974</td>
<td>115,318</td>
<td>70,747</td>
<td>105,795</td>
<td>66,244</td>
<td>108,098</td>
</tr>
</tbody>
</table>

Table 4  Queries used for CA computation

<table>
<thead>
<tr>
<th>ID</th>
<th>Keywords</th>
<th># of Results</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>bold, increase, text</td>
<td>54,135</td>
<td>G1</td>
</tr>
<tr>
<td>Q3</td>
<td>bold, increase, text, keyword</td>
<td>25,346</td>
<td></td>
</tr>
<tr>
<td>Q4</td>
<td>bold, increase, text, keyword, emph</td>
<td>20,766</td>
<td></td>
</tr>
<tr>
<td>Q5</td>
<td>date, listitem</td>
<td>50,706</td>
<td>G2</td>
</tr>
<tr>
<td>Q6</td>
<td>date, listitem, bold</td>
<td>16,355</td>
<td></td>
</tr>
<tr>
<td>Q7</td>
<td>date, listitem, bold, time</td>
<td>16,355</td>
<td></td>
</tr>
<tr>
<td>Q8</td>
<td>date, listitem, bold, time, emph</td>
<td>16,355</td>
<td></td>
</tr>
</tbody>
</table>

CA computation (Group 1). Algorithms of this group target at finding all CA from a set of IDLists, including SvS [9], Small Adaptive (SA) [9], Quantile-based (QB) [24], Probabilistic Intersection (PI) [21], FwdCA, BwdCA, and BwdCA^+. The last three are simplified from FwdSLCA, BwdSLCA, and BwdSLCA^+, where only function fwdGetCA, bwdGetCA and bwdGetCA^+ are called to return CA nodes. We do not compare these methods with the hash search-based methods, since they cannot be applied to the general set intersection problem where no hash table is available.

SLCA computation (Group 2). Algorithms of this group focus on SLCA computation, which consists of two sub-groups: (Group 2.1) algorithms that are not based on hash search, including Stack [26], IL [26], IMS [22], JDewey [4], and the three IDList-based algorithms, that is, FwdSLCA, BwdSLCA, and BwdSLCA^+; (Group 2.2) algorithms that are based on hash search, including HS [25], FwdSLCA-HS, BwdSLCA-HS, and HybSLCA-HS.

ELCA computation (Group 3). Algorithms of this group also consists of two sub-groups: (Group 3.1) algorithms that are not based on hash search, including DIL [12], IS [27], JDewey [4], FwdELCA, and BwdELCA; (Group 3.2) algorithms that are based on hash search, including HC [31], FwdELCA-HS, BwdELCA-HS, and HybELCA-HS.

All algorithms were implemented using Microsoft VC++.

To make a fair comparison, we only evaluate the performance of all algorithms on main memory resident data. We used the XMark (582MB)2 and DBLP (876MB)3 datasets in our experiment. After parsing the two datasets, Oracle Berkeley DB4 is used to store the keyword inverted lists by a hash file, where each key is a keyword k and the value associated with k is the inverted list of k, which maintains the set of Dewey/JDewey/IDDewey labels, or the set of entries of the corresponding IDList. When processing a given query Q, the set of inverted lists is firstly loaded into memory, and the running time is the averaged one over 100 runs with warm cache without taking the I/O cost into account.

We randomly selected 30 keywords for XMark dataset that falls into three categories according to their frequencies (i.e., |L^D| line in Table 3): (1) low frequency (100–1,000), (2) median frequency (10,000–40,000), (3) high frequency (300,000–600,000). We then generate queries by combining these keywords. Queries for DBLP are generated in a similar way. We mainly focus on results of XMark dataset, which is more complex and allows us to perform scalability test.

