Noname manuscript No.
(will be inserted by the editor)

Efficient Query Processing for XML Keyword Queries based on the IDList Index

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

Received: date / Accepted: date

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* (CAR) 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 SLCA and ELCA 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.

Junfeng Zhou, Jinjia Zhao
School of Information Science and Engineering, Yanshan University
The Key Laboratory for Computer Virtual Technology and System Integration of HeBei Province
E-mail: zhoujf@ysu.edu.cn

Zhifeng Bao
National University of Singapore

Wei Wang
The University of New South Wales

Xiaofeng Meng
Renmin University of China

1 Introduction

Keyword search on XML data has received much attention in the literature recently [1, 4, 7, 12, 15, 17–19, 23, 26–28, 32]. 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 [16].

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, 15, 17, 27] 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, 18, 19, 23, 26, 27] and Exclusive Lowest Common Ancestor (**ELCA**) [4, 12, 13, 28, 32]. 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 iff (1) v is an LCA node, and (2) the subtree T_v rooted at v contains at least one occurrence of each query keyword, after excluding the occurrences of the keywords in each subtree that is rooted at a descendant LCA node of v.

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 [24] 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 [1, 12, 18, 19, 23, 27, 28, 32] 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 are 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 [1,12,18,19,23,27,28,32] that are based on $OP1$ 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, 22, 25]. 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 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 [26,32] 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 [31], we have several major updates: (1) We propose several algorithms that are based on hash search to accelerate SLCA/ELCA computation in Section 7.

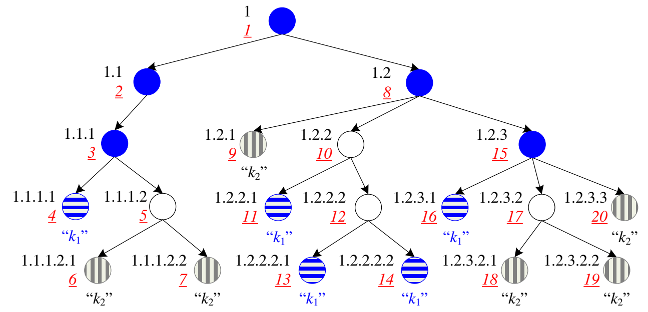

Fig. 1: A sample XML document, where solid circles denote CA nodes of $Q = \{k_1, k_2\}$, circles filled with horizontal lines denote nodes that directly contain k_1, and circles filled with vertical lines denote nodes that directly contain k_2.

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

The rest of the paper is organized as follows. In Section 2, we introduce preliminaries and related work. In Section 3, we introduce the CA tree and its properties. The inverted list and its properties are introduced in Section 4. The algorithms for SLCA/ELCA computation are discussed in Sections 5 and 6, respectively. In Section 7, we introduce algorithms for SLCA/ELCA computation based on hash search. In Section 8, we present an theoretical analysis and comparison of our and existing algorithms, and discuss extensions to other LCA based semantics. In Section 9, we present experimental results, then conclude our work in Section 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. Fig. 1 is a sample XML document.

The positional relationships between two nodes include Document Order ($\prec_d$), Equivalence ($=$), AD (ancestor - descendant, $\prec_a$), PC (parent-child, $\prec_p$), Ancestor-or-self ($\preceq_a$) and Sibling relationship. Given two nodes u and v, $u \prec_d v$ means that u is located before v in document order, $u \prec_a v$ means that u is an ancestor node of v, $u \prec_p v$ denotes that u is the parent node of v. If u and v represent the same node, we have $u = v$, and both $u \preceq_d v$ and $u \preceq_a v$ hold.

To accelerate the query processing, each node v is usually assigned with a label that uniquely represents v and can be used to compute some positional relationships to accelerate query processing. Existing methods [1, 12, 18, 19, 23, 27, 28, 32] usually assign each node v a Dewey label [24]. In Fig. 1, the Dewey label of each node is shown as the black sequence of numbers.

In our method, we assign each node an ID (the underlined number beside it in Fig. 1) that is compatible with the document order (we use the pre-order traversal number).

2.2 Query Semantics

Given a query $Q = \{k_1, k_2, ..., k_m\}$ and an XML document D, we use L_i^D 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 [26, 32]. All labels of L_i^D are sorted by document order. Let $LCA(v_1, v_2, ..., v_m)$ be the lowest common ancestor (LCA) of nodes $v_1, v_2, ..., v_m$, the LCAs of Q on D are defined as $LCA(Q) = \{v | v = LCA(v_1, v_2, ..., v_m), v_i \in L_i^D (1 \leq i \leq m)\}$. E.g., 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 [23, 27] and ELCA [12, 28, 32]. 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 $Q = \{k_1, k_2, ..., k_m\}$ and an XML document D, the set of SLCAs of Q on D are $SLCA(Q) = \{v | v \in LCA(Q)$ and $\nexists v' \in LCA(Q)$, such that $v \prec_a v'\}$.

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 $Q = \{k_1, k_2, ..., k_m\}$ and an XML document D, the set of ELCAs of Q on D are $ELCA(Q) = \{v | \exists v_1 \in L_1^D, ..., v_m \in L_m^D (v = LCA(v_1, ..., v_m) \wedge \forall i \in [1, m], \nexists x (x \in LCA(Q) \wedge child(v, v_i) \preceq_a x))\}$, 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 node 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 and 15. As node 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, thus node 8 is a qualified ELCA results. 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, i.e., 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 $|L_1| \leq |L_2| \leq \ldots \leq |L_m|$), 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*, i.e., which list should be probed first,

(2) the cost of each *probe operation*, i.e., how to efficiently find the *matched element*[1] for eliminator e, which is further affected by two orthogonal factors:

  (2.1) which *search method* is used (e.g., binary, galloping [3], or interpolation search [2]),

  (2.2) how large the *search interval* is.

2.3.1 Probe 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 [25] 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 [22] 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.

[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.

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, i.e., $m = \frac{a+b}{2}$. 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 $L[a'] \leq e \leq L[b']$, 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 $2^i (i = 1,2,3,...)$ 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: $a_i - b_{i-1} = 1$ 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.

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

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, i.e., 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, 15, 17, 23, 27, 28, 32] 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, 20], among which the two widely adopted semantics are SLCA [23,27] and ELCA [12,28,32].

To facilitate SLCA/ELCA computation, existing methods usually assign each node v a Dewey label [1, 12, 18, 19, 23, 27, 28], or one of its variants, such as JDewey [4] and IDDewey [26,32] (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 [27], IL [27] and IMS [23],
(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 [26].

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 Section 1, the two basic operations for these methods are $(OP1)$ testing the document order of two nodes and $(OP2)$ computing the LCA of two nodes, therefore, they heavily suffer from the CAR problem.

The second kind of algorithm, i.e., JDewey [4], performs set intersection operation on all lists of each tree depth from

Table 1: Worst case time complexities of different algorithms on SLCA and ELCA computation, where $|L_1^D|(|L_m^D|)$ is the length of the shortest (longest) inverted list consisting of Dewey/JDewey/IDDewey labels, ILD/ILJD/ILIDD means the inverted lists of Dewey/JDewey/IDDewey labels, m is the number of keywords of the given query, and d is the depth of the given XML tree.

Algorithm	Time Complexity	Labeling Scheme	Index
Stack [27]	$O(d \cdot m \cdot (\sum_1^m \|L_i^D\|))$	Dewey	ILD
DIL [12]	$O(d \cdot m \cdot (\sum_1^m \|L_i^D\|))$		
IL [27]	$O(d \cdot m \cdot \|L_1^D\| \cdot \log \|L_m^D\|)$		
IMS [23]	$O(d \cdot m \cdot \|L_1^D\| \cdot \log \|L_m^D\|)$		
IS [28]	$O(d \cdot m \cdot \|L_1^D\| \cdot \log \|L_m^D\|)$		
JDewey [4]	$O(d \cdot m \cdot \|L_1^D\| \cdot \log \|L_m^D\|)$	JDewey	ILJD
HS [26]	$O((\log d \cdot m + d) \cdot \|L_1^D\|)$	IDDewey	ILIDD + Hash Table
HC [32]	$O(d \cdot m \cdot \|L_1^D\|)$		

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 $OP1$ and $OP2$ 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, i.e., HS [26], takes the shortest list L_1^D as the working list and sequentially processes all IDDewey labels of L_1^D. In each iteration, it picks from L_1^D 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.

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 [28] takes the shortest list L_1^D as the working list. In each iteration, it picks from the shortest list L_1^D 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 $OP1$ and $OP2$ operations. The second kind of algorithm, i.e., JDewey, computes all ELCA nodes in the same way as that of SLCA computation. The third kind of algorithm, i.e., HC [32], works in the similar way as HS does. Therefore, for the same reason as discussed in Section 2.4.1, they all suffer from the CAR problem.

Table 1 shows the comparison of these algorithms, from which we know that IL [27], IMS [23], IS [28] and JDewey [4]

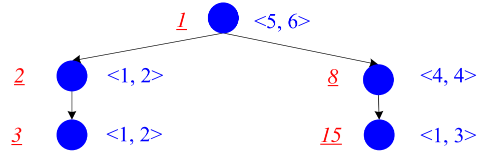

Fig. 2: A sample CA tree of $Q = \{k_1, k_2\}$ on the XML document of Fig. 1. Each italic number beside a node v denotes v's ID, and $< N_1, N_2 >$ is the vector associated with v denoting the number of nodes that directly contain "k_1" and "k_2" in the subtree rooted at v.

are better than Stack [27] and DIL [12] according to their time complexity; and HS [26] and HC [32] 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 Cornerstones of Our Method

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 node 1, 2, 3, 8 and 15. Obviously, we have the following lemma.

