2012 IEEE 28th International Conference on Data Engineering

# Fast SLCA and ELCA Computation for XML Keyword Queries based on Set Intersection

Junfeng Zhou†, Zhifeng Bao‡, Wei Wang♯, Tok Wang Ling‡, Ziyang Chen†, Xudong Lin†, Jingfeng Guo†

†*Yanshan University,* ‡*National University of Singapore,* ♯*The University of New South Wales*

†{*zhoujf,zychen,xdlin,jfguo*}*@ysu.edu.cn,* ‡{*baozhife,lingtw*}*@comp.nus.edu.sg,* ♯*weiw@cse.unsw.edu.au*

***Abstract*— In this paper, we focus on efficient keyword query processing for XML data based on SLCA and ELCA semantics. We propose for each keyword a novel form of inverted list, which includes IDs of nodes that directly or indirectly contain the keyword. We propose a family of efficient algorithms that are based on the set intersection operation for both semantics. We show that the problem of SLCA/ELCA computation becomes finding a set of nodes that appear in all involved inverted lists and satisfy certain conditions. We also propose several optimization techniques to further improve the query processing performance. We have conducted extensive experiments with many alternative methods. The results demonstrate that our proposed methods outperform existing ones by up to two orders of magnitude in many cases.**

## I. Introduction

Keyword search on XML data has been received much attention in the literature [1]–[10]. Finding efficient query processing method for keyword search on XML data is an important topic in this area, as many applications demand fast query execution speed for many users simultaneously [11].

Over the past few years, researchers have proposed several semantics [1]–[5], [10] to define meaningful results for keyword queries. Among them, the most widely adopted semantics are arguably Smallest Lowest Common Ancestor (**SLCA**) [4] and Exclusive Lowest Common Ancestor (**ELCA**) [2], [3]. Since both semantics are defined based on the notion of lowest common ancestor (LCA), existing query processing algorithms for them [2]–[4], [7], [8], [12] are mainly based on the Dewey labeling [13]. The Dewey label of a node $v$ is the concatenation of all its ancestor nodes' local label on the path from the document root to $v$. Existing algorithms also make heavy use of two basic operations: (a) $OP1$: testing certain positional relationship of two nodes, and (b) $OP2$: computing the LCA of two nodes. However, the cost of performing an $OP1$ or $OP2$ operation based on Dewey labels is linear to the height of the XML tree, as each component of the Dewey label needs to be accessed and compared. If a node $v$ is a common ancestor of several keyword instance nodes, all the nodes on the path from the root to $v$ will be repeatedly visited, which we call as common-ancestor-repetition (CAR). The CAR problem is inherent in algorithms based on Dewey labels and causes much inefficiency in practice, as the following example shows.

*Example 1:* Take the IL algorithm [4] for example. It is one of the highly competitive algorithms for SLCA computation [4], [7]. The majority part of the algorithm is based on $OP1$ and $OP2$ operations. We ran it for the query $\{bold, increase\}$ on the XMark dataset[1] of 582MB. We found that the time spent on $OP1$ and $OP2$ operations occupies more than 80% of the total running time. Each common ancestor node is visited 272 times on average. □

To address the CAR problem, we propose a suite of novel and efficient algorithms for answering SLCA and ELCA queries that depart from existing Dewey encoding based approaches. We propose to assign each node a unique ID which is compatible with the document order. We also propose a new kind of inverted index, where for each keyword $k_i$, the corresponding inverted list consists of all nodes that contain $k_i$ in its subtree. We show that the SLCA and ELCA computation can be cast into a variant of the set intersection problem. We also design several optimization techniques that exploit the positional relationships between nodes in XML tree to accelerate the computation. In our extensive experimental study, we find that our methods outperform existing approaches by up to two orders of magnitude in many cases; we also provide an in-depth analysis on the major factors that impact the performance of previous approaches. Another salient feature of our proposed query processing methods is that they are actually simple to implement and can leverage any existing fast set intersection algorithms.

The rest of the paper is organized as follows. In Section II, we introduce background knowledge. In Section III, we introduce the CA tree and its properties. The inverted list and its properties are introduced in Section IV. The algorithms for SLCA and ELCA computation are discussed in Section V and VI, respectively. We present experimental results in Section VII and discuss related work in Section VIII. Section IX concludes the paper.

## II. Preliminaries

### A. 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 $k$ if $k$ appears in the node name, attribute name, or text value of $v$. Fig. 1 is a sample XML document. The positional relationships between two nodes include Document Order ($\prec_d$), Equivalence (=), AD

[1] http://monetdb.cwi.nl/xml

1084-4627/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDE.2012.75

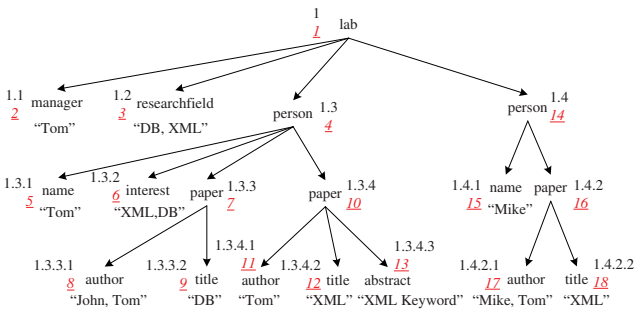


Fig. 1: A sample XML document.