9.2 Processed nodes

Each number in line |L^D| of Table 3 denotes the number of Dewey labels corresponding to a keyword. The number of nodes they need to process is shown in the N_{LD} rows, which are the sum of the lengths of all Dewey labels. The number of nodes our methods need to process is shown in the L rows. We can see that the ratio of processed nodes for each keyword between our methods and existing methods is at most 0.7 (necklace). In addition, with the increase of keyword frequency, the ratio decreases dramatically, with the minimum ratio as 0.24 (listitem).

2 http://www.monetdb.org/Home.
3 http://www.informatik.uni-trier.de/~ley/db/.
9.3 CA computation

We generated two groups of queries (Table 4) to test the efficiency of different set intersection algorithms.

The metrics include: (1) running time; (2) number of probe operations; (3) number of comparison operations, which is affected by both the search interval and search method.

9.3.1 Impacts of different probe order

As shown in Fig. 14a, with the increase of the number of lists and the decrease of the number of results, SA is more efficient than SvS. QB is more efficient than SA in most cases, but when the lists cannot be well partitioned into sublists, such as Q3 and Q4, it degenerates to SvS. PI has slightly worse performance than QB on average. FwdCA beats all existing methods for all queries in running time.

This excellent performance can be explained according to Fig. 14b, C, which show the number of probe and comparison operations, respectively. From Fig. 14b, we know that the number of probe operations of our method is much less than that of existing methods. Also note that except Q1, Q3 and Q4, the number of probe operations of QB is much larger than that of PI. But as shown in Fig. 14a, QB performs at least as good as PI in most cases, the reason can be further explained by referring to Fig. 14c. We can see that QB has much less comparison operations than PI, because when the initial set of lists are partitioned into sublists, the search interval for each probe operation of QB is much shorter than that of PI. According to Fig. 14c, our method achieves the best performance, the reason lies in the least number of comparison operations.

9.3.2 Impacts of reducing search interval

Figure 15a presents the running time of FwdCA, BwdCA, and BwdCA+. We can find: (1) BwdCA does not always beat FwdCA, (2) BwdCA+ outperforms both FwdCA and BwdCA for all cases.

According to Fig. 15b, BwdCA reduces many probe operations for Q1 to Q4, which can be verified according to Fig. 15c, that is, BwdCA reduces the number of comparison operations when compared with FwdCA for Q1 to Q4. However, when the number of probe and comparison operations cannot
be reduced significantly, FwdCA may be more efficient than BwdCA, this is because BwdCA needs to check whether a CA node is the parent node of the one computed in the previous iteration. Therefore, when both FwdCA and BwdCA possess similar number of comparison operations, FwdCA may be more efficient than BwdCA. Although BwdCA+ processes the same number of probe operations as that of BwdCA, from Fig. 15c, we find that BwdCA+ needs much less comparison operations than BwdCA, because BwdCA+ can reduce the search interval for each probe operation.

Note that as shown in Table 5, the number of comparison operations presented in Fig. 15c for BwdCA+ consists of three parts: (1) \( N_1 \), the number of actual comparison operations in the bwdBinSearch function, (2) \( N_2 \), the number of comparison operations in the setInterval function to reduce the search interval, and (3) \( N_3 \), the number of comparisons in line 2 of the fwdGetCA/bwdGetCA function, which is equal to \( n \times (m - 1) \), where \( n \) is the number of CA nodes, \( m \) is the number of keywords of a given query. For FwdCA and BwdCA, the number of comparison is \( N_1 + N_3 \).

From the above discussion, we conclude the following results: (1) using the maximum cursor as the eliminator to probe the shortest list can improve the overall performance in most cases; (2) the optimization technique of BwdSLCA+ can improve the overall performance remarkably.

9.4 SLCA computation

Based on the keywords in Table 3, we generated four groups of queries as shown in Table 6: (1) four queries with 2, 3, 4, 5 keywords of low frequency; (2) four queries of median frequency; (3) four queries of high frequency; (4) six queries with randomly selected keywords.

9.4.1 Evaluation metrics

The metrics for evaluating algorithms for SLCA and ELCA computation include: (1) running time and (2) number of comparison operations, which helps us understand the performance variance of these algorithms in an in-depth way.