Lemma 1 *For a given query Q and an XML document D,* $SLCA(Q) \subseteq ELCA(Q) \subseteq LCA(Q) \subseteq CA(Q)$.

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, i.e., $CA(Q) = V_T$, V_E is the set of parent-child edges between two nodes of V_T.

Fig. 2 shows the CA tree of $Q = \{k_1, k_2\}$ on the XML document of Fig. 1, based on which we can identify all SLCA and ELCA nodes by the following two lemmas, respectively.

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* $SLCA(Q) = V_T^{leaf}$.

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 [32].

Lemma 3 *For a given keyword query* $Q = \{k_1, k_2, ..., k_m\}$ *and its CA tree T on an XML document D, assume that for each node* $v \in V_T$, $S_{child}^v = \{v_1, v_2, ..., v_l\}$ *is the set of child nodes of v in T, v is associated with a vector* $N = \langle N_1, N_2, \ldots, N_m \rangle$, *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 iff $\nexists i \in [1,m]$, such that $v.N_i = \sum_{j=1}^{l} v_j.N_i$.

Proof As each $v_i \in S^v_{child}$ is a CA node, if v_i contains only one child CA node v_{i_c}, 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 and v_{i_c} contain the same set of keyword nodes w.r.t. keywords of Q. If v_{i_c} 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_v 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 child CA nodes. Thus, if $\nexists i \in [1,m]$, such that $v.N_i = \sum_{j=1}^{l} v_j.N_i$, it means that after removing all subtrees rooted at v's child CA nodes (or descendant LCA nodes), T_v still contains each keyword of Q at least once, that is, v is an ELCA node. ■

For instance, the vector of each CA node for $Q = \{k_1, k_2\}$ is shown on its right in Fig. 2. According to Lemma 3, the ELCA nodes are node 3, 8 and 15. Node 2 is not an ELCA node since node 2 and 3 contain the same set of keyword nodes of k_1. Similarly, node 1 is not an ELCA node, since after excluding the two subtrees rooted at node 2 and 8, the subtree rooted at node 1 does not contain any keyword of Q.

Thanks to Lemmas 1, 2 and 3, we can now implement SLCA or ELCA computation by pruning the CA tree appropriately. This leads to the basic ideas of our methods: (1) *for SLCA, find all leaf nodes of the corresponding CA tree*, (2) *for ELCA, find all CA nodes and check their satisfiability according to Lemma 3*.

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,

L_1

Pos	0	1	2	3	4	5	6	7	8	9	10	11
ID	1	2	3	4	8	10	11	12	13	14	15	16
PIDPos	-1	0	1	2	0	4	5	5	7	7	4	10
N_{Desc}	5	1	1	1	4	3	1	2	1	1	1	1

L_2

Pos	0	1	2	3	4	5	6	7	8	9	10	11	12
ID	1	2	3	5	6	7	8	9	15	17	18	19	20
PIDPos	-1	0	1	2	3	3	0	6	6	8	9	9	8
N_{Desc}	6	2	2	2	1	1	4	1	3	2	1	1	1

Fig. 3: Data organization for SLCA and ELCA computation.

- and *optionally* "N_{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.

Fig. 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_{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 "k_1".

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

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

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 $L_i[x]$ to denote an entry of L_i at Pos x. We have the assertion that if $pos(C_i) = x$ then $L_i[x] = C_i$.

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 [30], 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 [21] can be applied to update our IDList index accordingly.

5 SLCA Computation

5.1 Forward Solution

Our first algorithm on SLCA computation, i.e., FwdSLCA, computes all CA nodes in document order according to Property 1, filters them 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 $u \not\prec_p v$, 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 and 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_k, though other kinds of search can also be used. Function fwdEof checks whether we have exhausted an IDList by checking the cursor positions.

Algorithm 1: FwdSLCA(Q)

```
/*Q = {k_1,...,k_m}, 0 < |L_1| ≤ |L_2| ≤ ... ≤ |L_m|*/
1  u ← {−1,−1};
2  while (¬ fwdEof()) do
3    {flag, parent, v} ← fwdGetCA()
4    if (flag = TRUE) then
5      if (parent.ID ≠ u.ID) then output u.ID as an answer
6      u ← v
7      foreach (i ∈ [1,m]) do fwdAdvance(C_i)
8    else break
9  endwhile
10 output u.ID as an answer

Function fwdGetCA()
1  j ← 1; n ← 1
2  k ← argmax_i{C_i.ID}
3  while (n < m) do
4    if (j = k) then
5      j ← j + 1
6    endif
7    fwdBinSearch(L_j, C_k)
8    if (pos(C_j) ≥ |L_j|) then return {FALSE, NULL, NULL}
9    if (C_k.ID = C_j.ID) then
10     j ← j + 1; n ← n + 1
11   else
12     k ← j; j ← 1; n ← 1
13 endwhile
14 return {TRUE, L_1[C_1.PIDPos], C_1}

Function fwdEof()
1  if (∃i, such that pos(C_i) = |L_i|) then return TRUE
2  else return FALSE

Procedure fwdBinSearch(L_j, u)
1  s ← pos(C_j); e ← |L_j|
2  while (s < e) do
3    mid ← (s+e)/2
4    if (L_j[mid].ID = u.ID) then {C_j ← L_j[mid]; break}
5    else if (L_j[mid].ID < u.ID) then s ← mid + 1
6    else e ← mid − 1
7  endwhile
8  if (s > e) then C_j ← L_j[s]
```

Example 2 For query $Q = \{k_1, k_2\}$ and its two IDLists in Fig. 3, FwdSLCA will find all CA nodes of Q, i.e., 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, i.e., SLCA node, by testing their parent-child relationship. Consider node 3 and 8, node 8's Pos values is 4. Since $3 \neq L_1[L_1[4].PIDPos].ID = L_1[0].ID = 1$, 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, i.e., $|L_1|$, therefore, the worst-case time complexity of FwdSLCA is $O(m \cdot |L_1| \cdot \log(|L_m|))$.

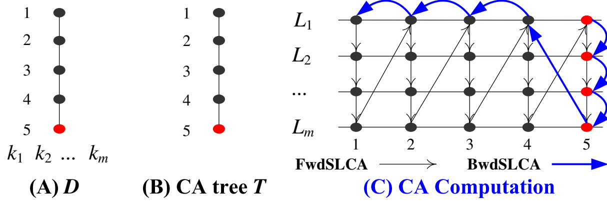

Fig. 4: The problem of Algorithm 1 (better viewed in color).

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, 29]).

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. 4 (A). The CA tree of Q on D is T in Fig. 4 (B). 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. 4 (C), 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 $v \neq u'$.*

Proof Assume the contrary that $v \neq u'$ but v is *not* an SLCA node. There are only two possible cases with regard to the relationship of v and u' in terms of document order.

Case I : $v \prec_d u'$. 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 SLCA 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 bwdGetCA is called in each iteration to find an ***SLCA*** node from the m IDLists, which in turn calls bwdBinSearch to locate the matched node for C_k. Note that in line 4.2 and 4.3 of 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 2: BwdSLCA(Q)

```
1  parent ← {−1,−1}
2  while (¬ bwdEof()) do
3    {flag, parent, v} ← bwdGetCA(parent)
4    if (flag = TRUE) then output v.ID as an answer
5    else break
6    foreach (i ∈ [1,m]) do bwdAdvance(C_i)
7    while (parent.ID = C_1.ID) do
8      parent ← L_1[C_1.PIDPos]; bwdAdvance(C_1)
9    endwhile
10 endwhile
```

Function bwdGetCA($parent$)

```
/*Only differences from fwdGetCA are shown here*/
2  k ← argmin_i{C_i.ID}
7  parent ←bwdBinSearch(L_j, C_k, parent)
8  if (pos(C_j) = −1) then return {FALSE, NULL, NULL}
14 return {TRUE, parent, C_1}
```

Function bwdEof()

```
1  if (∃i, such that pos(C_i) = −1) then return TRUE
2  else return FALSE
```

Function bwdBinSearch($L_j, u, parent$)

```
/*Only differences from fwdBinSearch are shown here*/
1  s ← 0; e ← pos(C_j)
4.1  if (L_j[m].ID = u.ID) then
4.2    if (L_j[m].ID = parent.ID) then
4.3      parent ← L_j[L_j[m].PIDPos]; s ← m; e ← m − 1
4.4    else C_j ← L_j[m]
8  if (s > e) then C_j ← L_j[e]
9  return parent
```

algorithm stops without computing any other CA node. As shown by the blue solid arrows in Fig. 4 (B). □

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

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.

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 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 $u \preceq_d C_k \prec_d v$. Now the search interval for each probe operation can be reduced from $[0, pos(C_j)]$ to $[pos(u), pos(v)]$ in line 1 of function 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,*

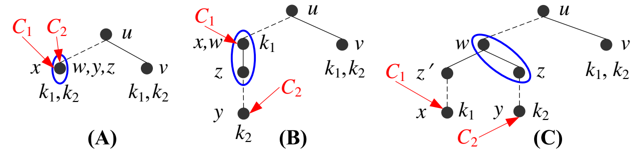

Fig. 5: The positional relationships between nodes in an XML document, where each solid (dashed) line from u to v means $u \prec_p v (u \preceq_a v)$.