(ancestor-descendant, $\prec_a$), PC (parent-child, $\prec_p$), Ancestor-or-self ($\preceq_a$) and Sibling relationship.

To accelerate the query processing, each node $v$ is usually assigned with a label that can uniquely represent $v$ and can be used to compute some positional relationships needed for query processing. Existing methods [2], [4], [7]–[9] usually assign to each node $v$ its Dewey label [13]. 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) which is compatible with the document order (we use the pre-order traversal number). For frequently updated XML data, we can use the encoding scheme of [14] to assign a unique ID to each node.

### B. Query Semantics

For a given query $Q = \{k_1, k_2, ..., k_m\}$ and an XML document $D$, inverted lists are often built to record which nodes directly contain which keywords. We use $L_i$ to denote the inverted list of $k_i$, of which all labels 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 $LCA(Q) = \{v | v = LCA(v_1, v_2, ..., v_m), v_i \in L_i (1 \leq i \leq m)\}$. E.g., the LCA nodes for $Q = \{XML, Tom\}$ on the XML document in Fig. 1 are nodes with IDs 1, 4, 10 and 16.

The most widely adopted variants of LCA are SLCA [4], [7] and ELCA [2], [3], [12]. SLCA defines a subset of $LCA(Q)$, of which no LCA in the subset is the ancestor of any other LCA, which can be formally defined as $SLCA(Q) = \{v | v \in LCA(L_1, L_2, \ldots, L_m)$ and $\nexists v' \in LCA(L_1, L_2, \ldots, L_m)$, such that $v \prec_a v'\}$. Consider the query $Q = \{XML, Tom\}$ on the sample XML document in Fig. 1. Although nodes 1 and 4 are LCAs, they are ancestors of node 10, and hence not SLCAs. The set of SLCAs of $Q$ are nodes 10 and 16.

The definition of ELCA is a bit more complex: a node $v$ is an ELCA node if the subtree $T_v$ rooted at $v$ contains at least one occurrence of all query keywords, after excluding the occurrences of the keywords in each subtree $T_{v'}$ that rooted at $v$'s descendant node $v'$ and already contains all of the query keywords, which can be formally defined as $ELCA(Q) = \{v | \exists v_1 \in L_1, ..., v_m \in L_m (v = LCA(v_1, ..., v_m) \wedge \forall i \in [1, m], \nexists x (x \in LCA(Q) \wedge child(v, v_i) \preceq_a x))\}$, where $child(v, v_i)$ is the child of $v$ on the path from $v$ to $v_i$. E.g., the matched ELCA nodes for $Q = \{XML, Tom\}$ on the XML document in Fig. 1 are those with IDs 1, 4, 10 and 16.

### C. Set Intersection Methods

Finding all the 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; and (2) the cost of each *probe operation*, i.e., how to efficiently find the *matched element*[2] for eliminator $e$, which is further affected by two orthogonal factors: (2.1) which *search method* is used (e.g., binary, galloping [15], or interpolation search [16]), and (2.2) how large the *search interval* is.

For *probe order*, the SvS algorithm [17] computes the final results by processing two lists each time from the shortest to the longest. It is a blocking algorithm and needs to buffer intermediate results. The Quantile-based algorithm (QB) [18] will encounter performance problems when the lists exhibit *locally skewed but globally uniform distribution*. In this case, the initial lists cannot be partitioned into sub-lists of large discrepancy in length, and QB's performance degrades accordingly. On the contrary, the Adaptive (Adaptive) [19], Small Adaptive (SA) [17], and Probabilistic Intersection (PI) [20] algorithms can dynamically adjust the probe order and output the results without buffering any intermediate results; however, they entail non-neglectable time overhead in choosing the probe order dynamically.

For *search interval*, assume that $n$ elements have been processed for the list $L_i$ to be probed, then the length of the search interval, i.e., the number of elements to be searched, is $|L_i| - n$. Obviously, the shorter the search interval is, the fewer comparisons are needed to complete the probe operation.

## III. Overview of Our Method

An important concept underlying our methods is Common Ancestor, which forms a superset of SLCA and ELCA results.

*Definition 1:* ***(Common Ancestor (CA))*** For a given 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 $Q = \{XML, Tom\}$, the CA nodes of $Q$ on the XML document in Fig. 1 are nodes 1, 4, 10, 14, and 16. Obviously, we have the following lemma.

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

[2]The *matched* element in $L_i$ to eliminator $e$ is the minimum element that is equal to or greater than $e$.