Specifically, all algorithms of Group 2.1 (see Sect. 9.1 for detailed information) can be classified into two subgroups: (Group 2.1.1) algorithms that are based on \( OP1 \) and \( OP2 \) operations, such as Stack, IL, and IMS; (Group 2.1.2) algorithms that are based on set intersection, such as JDewey, FwdSLCA, BwdSLCA, and BwdSLCA+.

For algorithms in Group 2.1.1, the total number of comparisons, that is, \( N_C \), can be computed as \( N_C = \sum_{i=1}^{M} n_i \), where \( n_i \) is the number of compared ID pairs to compute the LCA or positional relationships for two Dewey labels, \( M \) the number of LCA, and positional relationship operations.

For algorithms in Group 2.1.2, the total number of comparisons, that is, \( N_C \), for JDewey can be computed using the same formula, where \( n_i \) is the number of search steps of a binary search operation, and \( M \) is the number of binary search operations. For FwdSLCA and BwdSLCA, \( N_C = N_1 + N_3 \); while for BwdSLCA+, \( N_C = N_1 + N_2 + N_3 \), where \( N_1, N_2 \) and \( N_3 \) have the same meanings as that in Table 5.

For algorithms in Group 2.2, as shown in Table 7, their performance is affected by two factors: (1) the number of hash probe operations (2) the number of comparison operations \( N_C \). For HS, \( N_C \) is the number of compared ID pairs to compute the LCA for two Dewey labels, for FwdSLCA- HS, BwdSLCA-HS, and HybSLCA-HS, \( N_C \) is the number of nodes processed without hash probe operations.

9.4.2 Performance comparison and analysis

Figure 16 shows the running time of algorithms in Group 2 on queries Q9 to Q26, from which we have the following observations for algorithms of Group 2.1: (1) among existing algorithms, no one can beat others for all queries; (2) our methods perform much better than existing methods.

For existing methods, since Stack processes all nodes in document order, its performance is mainly affected by the number of processed nodes. IL and IMS algorithms can use the positional relationships between nodes to skip many useless ones, thus achieve better performance when there exists huge difference in the lengths of the set of inverted lists, for example, Q21 to Q26 in Fig. 16d; if the set of Dewey label lists have approximate lengths and the result selectivity becomes high, the chances of skipping useless elements are largely reduced, and the performance of IL and IMS may

---

**Table 5** The composition of the comparison operation for BwdCA+

<table>
<thead>
<tr>
<th>#</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N_1 )</td>
<td>554,767</td>
<td>1,456,241</td>
<td>1,165,748</td>
<td>1,450,364</td>
<td>1,026,313</td>
<td>712,245</td>
<td>719,006</td>
<td>915,088</td>
</tr>
<tr>
<td>( N_2 )</td>
<td>213,617</td>
<td>318,335</td>
<td>385,424</td>
<td>439,540</td>
<td>311,322</td>
<td>102,162</td>
<td>132,571</td>
<td>189,292</td>
</tr>
<tr>
<td>( N_3 )</td>
<td>58,316</td>
<td>68,371</td>
<td>70,038</td>
<td>83,064</td>
<td>50,706</td>
<td>33,710</td>
<td>49,065</td>
<td>65,430</td>
</tr>
</tbody>
</table>