- *y be the first node that precedes v in L_i,*
- *C_k be the eliminator used in the next iteration (hence satisfying $u \preceq_d C_k \preceq_d y \prec_d v$),*
- *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 $u \preceq_d w \preceq_d C_k(\prec_d z) \preceq_d y \prec_d v$, 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 $x \preceq_d y$, according to BwdSLCA, C_1, i.e., 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. 5 (A)). 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 $\boxed{u, v}$;
(A.2) $u \prec_a x$. In this case, the nodes between u and v in L_2 must be as $\boxed{u, \ldots, y, v}$, where $u \prec_d y \prec_d v$.
Therefore in Case A, z does not exist and $C_1 = x = w = y$, thus we have $u \preceq_d w \preceq_d C_1 \preceq_d y \prec_d v$.
(Case B) $x \prec_a y$ (Fig. 5 (B)). In this case, all nodes between u and v is as $\boxed{u, \ldots, x, \ldots, z, \ldots, y, v}$, where z is the child node on the path from x to y and $u \preceq_d x \prec_d z \preceq_d y \prec_d v$. Since $C_1 = x = w$, we have $u \preceq_d w \preceq_d C_1 \prec_d z \preceq_d y \prec_d v$.
(Case C) $x \prec_d y$ and $x \not\prec_a y$ (Fig. 5 (C)). In this case, the nodes between u and v must be as $\boxed{u, \ldots, w, \ldots, z, \ldots, y, v}$, where $u \preceq_d w \prec_d z \preceq_d y \prec_d v$, x satisfies that $w \prec_d x \prec_d z$. Since $C_1 = x$, we have $u \preceq_d w \prec_d C_1 \prec_d z \preceq_d y \prec_d v$.

By summarizing all the above three cases, we have $u \preceq_d w \preceq_d C_1(\prec_d z) \preceq_d y \prec_d v$. 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, $u \preceq_d w \preceq_d C_k(\prec_d z) \preceq_d y \prec_d v$. ■

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

5.3.2 The BwdSLCA$^+$ Algorithm

The optimized backward algorithm, i.e., BwdSLCA$^+$, is the same as BwdSLCA except that in function bwdGetCA, bwd-

Algorithm 3: BwdSLCA$^+(Q)$

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

Function bwdBinSearch$^+(L_j, u, parent)$

/*Only differences from bwdBinSearch are shown here*/

1 $\{s,e\} \leftarrow$ setInterval(L_j, u)

Function setInterval(L_j, u)

1 **if** $(C_j.ID = u.ID)$ **then** $\{s \leftarrow pos(C_j); e \leftarrow s;$ **return** $\{s,e\}\}$
2 $e \leftarrow pos(C_j)$
3 $s \leftarrow pos(L_j[C_j.PIDPos])$
4 **while**$(L_j[s] > u.ID)$ **do**
5 $\quad \{e \leftarrow s; s \leftarrow pos(L_j[L_j[s].PIDPos])\}$
6 **endwhile**
7 **return** $\{s,e\}$

BinSearch 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. 5 (A). 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.

Example 5 For query $Q = \{k_1, k_2\}$ and its two IDLists in Fig. 3, C_1 and C_2 point to node 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. 6 (A)), 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. 6 (B)). 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, i.e., 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. 6 (C)). 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, i.e., node 3, then output node 3 as an SLCA result. After that, as node 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, i.e., $O(m \cdot |L_1| \cdot \log(|L_m|))$.

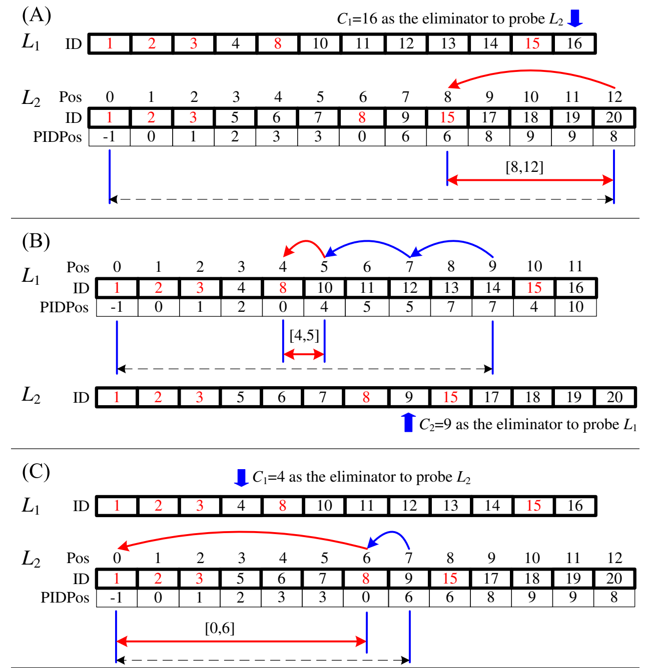

Fig. 6: Reducing the search intervals for different probe operations.

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 with two vectors, one is $N = \langle N_1, N_2, \ldots, N_m \rangle$ denoting the N_{Desc} values of all keywords in Q, the other is $n = \langle n_1, n_2, \ldots, n_m \rangle$ used to online add up the N value of e's child nodes in the CA tree. According to Lemma 3, when e is popped from S, if $\nexists i \in [1,m]$, 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 $flag =$ 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

Algorithm 4: FwdELCA(Q)

```
1  while (¬ fwdEof(L_i)) do
2    {flag, parent, v} ← fwdGetCA(Q);
3    if (flag = TRUE) then
4      while (¬ isEmpty(S) and top(S).ID ≠ parent.ID)) do
5        w ← pop(S)
6        modTop(S, w)
7        if (isELCA(w)) then output w.ID as an answer;
8      endwhile
9      push(S, v);
10   else break
11   foreach (i ∈ [1, m]) do fwdAdvance(C_i)
12 endwhile
13 while (¬ isEmpty(S)) do
14   w ← pop(S)
15   modTop(S, w)
16   if (isELCA(w)) then output w.ID as an answer;
17 endwhile
Function modTop(S, v)
1  foreach (i ∈ [1, m]) do top(S).n_i ← top(S).n_i + v.N_i
Function isELCA(w)
1  if(∄i ∈ [1, m], such that w.N_i = w.n_i) then return TRUE
2  return FALSE;
```

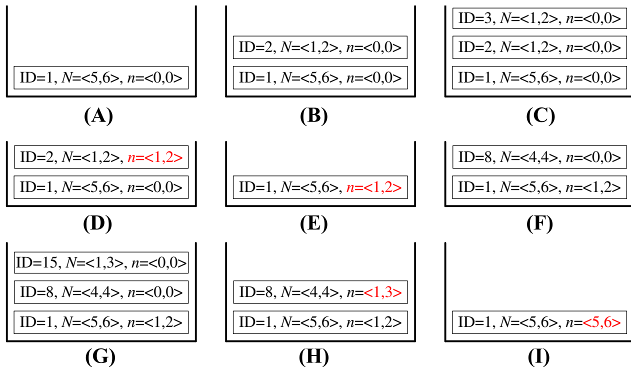

Fig. 7: Forward ELCA computation of Example 6.

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 TRUE (line 7). Note that when each element is pushed into stack S, all its n_i values are initialized to 0.

Example 6 For query $Q = \{k_1, k_2\}$ and its IDLists in Fig. 3, as shown in Fig. 7 (A)-(C), since node 1 is the parent of node 2, which in turn is the parent of node 3, the three CA nodes, i.e., node 1, 2 and 3, are pushed into stack S. The next CA node returned from fwdGetCA is node 8. Since node 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. 7 (D) and (E); then we check the satisfiability of node 3 and node 2 according to Lemma 3, and output node 3 as an ELCA node. Finally, node 8 and node 15 are pushed into S (Fig. 7 (F) and (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 node 1 and 2 are not ELCA nodes, since each N_i of node 1 and 2 is equal to their n_i (see Fig. 7 (D) and (I)). Therefore, the set of ELCA results are node 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, i.e., $O(m \cdot |L_1| \cdot \log(|L_m|))$.

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 $u' \preceq_p v$, 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 $u' \preceq_d v \prec_d v'$.*

Proof As shown in Fig. 8, there are three kinds of positional relationships for u', v', u and v: Case 1 (Fig. 8 (A)), u' has no other descendant CA nodes except v', thus the next CA node must be u' itself, i.e., $u' = v \prec_d v'$. Case 2 (Fig. 8 (B)), 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. 8 (C)), 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 $u' \preceq_d v \prec_d v'$. ■

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*

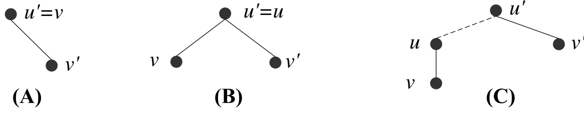

Fig. 8: The positional relationships for CA nodes, each solid (dashed) line from u to v means $u \prec_p v (u \prec_a v)$.

Algorithm 5: BwdELCA(Q)

1 **while** ($\neg$ bwdEof(L_i)) **do**
2 $\{flag, parent, v\} \leftarrow$ 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 $w \leftarrow$ pop(S)
11 **if**(isELCA(w)) **then** output $w.ID$ as an answer
12 **if**(isEmpty(S)) **then**
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 **endif**
23 **else break**
24 **foreach** ($i \in [1, m]$) **do** bwdAdvance(C_i)
25 **endwhile**

Function pushParent($S, parent, v$)

1 **foreach**($i \in [1, m]$) **do** $parent.n_i \leftarrow v.N_i$
2 push($S, parent$)

Function bwdGetCA() /*Based on fwdGetCA*/

2 $k \leftarrow \text{minarg}_i\{C_i.ID\}$
7 bwdBinSearch(L_j, C_k)
8 **if** ($pos(C_j) = -1$) **then return** {FALSE, NULL, NULL}

Function bwdBinSearch(L_j, u) /*Based on fwdBinSearch*/

1 $\{s, e\} \leftarrow$ setInterval(L_j, u)
8 **if** ($s > e$) **then** $C_j \leftarrow L_j[e]$

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. 8 (A)), 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. 8 (B), we add up the N value of v to the top element of S in line 19; otherwise (the case of Fig. 8 (C)), we directly push v's parent into S in line 20. Note that in the BwdELCA algorithm, we use the optimization technique proposed in Section 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. 9 (A)-(D). The first CA node is 15, which corresponds to Fig. 8 (C), thus 15 is output as an ELCA node, and its parent, i.e., node 8, is pushed into stack S (Fig. 9 (A)). The next CA node is node 8, which corresponds to Fig. 8 (A), 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, i.e., node 1, is pushed into S (Fig. 9 (B)). The third CA node is 3, which corresponds to Fig. 8 (C), thus node 3 is a leaf node of the CA tree, and it is output as an ELCA node, then its parent, i.e., node 2, is pushed into S (Fig. 9 (C)). The next CA node is 2, which corresponds to Fig. 8 (A), 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 CA node is 1, which also corresponds to Fig. 8 (A). Thus node 1 is popped out from the stack. After that, the processing stops. □

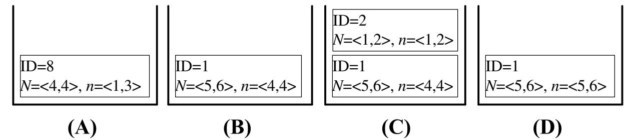

Fig. 9: Backward ELCA computation of Example 7.

Time Complexity: Same as the FwdELCA algorithm, the time complexity of BwdELCA is $O(m \cdot |L_1| \cdot \log(|L_m|))$.

7 Combining IDList with Hash Search

All algorithms in Sections 5 and 6 rely on the probe operation (implemented by binary search) to align the cursors of inverted lists, hence leading to a $O(\log |L_m|)$ term in the time complexity. To further improve the overall performance, we

(A) H_{Freq}

Key: *keyword*	k_1	k_2	...
Value: *Frequency*	12	13	...

(B) H_{k_1}

Key: *ID*	1	2	3	4	8	10	11	12	13	14	15	16
Value: *Count*	5	1	1	1	4	3	1	2	1	1	1	1

(C) H_{k_2}

Key: *ID*	1	2	3	5	6	7	8	9	15	17	18	19	20
Value: *Count*	6	2	2	2	1	1	4	1	3	2	1	1	1

Fig. 10: Illustration of the hash tables used in our methods. (A) is the hash table H_{Freq}, which records for each keyword, the length of its IDList; (B) and (C) are the two hash tables corresponding to the two IDLists of k_1 and k_2, and denoted as H_{k_1} and H_{k_2}, respectively. $H_{k_i}(i \in [1,2])$ 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.

consider the existence of *additional* hash indexes [32] 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 [32], except that in [32], H_{Freq} records, for each keyword k_i, the length of its inverted list of IDDewey labels, i.e., $|L_i^D|$.

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. 10(A). 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. 10(B)). Similarly, node 4 does not contain k_2 in its subtree, which can be denoted as $4 \notin H_{k_2}$ (Fig. 10(C)).

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 Section 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, i.e., 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 $Q = \{k_1, k_2, ..., k_m\}$ 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 node 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: (Fig. 11 (A), $x \prec_p u$, 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, i.e., $u \prec_d v' \prec_d y$ 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: (Fig. 11 (B), $z \prec_p x$, u is the last child node of x, y is the next sibling node of x, $y.PIDPos = p_z < u.PIDPos = p_x$). 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 $u.PIDPos = p_x > v'.PIDPos = y.PIDPos = p_z$. Therefore, y is not the first node after u, such that $y.PIDPos \leq u.PIDPos$, which contradicts the assumption.

Case 3: (Fig. 11 (C), $q \prec_a z \wedge q \prec_p y$, 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, $y.PIDPos = p_q < u.PIDPos = p_x$). 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, $u.PIDPos = p_x > v'.PIDPos \geq y.PIDPos = p_q$, and y is not the first node after u, such that $y.PIDPos \leq u.PIDPos$, which contradicts the assumption.

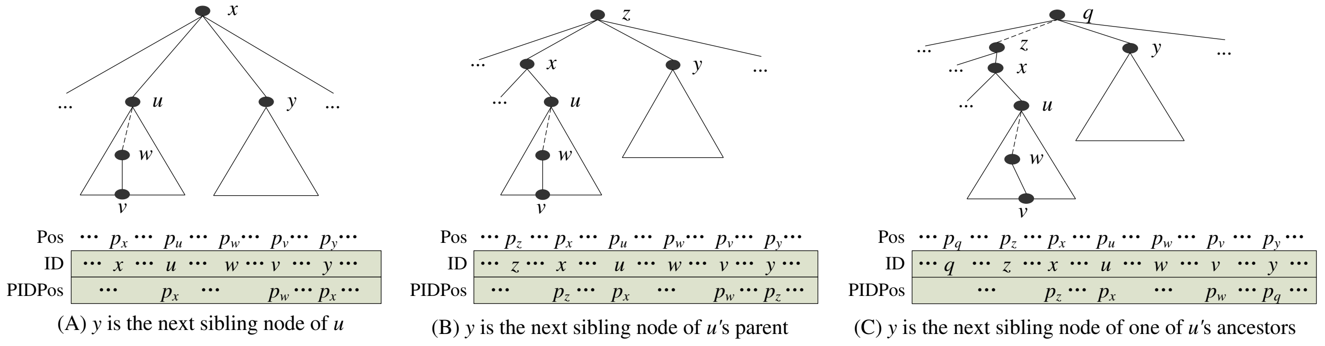

Fig. 11: Positional relationships between nodes in an IDList L_i, where u is a child non-CA node of x, y is the first node after u in L_i, such that $y.PIDPos \leq u.PIDPos$. The corresponding IDList for each sub-figure is shown under it.

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 don't 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, then checks whether C_1 is a CA node by calling Function isCA($C_1.ID$) 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 to 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, i.e., node 1, 2 and 3, then find a non-CA node, i.e., node 4. As node 4 does not have other descendant 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 node 10's parent, i.e., $parPos = 4$, then sequentially processes each node after node 10 by comparing the position value of its parent with *parPos*. If $parPos < C_1.PIDPos$, we directly move C_1 to the next node. In such a way, node 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. □

Algorithm 6: FwdSLCA-HS(Q)

/*$Q = \{k_1, ..., k_m\}$, $0 < |L_1| \leq |L_2| \leq ... \leq |L_m|$*/
1 $u \leftarrow \{-1, -1\}$ /*u denotes the previous CA node*/
2 $parPos \leftarrow -1$
3 **while** $(\neg$ eof$(L_1))$ **do**
4 $\quad parPos \leftarrow C_1.PIDPos$
5 $\quad$ **if** $(isCA(C_1.ID))$ **then**
6 $\quad\quad$ **if** $(L_1[C_1.PIDPos].ID \neq u.ID)$ **then**
7 $\quad\quad\quad$ output $u.ID$ as an answer
8 $\quad\quad$ **endif**
9 $\quad\quad u \leftarrow C_1$
10 $\quad\quad$ fwdAdvance(C_1)
11 $\quad$ **else**
12 $\quad\quad$ fwdAdvance(C_1)
13 $\quad\quad$ **while** $(parPos < C_1.PIDPos)$ **do**
14 $\quad\quad\quad$ fwdAdvance(C_1)
15 $\quad\quad$ **endwhile**
16 $\quad$ **endif**
17 **endwhile**
18 output $u.ID$ as an answer

Function isCA(id_v)

1 **if** $(\exists k_i \in Q$, such that $id_v \notin H_{k_i})$ **then return** FALSE
2 **return** TRUE

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 not CA nodes, then any node w satisfying $u \prec_d w \prec_d v$ cannot be a CA node of Q.*