**Table 6** Queries used for SLCA and ELCA computation on 582MB XML keyword queries

<table>
<thead>
<tr>
<th>ID</th>
<th>Keywords</th>
<th>( R_E )</th>
<th>( R_S )</th>
<th>Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q7</td>
<td>villages, books</td>
<td>9</td>
<td>9</td>
<td>Low</td>
</tr>
<tr>
<td>Q10</td>
<td>baboon, patient, arizona</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Q11</td>
<td>cabbage, tissue, shocks, baboon</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Q12</td>
<td>shocks, necklace, cognition, cabbage, tissue</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Q13</td>
<td>female, order</td>
<td>570</td>
<td>579</td>
<td></td>
</tr>
<tr>
<td>Q14</td>
<td>privacy, check, male</td>
<td>29</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>Q15</td>
<td>takano, province, school, gender</td>
<td>107</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>Q16</td>
<td>school, gender, education, takano, province</td>
<td>107</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>Q17</td>
<td>bold, increase</td>
<td>34,136</td>
<td>34,189</td>
<td></td>
</tr>
<tr>
<td>Q18</td>
<td>date, list, item, emph</td>
<td>43,777</td>
<td>43,792</td>
<td></td>
</tr>
<tr>
<td>Q19</td>
<td>in, category, text, header, date</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Q20</td>
<td>header, date, keyword, in, category, text</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Q21</td>
<td>in, category, cabbage</td>
<td>225</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>Q22</td>
<td>province, bold, increase</td>
<td>427</td>
<td>436</td>
<td></td>
</tr>
<tr>
<td>Q23</td>
<td>list, item, emph, arizona</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Q24</td>
<td>bold, increase, books, takano</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Q25</td>
<td>emph, arizona, villages, education</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Q26</td>
<td>check, header, date, baboon</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
not be as efficient as that of Stack, for example, Q17 and Q18. Although JDewey is based on set intersection method, it needs to process all lists of each level from the leaf to the root; and for all lists of each level, after finding the set of common nodes, it needs to recursively delete all ancestor nodes in all lists of higher levels, which is very expensive in practice.

The above results can be further verified according to the number of comparison operations they have done, which was shown in Fig. 17, from which we have several observations: (1) our methods always need the least comparison operations, therefore achieve the best performance for all queries; (2) when the cost of the saved probe operations is less than the additional cost for skipping useless probe operations, FwdSLCA is more efficient than BwdSLCA; (3) BwdSLCA\textsuperscript{+} always performs better than FwdSLCA and BwdSLCA, the reason lies in two aspects: (3.1) in most cases as shown in Fig. 17, the number of comparison operations of BwdSLCA\textsuperscript{+} is less than that of FwdSLCA and BwdSLCA, (3.2) even in some cases, BwdSLCA\textsuperscript{+} needs more comparison operations, as shown in Table 5, 15–30\% comparison operations exist in the setInterval function, which can be done much more efficiently than the comparison operations in each probe operation.

For example, even though BwdSLCA\textsuperscript{+} needs more comparison operations for some queries, it actually saves 5–45\% cost compared with FwdSLCA and BwdSLCA for Q9–Q26.

From Fig. 16, we have the following observations for algorithms of Group 2.2, that is, HS, FwdSLCA-HS, BwdSLCA-HS, and HybSLCA-HS: (1) FwdSLCA-HS and HybSLCA-HS always beat HS for all queries; (2) BwdSLCA-HS outper-
Table 7 The composition of the comparison operation for HS, FwdSLCA-HS, and BwdSLCA-HS, $N_H$ is the number of hash probe operations, $N_C$ is the number of comparison operations