Algorithm 7: BwdSLCA-HS(Q)

```
/*Q = {k_1,...,k_m}, |L_1| ≤ |L_2| ≤ ... ≤ |L_m|*/
1  u ← {−1,−1}      /*u denotes a non-leaf CA node*/
2  prePos ← |L_1|; curPos ← |L_1|
3  while (¬ eof(L_1)) do
4    if (u ≠ C_1 ∧ curPos ≠ pos(C_1) ∧ ¬isCA(C_1.ID)) then
5      prePos ← pos(C_1)
6      C_1 ← L_1[C_1.PIDPos]
7    else
8      curPos ← pos(C_1)
9      if (curPos = prePos − 1) then
10       if (u ≠ C_1) then
11         output C_1.ID as an answer
12       endif
13       u ← L_1[C_1.PIDPos]
14       prePos ← curPos
15       C_1 ← L_1[curPos − 1]
16     else
17       C_1 ← L_1[prePos − 1]
18     endif
19   endif
20 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. *curPos* always points to v, *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 *curPos* and *prePos* point to a virtual node after the last node of L_1 (line 2).

In each iteration (line 4 to 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 *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 ($u = C_1$). 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: $curPos = prePos - 1$ (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: $curPos \neq prePos - 1$, it means that some nodes between C_1 and $L_1[prePos]$ have not been processed, thus we move C_1 to the node before *prePos* (line 17).

Case 3: C_1 points to the CA node identified most recently (pointed by *curPos*, $curPos = pos(C_1)$). In this case, we process C_1 in two sub-cases:

Case 3.1: $curPos \neq prePos - 1$ (line 9 returns FALSE), it means that some nodes between C_1 and $L_1[prePos]$ have not been processed. We just move C_1 to the node before *prePos* (line 17).

Case 3.2: $curPos = prePos - 1$ (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: $C_1 \in CA(Q) \wedge u \neq C_1$, and C_1 is not equal to a previously identified CA node ($curPos \neq pos(C_1)$). We firstly set $curPos = pos(C_1)$ 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 *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, $u = \{-1,-1\}, prePos = curPos = 12$, C_1 points to node 16. In the first iteration (Fig. 12 (A), Case 1), as node 16 is not a CA node, it moves C_1 to its parent node, i.e., node 15 (line 6). In the second iteration (Fig. 12 (B), 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, $prePos = 10$, and C_1 points to node 14. The next three iterations (Fig. 12 (C) to (E), Case 1) are similar to the first iteration. After that, C_1 points to node 8, *prePos* points to node 10. In the 6th iteration (Fig. 12 (F), Case 2.1), since $u = C_1$, it makes u point to node 1, the parent of node 8 (line 13), and it makes *prePos* point to node 8 (line 14) and C_1 point to node 4 (line 15). In the 7th iteration (Fig. 12 (G), Case 1), since node 4 is not a CA node, *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. 12 (H), Case 4), it outputs node 3 as an SLCA result, and makes u point to node 2, the parent of node 3 (line 13), *prePos* point to node 3 (line 14) and C_1 point to node 2 (line 15). The next two iterations (Fig. 12 (I) and (J), Case 2.1) are processed in the same way as the 6th iteration (Fig. 12 (F)). 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

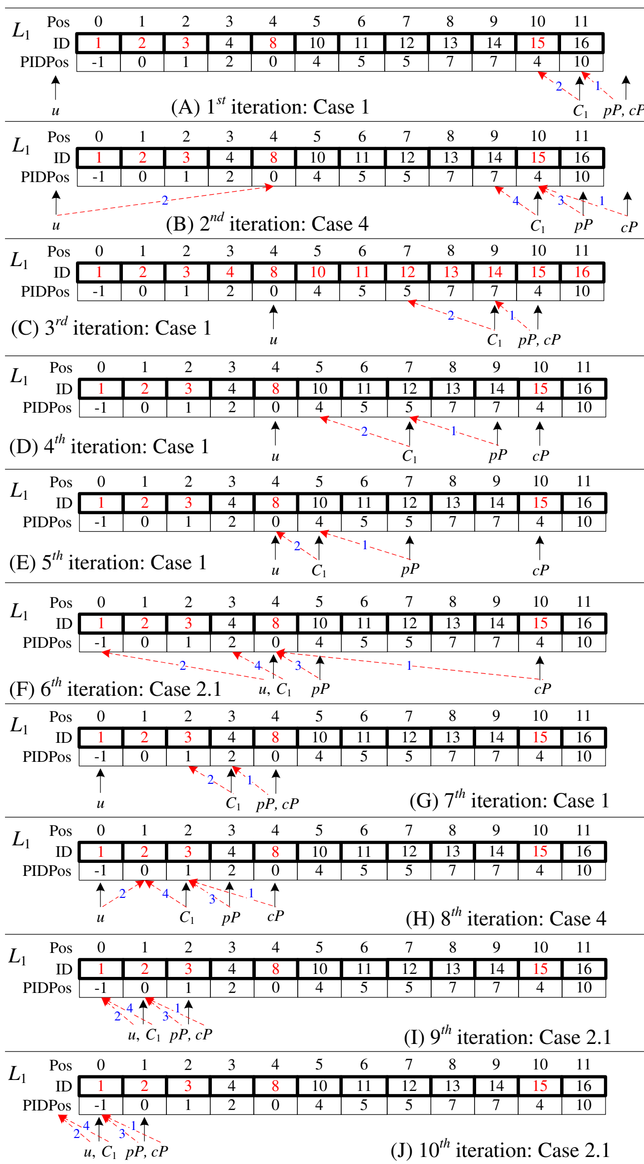

Fig. 12: Backward ELCA computation of Example 9, $pP = prePos$, $cP = curPos$, blue numbers on dashed arrows denote the processing orders of the corresponding iteration.

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, i.e., node 1, 2, 3, 4 and 8, and node 5, 6, 7, 9, 10, 11 and 12 are processed without hash probe operations. Even though node 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, i.e., node 6, 9, 10 and 11, and the two CA nodes, i.e., node 1 and 2, are processed

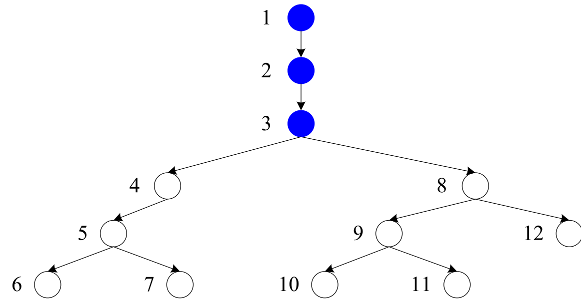

Fig. 13: A sample XML tree consists of nodes in L_1 for query $Q = \{k_1, k_2\}$, where each number is a node ID, node 1 to 3 are CA nodes, node 4 to 12 are non-CA nodes.

without hash probe operations. However, it still needs to process node 3, 4, 5, 7, 8 and 12 by hash probe operations, which is not as efficient as FwdSLCA-HS. □

To take advantage 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 L_1 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 $v \in L_1$ 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 isn't 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: $v \notin CA(Q) \wedge u \in CA(Q)$. We process all nodes between u and v using FwdSLCA-HS.

Case 3: $v \notin CA(Q) \wedge u \notin CA(Q)$. 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 $Q = \{k_1, k_2\}$ 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, i.e., node 8, is a CA node. As node 8 is not a CA node (Case 3), HybSLCA-HS directly skips node 9 to 11. As the parent of node 8 is node 3, which is a CA node (Case 2), we process all nodes between node 3 and 8 using FwdSLCA-HS. Therefore, only node 4 is processed using hash probe operations. After that, node 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 L_1 into the following mutually disjoint subsets.

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

Node set			FwdSLCA-HS	BwdSLCA-HS	HybSLCA-HS
S_1			$\blacktriangleright$	$\triangleright$	$\triangleright$
S_2			$\blacktriangleright$	$\blacktriangleright$	$\blacktriangleright$
S_3			$\blacktriangleright$	$\blacktriangleright$	$\blacktriangleright$
S_4	S_{4_1}	S'_{4_1}	$\triangleright$	$\blacktriangleright$	$\blacktriangleright$
		S''_{4_1}	$\triangleright$	$\blacktriangleright$	$\triangleright$
	S_{4_2}	S'_{4_2}	$\triangleright$	$\curvearrowright$	$\curvearrowright$
		S''_{4_2}	$\triangleright$	$\curvearrowright$	$\triangleright$

S_1: the set of non-leaf CA nodes in the CA tree.
S_2: the set of leaf CA nodes (SLCA nodes).
S_3: the set of non-CA nodes, of which each one is a child of a CA node.
S_4: 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.

- S_{4_1}: let $\mathscr{F}_u = \{T_{v_1}, T_{v_2}, ..., T_{v_n}\}$ be the set of subtrees that are rooted at all child nodes of u, where $u \in S_3, v_n$ is the rightest child node of u. S_{4_1} consists of all nodes on the rightest path of T_{v_n} for all nodes of S_3. S_{4_1} can also be further classified into two subsets.
  - S'_{4_1}: each node $v \in S'_{4_1}$ is in a subtree t_u, $u \in S_3$ is the rightest node among its sibling nodes.
  - S''_{4_1}: the set of nodes of S_{4_1} after excluding S'_{4_1}.
- S_{4_2}: the set of nodes of S_4 after excluding S_{4_1}, which also consists of two subsets.
  - S'_{4_2}: each node $v \in S'_{4_2}$ is in a subtree t_u, $u \in S_3$ is the rightest node among its sibling nodes.
  - S''_{4_2}: the set of nodes of S_{4_2} after excluding S'_{4_2}.

Obviously, $L_1 = S_1 \bigcup S_2 \bigcup S_3 \bigcup S'_{4_1} \bigcup S''_{4_1} \bigcup S'_{4_2} \bigcup S''_{4_2}$ and $|L_1| = |S_1| + |S_2| + |S_3| + |S'_{4_1}| + |S''_{4_1}| + |S'_{4_2}| + |S''_{4_2}|$.

Example 12 Consider the XML tree in Fig. 13, which consists of all nodes in L_1 of $Q = \{k_1, k_2\}$. According to the above definition, $S_1 = \{1,2\}$, $S_2 = \{3\}$, $S_3 = \{4,8\}$, $S_4 = \{5,6,7,9,10,11,12\}$, $S_{4_1} = \{5,7,12\}$, S'_{4_1}=$\{12\}$, S''_{4_1}=$\{5,7\}$, S_{4_2}=$\{6,9,10,11\}$, S'_{4_2}=$\{9,10,11\}$, $S''_{4_2} = \{6\}$. □

Table 2 shows the comparison of our three hash based algorithms. Obviously, even though FwdSLCA-HS needs to process all nodes, all nodes of S_4 are processed without hash probe operations. Thus when $|L_1| \gg |\bigcup_{i=1}^{3} S_i|$, the overall performance of FwdSLCA-HS is dominated by the number of nodes of S_4; otherwise, its overall performance is dominated by the number of nodes in $\bigcup_{i=1}^{3} S_i$.

BwdSLCA-HS, on the other hand, does not need to process all nodes of L_1. 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_{4_2} 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_1} into costly hash probe operations.

As a comparison, HybSLCA-HS is more adaptive to various cases on average. When $|L_1| \gg |\bigcup_{i=1}^{3} S_i|$, compared with BwdSLCA-HS, HybSLCA-HS does not need to afford the cost of processing all nodes of S''_{4_1} using hash probe operations; and compared with FwdSLCA-HS, it can be as clever as BwdSLCA-HS to skip all nodes of S'_{4_2}. When $|S_4| \ll |\bigcup_{i=1}^{3} S_i|$, 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 $O(m \cdot |\bigcup_{i=1}^{3} S_i| + |S_4|)$, $O(m \cdot |S_2 \bigcup S_3 \bigcup S_{4_1}| + |S_1|)$ and $O(m \cdot |S_2 \bigcup S_3 \bigcup S'_{4_1}| + |S_1 \bigcup S''_{4_1} \bigcup S''_{4_2}|)$, 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 Section 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. Therefore, except FwdELCA-HS which has the same time complexity as FwdSLCA-HS, the worst-case of BwdELCA-HS and HybELCA-HS are $O(m \cdot |S_1 \bigcup S_2 \bigcup S_3 \bigcup S_{4_1}|)$ and $O(m \cdot |S_1 \bigcup S_2 \bigcup S_3 \bigcup S'_{4_1}| + |S''_{4_1} \bigcup S''_{4_2}|)$, respectively.

Since our current hash table H_{k_i} maintains for each ID id_v, the number of occurrence of k_i in the subtree rooted at v, the IDList for ELCA computation can be simplified by removing the third row in Fig. 3.

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 $O(d \cdot m \cdot (\sum_i^m |L_i^D|))$, while the time complexity of IL, IS, IMS and JDewey is $O(d \cdot m \cdot |L_1^D| \cdot \log |L_m^D|)$. Since $|L_i| \leq d \cdot |L_i^D|$ due to the sharing of common ancestors in IDList,

our algorithms, i.e., 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 Section 5 and 6 to accelerate the overall performance.

Compared with existing methods that use hash table in Table 1, i.e., HS and HC, according to their time complexities, i.e., $O((\log d \cdot m + d) \cdot |L_1^D|)$ and $O(d \cdot m \cdot |L_1^D|)$, we know that their overall performance is dominated by the length of L_1^D. On the contrary, our algorithms, i.e., 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, i.e., sharing of common ancestors in IDList. And according to the time complexities of our methods (see Section 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, 15, 17], 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:

- **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) [25], Probabilistic Intersection (PI) [22], 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 don't 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 subgroups: (Group 2.1) algorithms that are not based on hash search, including Stack [27], IL [27], IMS [23], JDewey [4], and the three IDList based algorithms, i.e., FwdSLCA, BwdSLCA and BwdSLCA$^+$; (Group 2.2) algorithms that are based on hash search, including HS [26], 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 [28], JDewey [4], FwdELCA and BwdELCA; (Group 3.2) algorithms that are based on hash search, including HC [32], FwdELCA-HS, BwdELCA-HS and HybELCA-HS.

All algorithms were implemented using Microsoft VC++. To make a fair comparison, we only evaluate 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 DB[4] 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 keyword query Q, the set of inverted lists are firstly loaded into memory, and the running time is the averaged 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 fashion. In the interest of space, we mainly focus on results of the 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_{L^D} 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).

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.

[2] http://www.monetdb.org/Home
[3] http://www.informatik.uni-trier.de/~ley/db/
[4] http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html

Table 3: Statistics of keywords used in our experiment.

Keyword	tissue	baboon	necklace	arizona	cabbage	hooks	shocks	patients	cognition	villages
$\|L^D\|$	384	725	200	451	366	461	596	382	495	829
N_{LD}	3,252	6,013	1,704	2,255	3,016	3,869	4,823	3,306	4,192	6,981
$\|L\|$	2,281	4,014	1,198	1,355	2,071	2,674	3,302	2,309	2,857	4,867

Keyword	male	takano	order	school	check	education	female	province	privacy	gender
$\|L^D\|$	18,441	17,129	16,797	23,561	36,304	35,257	19,902	33,520	31,232	34,065
N_{LD}	113,081	100,338	116,429	143,894	213,449	187,843	113,081	173,900	129,455	177,450
$\|L\|$	62,162	53,324	68,049	91,138	106,974	115,318	70,747	105,795	66,244	108,098

Keyword	bidder	listitem	keyword	bold	text	time	date	emph	incategory	increase
$\|L^D\|$	299,018	304,969	352,121	368,544	535,268	313,398	457,232	350,560	411,575	304,752
N_{LD}	1,196,072	2,353,169	3,072,825	3,218,152	4,086,735	1,616,385	2,463,062	3,057,771	2,057,875	1,542,891
$\|L\|$	353,220	574,800	1,236,016	1,270,061	1,670,362	730,709	1,112,961	1,229,698	520,333	683,370

Table 4: Queries used for CA computation.

ID	Keywords	# of Results	Group
Q1	bold, increase	34,136	G1
Q2	bold, increase, text	34,136	
Q3	bold, increase, text, keyword	25,346	
Q4	bold, increase, text, keyword, emph	20,766	
Q5	date, listitem	50,706	G2
Q6	date, listitem, bidder	16,355	
Q7	date, listitem, bidder, time	16,355	
Q8	date, listitem, bidder, time, text	16,355	

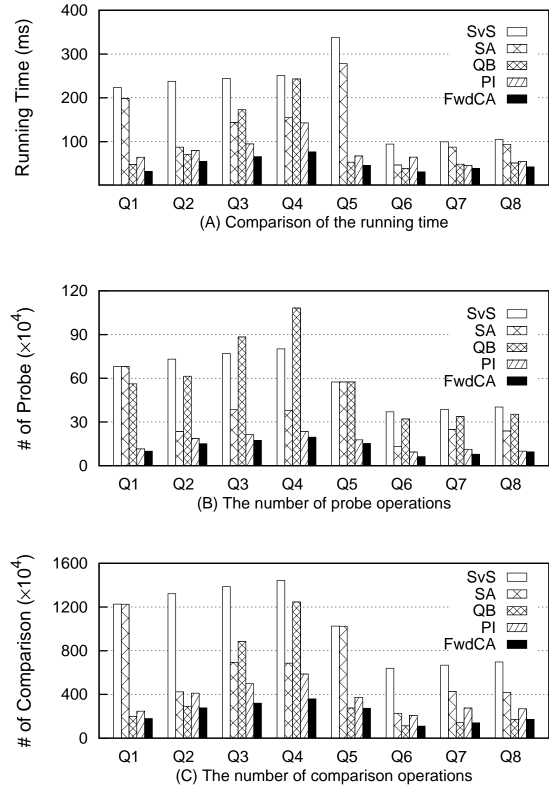

Fig. 14: Comparison of CA computation with existing methods.

9.3.1 Impacts of Different Probe Order