<table>
<thead>
<tr>
<th>Query</th>
<th>$N_H$</th>
<th>FwdSLCA-HS</th>
<th>BwdSLCA-HS</th>
<th>HybSLCA-HS</th>
<th>FwdSLCA-HS</th>
<th>BwdSLCA-HS</th>
<th>HybSLCA-HS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q9</td>
<td>1899</td>
<td>471</td>
<td>2,457</td>
<td>509</td>
<td>483</td>
<td>1,968</td>
<td>2</td>
</tr>
<tr>
<td>Q10</td>
<td>764</td>
<td>1</td>
<td>2,651</td>
<td>12,646</td>
<td>4,369</td>
<td>15,528</td>
<td>58,244</td>
</tr>
<tr>
<td>Q11</td>
<td>3,361</td>
<td>396</td>
<td>1,825</td>
<td>407</td>
<td>388</td>
<td>1,433</td>
<td>2</td>
</tr>
<tr>
<td>Q12</td>
<td>2,383</td>
<td>240</td>
<td>1,116</td>
<td>230</td>
<td>222</td>
<td>881</td>
<td>2</td>
</tr>
<tr>
<td>Q13</td>
<td>61,044</td>
<td>17,755</td>
<td>59,330</td>
<td>17,775</td>
<td>18,875</td>
<td>44,412</td>
<td>1,323</td>
</tr>
<tr>
<td>Q14</td>
<td>31,972</td>
<td>2,851</td>
<td>12,646</td>
<td>2,369</td>
<td>18,528</td>
<td>58,244</td>
<td>10</td>
</tr>
<tr>
<td>Q15</td>
<td>153,856</td>
<td>18,317</td>
<td>53,878</td>
<td>16,309</td>
<td>17,449</td>
<td>37,033</td>
<td>3</td>
</tr>
<tr>
<td>Q16</td>
<td>198,913</td>
<td>18,435</td>
<td>54,331</td>
<td>16,309</td>
<td>17,449</td>
<td>37,033</td>
<td>3</td>
</tr>
<tr>
<td>Q17</td>
<td>1,057,672</td>
<td>241,253</td>
<td>438,336</td>
<td>241,088</td>
<td>413,176</td>
<td>438,435</td>
<td>17,877</td>
</tr>
<tr>
<td>Q18</td>
<td>2,216,869</td>
<td>185,599</td>
<td>379,864</td>
<td>124,926</td>
<td>472,010</td>
<td>449,936</td>
<td>9,355</td>
</tr>
<tr>
<td>Q19</td>
<td>399,018</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>299,018</td>
<td>353,219</td>
<td>0</td>
</tr>
<tr>
<td>Q20</td>
<td>897,054</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>299,018</td>
<td>353,219</td>
<td>0</td>
</tr>
<tr>
<td>Q21</td>
<td>1,060</td>
<td>458</td>
<td>1,001</td>
<td>463</td>
<td>1,244</td>
<td>1,351</td>
<td>8</td>
</tr>
<tr>
<td>Q22</td>
<td>46,858</td>
<td>4,920</td>
<td>16,488</td>
<td>2,589</td>
<td>34,886</td>
<td>102,079</td>
<td>392</td>
</tr>
<tr>
<td>Q23</td>
<td>451</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>451</td>
<td>1,334</td>
<td>0</td>
</tr>
<tr>
<td>Q24</td>
<td>3,601</td>
<td>568</td>
<td>2,649</td>
<td>304</td>
<td>477</td>
<td>2,145</td>
<td>2</td>
</tr>
<tr>
<td>Q25</td>
<td>451</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>451</td>
<td>1,354</td>
<td>0</td>
</tr>
<tr>
<td>Q26</td>
<td>2,639</td>
<td>179</td>
<td>1,062</td>
<td>178</td>
<td>745</td>
<td>3,518</td>
<td>0</td>
</tr>
</tbody>
</table>

forms HS for most queries, especially for query Q19 and Q20, it outperforms HS by more than two orders of magnitude; (3) no one of our methods can beat others for all queries, and HybSLCA-HS always exhibits approximate optimal performance for all queries. The above observations can be further explained by results in Table 7.

For the first observation, from Table 7 we know that even though FwdSLCA-HS suffers from more comparison operations for most queries, FwdSLCA-HS still beats HS for all queries, especially for Q19 and Q20, it outperforms HS by more than an order of magnitude. This is because FwdSLCA-HS greatly reduces the number of hash probe operations as compared with HS, and the cost of each hash probe operation is much more expensive than that of comparison operation between two integers. HybSLCA-HS usually suffers from similar (or, much less) hash probe and comparison operations as compared with FwdSLCA-HS, therefore beats HS for all queries.

For the second observation, from Table 7 we know that compared with HS, BwdSLCA-HS significantly reduces the number of hash probe operations, and at the same time, suffers from much less comparison operations than that of HS, thus can beat HS for most queries.