As shown in Fig. 14 (A), 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. 14 (B) and Fig. 14 (C), which show the number of probe and comparison operations, respectively. From Fig. 14 (B) 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. 14 (A), QB performs similarly to or more efficient than PI in many cases, the reason can be further explained by referring to Fig. 14 (C). 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. 14 (C), our method achieves the best performance, the reason lies in the least number of comparison operations.

9.3.2 Impacts of Reducing Search Interval

Fig. 15 (A) 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. 15 (B), BwdCA reduces many probe operations for Q1 to Q4, which can be verified according to Fig. 15 (C), i.e., 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. 15 (C), we find

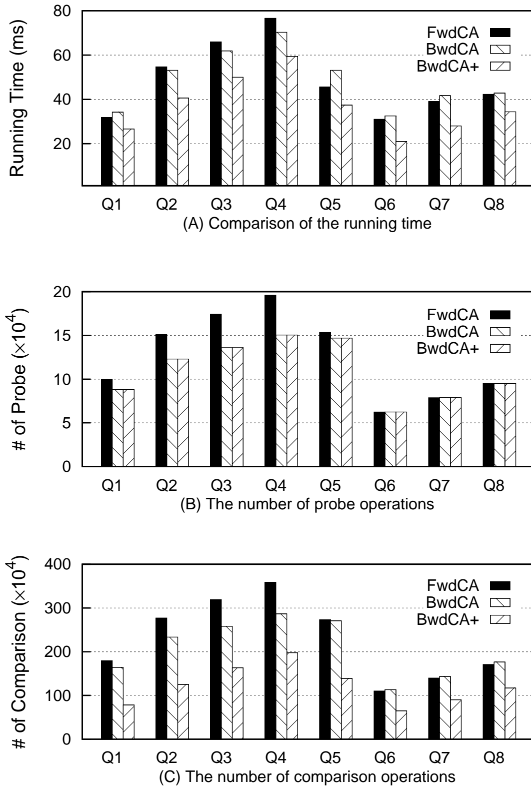

Fig. 15: Comparison of CA computation between our methods.

Table 5: The composition of the comparison operation for BwdCA$^+$.

#	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
N_1	534,767	865,241	1,169,748	1,450,364	1,026,313	512,245	719,006	915,088
N_2	213,617	318,335	385,424	439,540	311,322	102,162	132,571	189,292
N_3	34,136	68,272	76,038	83,064	50,706	32,710	49,065	65,420

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. 15 (C) 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.

Table 6: Queries used for SLCA and ELCA computation on 582MB XMark dataset, $R_S(R_E)$ denotes the number of SLCA (ELCA) results.

ID	Keywords	R_S	R_E	Freq.
Q9	villages,hooks	9	9	Low
Q10	baboon,patients,arizona	1	1	
Q11	cabbage,tissue,shocks,baboon	9	9	
Q12	shocks,necklace,cognition,cabbage,tissue	9	9	
Q13	female,order	570	579	Med
Q14	privacy,check,male	29	34	
Q15	takano,province,school,gender	107	108	
Q16	school,gender,education,takano,province	107	108	
Q17	bold,increase	34,136	34,189	High
Q18	date,listitem,emph	43,777	43,792	
Q19	incategory,text,bidder,date	1	1	
Q20	bidder,date,keyword,incategory,text	1	1	
Q21	incategory,cabbage	224	224	Random
Q22	province,bold,increase	427	436	
Q23	listitem,emph,arizona	1	1	
Q24	bold,increase,hooks,takano	6	7	
Q25	emph,arizona,villages,education	1	1	
Q26	check,bidder,date,baboon	1	2	

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 Section 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, i.e., 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, i.e., 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

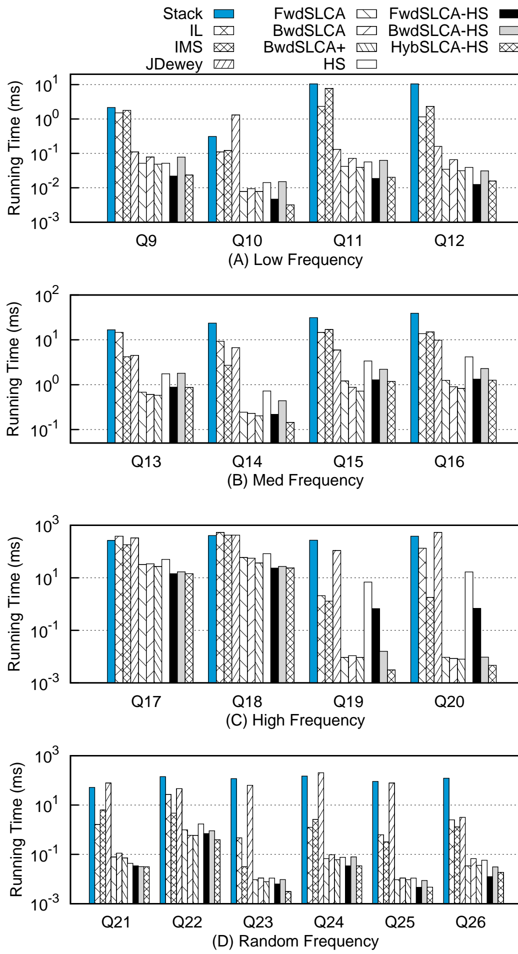

Fig. 16: Comparison of running time for SLCA computation.

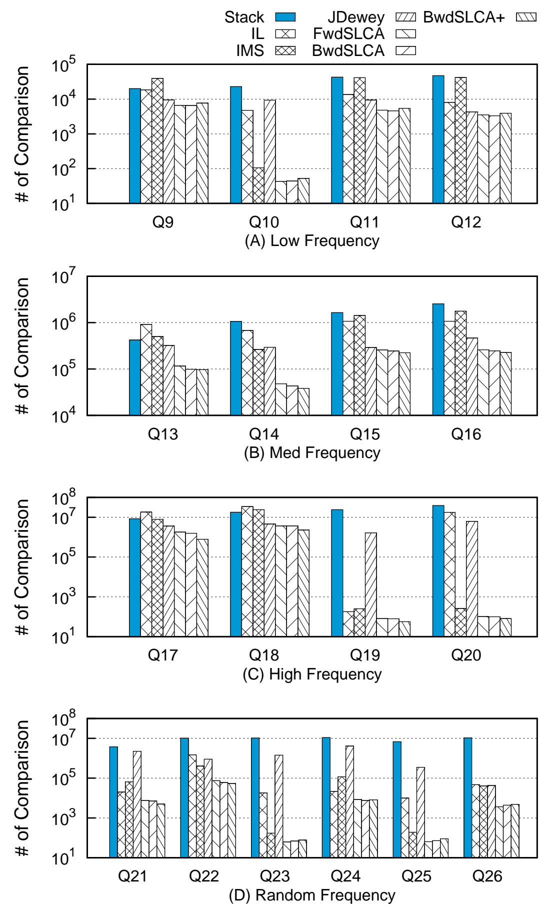

Fig. 17: Number of comparison operations.

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

Fig. 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, e.g., Q21 to Q26 in Fig. 16 (D); 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 not be as efficient as that of Stack, e.g., 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$^+$ 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$^+$ is less than that of FwdSLCA and BwdSLCA, (3.2) even in some cases, BwdSLCA$^+$ needs more comparison operations, as shown in Table 5, 15% to 30% comparison operations exist in the setInterval function, which can be done much more efficient than the comparison operations in

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. For HS algorithm, N_C is the number of comparison operations between integers for LCA computation; while for FwdSLCA-HS, BwdSLCA-HS and HybSLCA-HS, it denotes the number of nodes that are processed without hash probe operations.

Query	N_H				N_C			
	HS	FwdSLCA-HS	BwdSLCA-HS	HybSLCA-HS	HS	FwdSLCA-HS	BwdSLCA-HS	HybSLCA-HS
Q9	1899	471	2,437	509	483	1,968	2	1,930
Q10	764	1	4	5	382	1,354	0	0
Q11	3,361	396	1,825	407	388	1,433	2	1,402
Q12	2,383	240	1,116	250	222	881	2	841
Q13	61,044	17,753	59,530	17,775	18,875	44,412	1,323	44,390
Q14	31,937	2,651	12,646	2,369	18,528	58,244	10	9,938
Q15	153,856	18,317	53,878	16,309	17,449	37,033	3	37,015
Q16	198,913	18,435	54,331	16,309	17,449	37,033	3	37,015
Q17	1,057,672	241,253	438,336	241,088	413,176	438,435	17,877	438,600
Q18	2,216,869	185,599	379,864	124,926	472,010	449,936	9,355	449,874
Q19	299,018	1	3	4	299,018	353,219	0	4
Q20	897,054	3	7	4	299,018	353,219	0	4
Q21	1,060	458	1,001	463	1,254	1,351	8	981
Q22	46,858	4,920	16,488	2,589	34,886	102,079	352	7,087
Q23	451	1	4	5	451	1,354	0	0
Q24	3,601	568	2,649	304	477	2,145	2	1,617
Q25	451	1	4	5	451	1,354	0	0
Q26	2,659	179	1,062	178	745	3,518	0	2,801

each probe operation. E.g., even though BwdSLCA$^+$ needs more comparison operations for some queries, it actually saves 5% to 45% cost compared with FwdSLCA and BwdSLCA for Q9 to Q26.

From Fig. 16, we have the following observations for algorithms of ***Group 2.2***, i.e., HS, FwdSLCA-HS, BwdSLCA-HS and HybSLCA-HS: (1) FwdSLCA-HS and HybSLCA-HS always beat HS for all queries; (2) BwdSLCA-HS outperforms HS for most queries, especialy 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. E.g., 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.

Fig. 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[5] over Stack is more sig-

[5] In Fig. 18, we take the result selectivity of existing methods as that of our method to make a fair comparison.

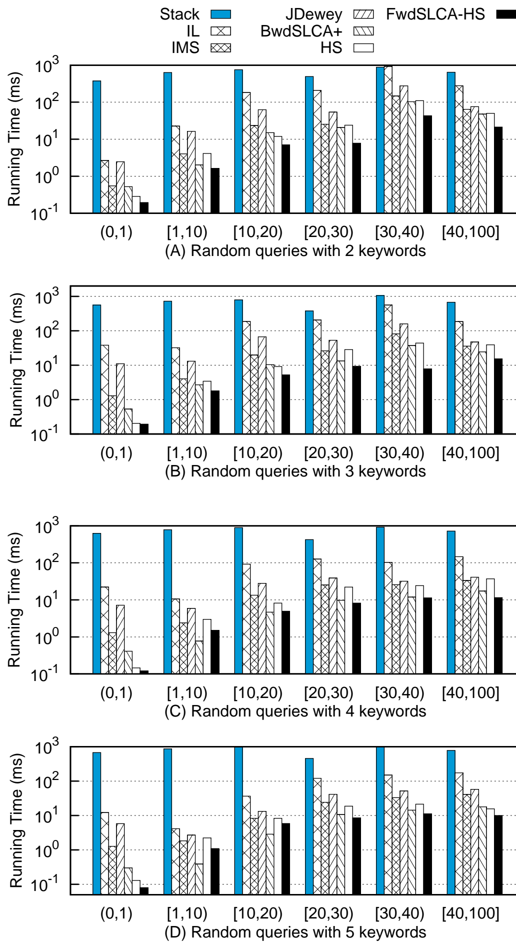

Fig. 18: Comparison of running time with different result selectivities.

Table 8: Queries used for SLCA computation on DBLP dataset, each number after a keyword denotes the number of Dewey labels in the corresponding inverted list.

Query	Keywords	# of Results
QD1	article(691464), book(10183)	1456
QD2	algorithm(42695), article(691464)	18349
QD3	data(66317), article(691464)	26611
QD4	article(691464), database(18630)	5753
QD5	xml(5323), article(691464)	1033
QD6	year(1695239), 2001(59663)	59355
QD7	book(10183), article(691464), mining(11853)	6
QD8	algorithm(42695), article(691464), 2001	521
QD9	article(691464), data(66317), mining(11853)	1563
QD10	data(66317), xml(5323), article(691464)	209

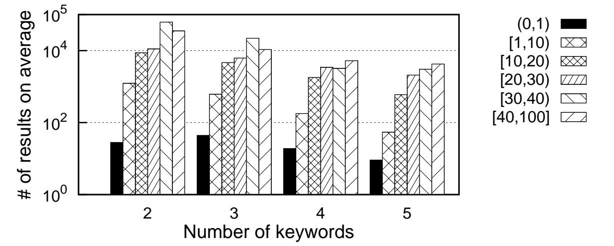

Fig. 19: Average number of results for each result selectivity.

nificant 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. 18 (A) and (B), with the increase of the result selectivity, the average time used by most methods for result 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 due to avoiding the CAR problem by adopting IDList, such that the number of processed nodes (see Section 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 subfigures 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.

Fig. 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).