For the third observation, from Table 7 we know that FwdSLCA-HS is more eager in reducing hash probe operations, while usually suffers from more comparison operations; BwdSLCA-HS is on the other side, it usually suffers from more hash probe operations, while possesses the benefits of having much less comparison operations. In general, only when both algorithms suffer from similar hash probe operations, and FwdSLCA-HS suffers from much more comparison operations, BwdSLCA-HS can beat FwdSLCA-HS. For example, for Q19 and Q20, BwdSLCA-HS outperforms FwdSLCA-HS by more than two orders of magnitude. In other cases, FwdSLCA-HS is more efficient than BwdSLCA-HS. As a comparison, HybSLCA-HS possesses the benefits of both FwdSLCA-HS and FwdSLCA-HS. It not only greatly reduces the number of hash probe operations, such as Q15 to Q18, Q22 and Q24, but also the number of comparison operations, such as Q10, Q19, Q20, Q23 and Q25.

By comparing our methods of Group 2, we know that FwdSLCA-HS, BwdSLCA-HS and HybSLCA-HS can work better than FwdSLCA, BwdSLCA and BwdSLCA+ in many cases because of their lower time complexity. Even though, FwdSLCA, BwdSLCA and BwdSLCA+ still beat FwdSLCA-HS, BwdSLCA-HS and HybSLCA-HS for some queries, such as Q13, Q15 and Q16. This is because FwdSLCA, BwdSLCA, and BwdSLCA+ can utilize nodes in other lists to skip more useless nodes, while FwdSLCA-HS, BwdSLCA-HS, and HybSLCA-HS can only utilize the positional relationship between nodes of $L_1$ to skip useless nodes.

Besides the 18 queries in Table 6, we generated 174,406 queries by combining all keywords of Table 3.

Figure 18 shows the impacts of result selectivity on the performance of these algorithms grouped into different selectivity ranges. The selectivity of a query $Q$ is defined as $|R|/|L_1|$, where $R$ is the size of the query result, and $L_1$ is the size of the smallest IDList. For algorithms of Group 2.1, we can see that (1) the speedup of the IL, IMS, JDewey and our BwdSLCA+ algorithms over Stack is more significant when the result selectivity is low, (2) our method outperforms all existing algorithms in all cases. For algorithms of Group 2.2, FwdSLCA-HS always outperforms HS for all cases. Moreover, FwdSLCA-HS performs better than BwdSLCA+ on average.

Note that in Fig. 18a, b, with the increase of the result selectivity, the average time used by most methods for result selection increases. In Fig. 18a, we take the result selectivity of existing methods as that of our method to make a fair comparison.
selectivity in [40,100] is less than that in [30,40), which can be explained by Fig. 19, where for queries with 2 and 3 keywords, the number of results decreases with the change of result selectivity from [30,40) to [40,100].

Table 8 shows the 10 queries on DBLP dataset, and the experimental results are shown in Table 9. From Table 9 we know that for queries of Group 2.1, our methods are much more efficient than existing methods; for queries of Group 2.2, our methods can beat HS for all queries, and HybSLCA-HS always performs best. Note that compared with IL and IMS, the speedup of FwdSLCA, BwdSLCA and BwdSLCA+ is not as significant as that of Fig. 16. This is because the DBLP document is very shallow, and the cost of testing the document order and computing the LCA on DBLP dataset is not as expensive as that on XMark dataset. For HS, FwdSLCA-HS, BwdSLCA-HS, and HybSLCA-HS, we have similar observation.

From the above discussion, we conclude that our methods outperform all existing methods. This is mainly because we avoid the CAR problem by adopting IDList, such that the number of processed nodes (see Sect. 9.2) and the number of comparison operations are largely reduced, especially for queries with low result selectivity.

9.4.3 Scalability

We investigate the scalability from two aspects based on Fig. 18: (1) fixing the number of keywords and varying the result selectivity, which is just explained in the previous paragraphs, (2) fixing the result selectivity and varying the number of keywords, which can be obtained from the four sub-figures of Fig. 18 and is omitted due to lack of space. The general trend is: the performance of Stack will become worse with the increase of the number of keywords, but the performance of IL, IMS, JDewey, HS, BwdSLCA+, and FwdSLCA-HS will be better with the increase of the number of keywords; meanwhile, compared with existing methods, the performance gain of our method is more significant with the increase of the number of keywords.

Figure 20 shows the scalability when executing Q17 on XML documents of different sizes, from which we find our methods achieve better scalability. For other queries, we have similar results, which are omitted due to space limit.
Table 9  Comparison of the running time for different algorithms on DBLP dataset based on SLCA semantics (ms)

<table>
<thead>
<tr>
<th>Query</th>
<th>Stack</th>
<th>IL</th>
<th>IMS</th>
<th>JDewey</th>
<th>FwdSLCA</th>
<th>BwdSLCA</th>
<th>BwdSLCA+</th>
<th>HS</th>
<th>FwdSLCA-HS</th>
<th>BwdSLCA-HS</th>
<th>HybSLCA-HS</th>
</tr>
</thead>
<tbody>
<tr>
<td>QD1</td>
<td>149.6</td>
<td>8.91</td>
<td>2.66</td>
<td>35.9</td>
<td>0.94</td>
<td>0.93</td>
<td>0.79</td>
<td>0.41</td>
<td>0.29</td>
<td>0.39</td>
<td>0.28</td>
</tr>
<tr>
<td>QD2</td>
<td>168.4</td>
<td>42.97</td>
<td>42.34</td>
<td>45.71</td>
<td>12.66</td>
<td>15.16</td>
<td>13.28</td>
<td>14.99</td>
<td>5.48</td>
<td>6.46</td>
<td>5</td>
</tr>
<tr>
<td>QD3</td>
<td>174.8</td>
<td>61.56</td>
<td>67.03</td>
<td>48.27</td>
<td>15.63</td>
<td>18.75</td>
<td>15.15</td>
<td>18.03</td>
<td>7.81</td>
<td>9.28</td>
<td>6.41</td>
</tr>
<tr>
<td>QD4</td>
<td>156</td>
<td>16.09</td>
<td>8.44</td>
<td>38.78</td>
<td>3.9</td>
<td>4.38</td>
<td>3.91</td>
<td>2.22</td>
<td>1.68</td>
<td>2.32</td>
<td>1.33</td>
</tr>
<tr>
<td>QD5</td>
<td>149.8</td>
<td>5.47</td>
<td>2.34</td>
<td>34.03</td>
<td>0.63</td>
<td>0.78</td>
<td>0.62</td>
<td>0.31</td>
<td>0.26</td>
<td>0.44</td>
<td>0.19</td>
</tr>
<tr>
<td>QD6</td>
<td>477.4</td>
<td>99.22</td>
<td>107.5</td>
<td>207.99</td>
<td>52.34</td>
<td>40.94</td>
<td>43.59</td>
<td>23.81</td>
<td>23.78</td>
<td>21.8</td>
<td>19.3</td>
</tr>
<tr>
<td>QD7</td>
<td>184.2</td>
<td>4.84</td>
<td>1.25</td>
<td>35.61</td>
<td>0.31</td>
<td>0.16</td>
<td>0.31</td>
<td>0.38</td>
<td>0.34</td>
<td>0.42</td>
<td>0.33</td>
</tr>
<tr>
<td>QD8</td>
<td>205.8</td>
<td>24.07</td>
<td>11.41</td>
<td>41.57</td>
<td>2.5</td>
<td>2.81</td>
<td>3.13</td>
<td>6.38</td>
<td>5.25</td>
<td>7.71</td>
<td>4.61</td>
</tr>
<tr>
<td>QD9</td>
<td>199.6</td>
<td>14.68</td>
<td>6.56</td>
<td>43.04</td>
<td>2.03</td>
<td>2.5</td>
<td>2.35</td>
<td>1.73</td>
<td>1.34</td>
<td>2.11</td>
<td>1.2</td>
</tr>
<tr>
<td>QD10</td>
<td>194.3</td>
<td>4.38</td>
<td>2.81</td>
<td>42.34</td>
<td>0.63</td>
<td>0.78</td>
<td>0.78</td>
<td>0.33</td>
<td>0.28</td>
<td>0.53</td>
<td>0.27</td>
</tr>
</tbody>
</table>