Query	Stack	IL	IMS	JDewey	FwdSLCA	BwdSLCA	BwdSLCA$^+$	HS	FwdSLCA-HS	BwdSLCA-HS	HybSLCA-HS
QD1	149.6	8.91	2.66	35.9	0.94	0.93	*0.79*	0.41	0.29	0.39	*0.28*
QD2	168.4	42.97	42.34	45.71	*12.66*	15.16	13.28	14.99	5.48	6.86	*5*
QD3	174.8	61.56	67.03	48.27	15.63	18.75	*15.15*	18.03	7.81	9.28	*6.41*
QD4	156	16.09	8.44	38.78	*3.9*	4.38	3.91	2.22	1.68	2.32	*1.33*
QD5	149.8	5.47	2.34	34.03	0.63	0.78	*0.62*	0.31	0.26	0.44	*0.19*
QD6	477.4	99.22	107.5	207.99	52.34	*40.94*	43.59	23.81	23.78	21.8	*19.3*
QD7	184.2	4.84	1.25	35.61	0.31	*0.16*	0.31	0.38	0.34	0.42	*0.33*
QD8	205.8	24.07	11.41	41.57	*2.5*	2.81	3.13	6.38	5.25	7.71	*4.61*
QD9	199.6	14.68	6.56	43.04	*2.03*	2.5	2.35	1.73	1.34	2.11	*1.2*
QD10	194.3	4.38	2.81	42.34	*0.63*	0.78	0.78	0.33	0.28	0.53	*0.27*

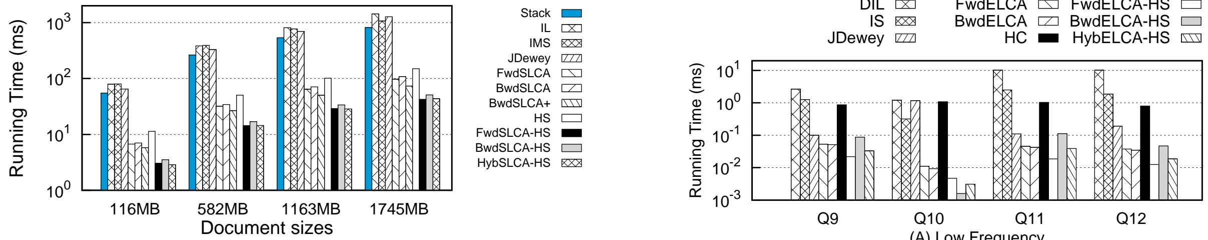

Fig. 20: Running time of Q17 on different XML documents.

9.5 ELCA computation

Fig. 21 shows the running time of different algorithms on queries Q9 to 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, e.g., 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.

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 achieves 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, because algorithms of Group 3.1 can utilize nodes in other lists to skip more useless nodes.

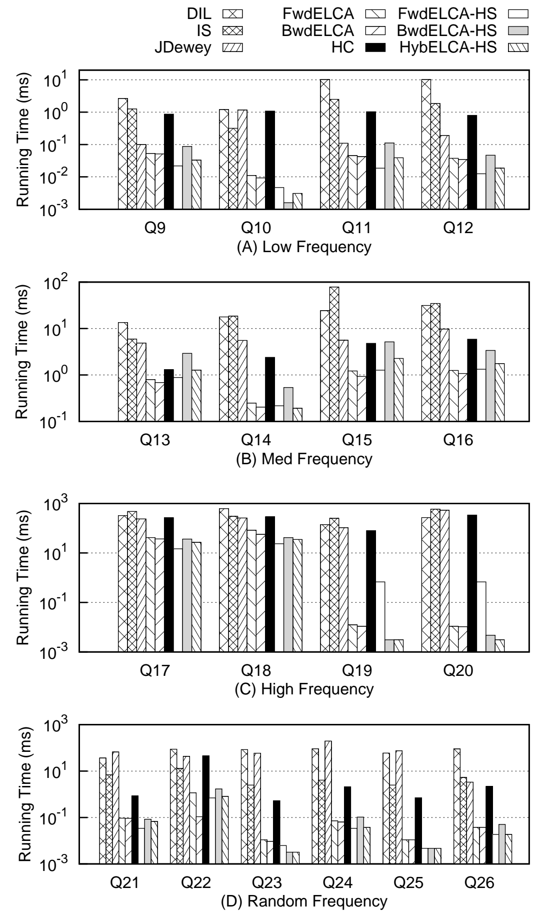

Fig. 21: Comparison of running time for ELCA computation.

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 opti-

mization 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, following problems are still needed to, and will be addressed in our future work.

(1) IDLists are tailored to XML keyword queries, and we need to consume more space to maintain indexes for both structured and keyword queries if we need to support the two kinds of query processing. Our future work will focus on designing algorithms on IDLists to answer structured queries, such that to have a general-purpose indexing scheme for building a realistic XML search engine for both kinds of queries.
(2) It is difficult to find the parent node by PIDPos if current IDList is organized as a B^+ tree, where each key is an ID value, and the data associated with this key is the corresponding entry in IDList. More importantly, the optimization technique introduced in Section 5.3 cannot be used to accelerate the probe operation in this case. We plan to implement a disk-based B^+ tree index to solve this problem in the future.
(3) Even though we focused on XML keyword query processing on in memory data and we did not discuss the compression issue, we need to address it after implementing the disk-based B^+ tree index to support all our algorithms proposed in this paper.

11 Acknowledgment

This research was partially supported by the grants from the Natural 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 Universities, the Research Funds of Renmin University (No. 11XNL01010XNI018), and the Research Funds from Education Department of Hebei Province (No. Y2012014). Zhifeng Bao was partially supported by the Singapore Ministry of Education Grant No. R252-000-394-112 under the project name of UTab, and Wei Wang was partially supported by ARC DP0987273 and DP0881779.

References

1. Z. Bao, T. W. Ling, B. Chen, and J. Lu. Effective xml keyword search with relevance oriented ranking. In *ICDE*, 2009.
2. J. Barbay, A. Lpez-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In *WEA*, pages 146–157, 2006.
3. J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3), 1976.
4. L. J. Chen and Y. Papakonstantinou. Supporting top-k keyword search in xml databases. In *ICDE*, 2010.
5. Y. Chen, W. Wang, and Z. Liu. Keyword-based search and exploration on databases. In *ICDE*, 2011.
6. Y. Chen, W. Wang, Z. Liu, and X. Lin. Keyword search on structured and semi-structured data. In *SIGMOD Conference*, pages 1005–1010, 2009.
7. S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. Xsearch: A semantic search engine for xml. In *VLDB*, 2003.
8. E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *SODA*, 2000.
9. E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX*, pages 91–104, 2001.
10. B. Ding and A. C. König. Fast set intersection in memory. *PVLDB*, 4(4):255–266, 2011.
11. D. K. Fisher, F. Lam, W. M. Shui, and R. K. Wong. Efficient ordering for xml data. In *CIKM*, pages 350–357, 2003.
12. L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *SIGMOD Conference*, 2003.
13. L. Kong, R. Gilleron, and A. Lemay. Retrieving meaningful relaxed tightest fragments for xml keyword search. In *EDBT*, pages 815–826, 2009.
14. C. Li, T. W. Ling, and M. Hu. Efficient updates in dynamic xml data: from binary string to quaternary string. *VLDB J.*, 17(3), 2008.
15. G. Li, J. Feng, J. Wang, and L. Zhou. Effective keyword search for valuable lcas over xml documents. In *CIKM*, 2007.
16. G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a tastier approach. In *SIGMOD Conference*, 2009.
17. Y. Li, C. Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, 2004.
18. Z. Liu and Y. Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD Conference*, 2007.
19. Z. Liu and Y. Chen. Reasoning and identifying relevant matches for xml keyword search. *PVLDB*, 1(1):921–932, 2008.
20. Z. Liu and Y. Chen. Processing keyword search on xml: a survey. *World Wide Web*, 14(5-6):671–707, 2011.
21. C. D. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.
22. V. Raman, L. Qiao, W. Han, I. Narang, Y.-L. Chen, K.-H. Yang, and F.-L. Ling. Lazy, adaptive rid-list intersection, and its application to index anding. In *SIGMOD Conference*, 2007.
23. C. Sun, C. Y. Chan, and A. K. Goenka. Multiway slca-based keyword search in xml data. In *WWW*, 2007.
24. I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD Conference*, 2002.
25. D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. *PVLDB*, 2(1), 2009.
26. W. Wang, X. Wang, and A. Zhou. Hash-search: An efficient slca-based keyword search algorithm on xml documents. In *DASFAA*, pages 496–510, 2009.
27. Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD Conference*, 2005.
28. Y. Xu and Y. Papakonstantinou. Efficient lca based keyword search in xml data. In *EDBT*, 2008.
29. H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.
30. C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, 2001.
31. J. Zhou, Z. Bao, W. Wang, T. W. Ling, Z. Chen, X. Lin, and J. Guo. Fast slca and elca computation for xml keyword queries based on set intersection. In *ICDE*, pages 905–916, 2012.
32. R. Zhou, C. Liu, and J. Li. Fast elca computation for keyword queries on xml data. In *EDBT*, 2010.