Fig. 20  Running time of Q17 on different XML documents

9.5 ELCA computation

Figure 21 shows the running time of different algorithms on queries Q9–Q26 for ELCA computation.

As shown in Fig. 21, for queries of Group 3.1, the performance of DIL is mainly affected by the number of involved Dewey labels. Even though IS can utilize the positional relationships to skip many useless Dewey labels, for queries with similar lengths, it may not be as efficient as DIL, for example, Q15, Q17, Q19, and Q20. As discussed before, JDewey needs to process all lists of each level from the leaf to the root; and for all lists of each level, after finding the set of common nodes, it needs to recursively delete all ancestor nodes in all lists of higher levels, which is very expensive in practice. In contrast, our methods always need the least comparison operations, thus achieve the best performance for all queries as compared with algorithms of Group 3.1. From Fig. 21, we also know that BwdELCA is more efficient than FwdELCA, because BwdELCA uses the optimization techniques introduced in Sect. 5 to reduce the search interval for each probe operation.

For queries of Group 3.2, HC can beat other existing methods of Group 3.1 in most cases except when the shortest list is very long, such as Q17, Q18, and Q20. As a comparison, our methods achieve best performance gain by avoiding the CAR problem.

Note that compared with our methods of Group 3.1, our methods in Group 3.2 may not be as efficient as that of Group 3.1, such as Q13, Q15, Q16, Q22. This is because our methods in Group 3.1 can utilize nodes in other lists to skip more useless nodes, while FwdELCA-HS, BwdELCA-HS and HybSLCA-HS can only utilize the positional relationship between nodes of $L_1$ to skip useless nodes.
10 Conclusions

A bottleneck for existing Dewey-based algorithms for SLCA and ELCA computation is the repeated access and comparison of common-ancestor nodes. In this paper, we proposed to use IDLists, rather than inverted lists of Dewey labels, for SLCA and ELCA computation. We showed that both SLCA and ELCA computations can be cast into a set intersection problem among IDLists of query keywords. Several optimization methods based on the query semantics and structural relationship between nodes have been devised to speed up the query processing. Further, we proposed several hash search-based methods to reduce the time complexity. Experimental results demonstrate superior performance of our proposed methods over existing ones.

Besides the above benefits, the following problems are still needed to, and will be addressed in our future work.

(1) IDLists are tailored to XML keyword queries, a possible downside is that we still need to maintain inverted Dewey label lists to support structured queries involving keyword constraints. Hence, one of our future work will focus on designing new algorithms based on IDlists to answer structured queries, such that it will become a general-purpose indexing scheme for building a flexible and efficient XML search engine for both structured and keyword queries.

(2) It is difficult to find the parent node by PIDPos if current IDList is organized as a $B^+$ tree index, where each key is an ID value, and the data associated with this key is the corresponding entry in IDList. We will continue to study disk-based index to solve this problem in the future.

Acknowledgments This research was partially supported by the grants from the National Science Foundation of China (No. 61073060, 60833005, 61070055, 91024032, 91124001), the National Science and Technology Major Project (No. 2010-ZX01042-002-003), the Fundamental Research Funds for the Central Univ., the Research Funds from Education Department of Hebei Province (No. Y2012014). Zhifeng Bao’s research is carried out at the SeSaMe Centre. It is supported by the Singapore NRF under its IRC@SG Funding Initiative and administered by the IDMPO. Wei Wang was partially supported by ARC DP130103401 and DP130103405.

References