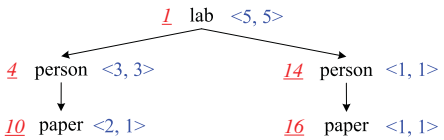


Fig. 2: A sample CA tree of $Q = \{XML, Tom\}$ on the XML document of Fig. 1. Each italic number beside a node $v$ denotes $v$'s ID, and $< N_1, N_2 >$ is the vector associated with $v$ denoting the number of nodes that directly contain "XML" and "Tom" in the subtree rooted at $v$.

*Definition 2:* ***(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 $Q = \{XML, Tom\}$ on the XML document of Fig. 1, based on which we can identify all SLCA and ELCA nodes using 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 $SLCA(Q) = V_T^{leaf}$.

*Proof:* Since each node $v \in V_T^{leaf}$ does not have other CA nodes as its descendant nodes, $v$ is an SLCA node, i.e., $V_T^{leaf} \subseteq SLCA(Q)$. For each node $v \in SLCA(Q) \subseteq V_T = CA(Q)$, assume that $v \notin V_T^{leaf}$, then there must exist at least one node $u \in V_T^{leaf}$, such that $v \prec_a u$, therefore $v$ is not an SLCA node, which contradicts the assumption. ■

For instance, the matched SLCA nodes for $Q = \{XML, Tom\}$ are the leaf nodes of its CA tree with IDs 10 and 16.

According to the definition of ELCA [2], we have the following lemma, which is similar to [3].

*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 if $\nexists i \in [1, m]$, such that $v.N_i = \sum_{j=1}^{l} v_j.N_i$.

For instance, the vector of a CA node for query $Q = \{XML, Tom\}$ is shown on its right in Fig. 2. According to Lemma 3, the ELCA nodes are nodes 1, 4, 10 and 16. Node 14 is not an ELCA since the two subtrees rooted at nodes 14 and 16 contain the same number of nodes for each keyword.

According to Lemma 1, 2 and 3, CA computation followed by appropriate pruning can be used to implement SLCA or ELCA computation. This gives rise to the basic ideas of our methods: (1) *for SLCA, find all CA node and check whether they are leaf nodes of the corresponding CA tree*, (2) *for ELCA, find all CA nodes and check their satisfiability according to Lemma 3.*

## IV. IDList and its Properties

As shown in Fig. 1, we assign each node an ID (underlined number) that is compatible with the document order

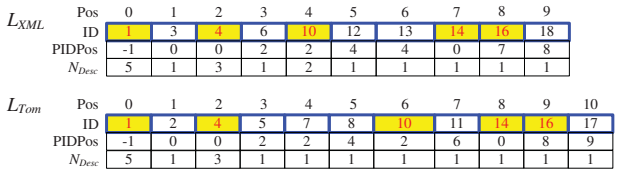


Fig. 3: Illustration of data organization for SLCA and ELCA computation.

to uniquely represent this node. For a given keyword query $Q = \{k_1, k_2, ..., k_m\}$, each keyword $k_i$ corresponds to a list $L_i$ of 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 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 $L_i$, "$N_{Desc}$" denotes the number of nodes that contain $k_i$ in the subtree rooted at $v$ (only required for *ELCA* computation). Entries in $L_i$ are sorted in ascending order according to their ID values. Hereafter, we call the inverted list used in our method *IDList*, which can be efficiently constructed when parsing the XML document in document order.

Fig. 3 shows the two inverted IDLists of "XML" and "Tom", respectively, where each entry is a column in the table. Take $L_{\text{XML}}$ 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 $L_{\text{XML}}$ 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 "XML".

For simplicity, we do not differentiate a node, its ID, and the corresponding entry of an IDList unless there is ambiguity. For example, when we say node 4, it denotes the node in Fig. 1 with ID 4, it also denotes the entry in $L_{\text{XML}}$ and $L_{\text{Tom}}$ 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 $L_i[x] = C_i$.

According to the data organization of Fig. 3, we have the following properties.

*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 $R(L_1, ..., L_m) = V_T$. 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.

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

## V. SLCA Computation

In this section, we consider a forward and two backward algorithms to compute SLCA results based on the optimized set intersection operation.

### A. Forward Solution

Our first SLCA query processing algorithm, i.e., FwdSLCA, computes all CA nodes in document order, filters them on-the-fly using Property 2 and Lemma 2, and outputs leaf nodes of the CA tree as the final SLCA answers.

*1) The Algorithm:* The pseudocode is shown in Algorithm 1. It finds a CA node $v$ by intersecting $m$ lists (line 3) in each iteration (lines 2-9). Since it processes nodes in the document order, it cannot determine if an SLCA candidate is really an SLCA result until it obtains the next SLCA candidate [4]. Hence, it buffers the last SLCA candidate $u$. In line 4, if no CA node exists in the remaining entries of all lists (when $flag$ = FALSE), we can break out of the loop (line 8), and output the buffered SLCA candidate $u$ as a result (line 10); otherwise, when $flag$ = TRUE, we check whether the CA node $u$ of the previous iteration 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, thus we directly output it as a qualified result in line 5. Then in line 6, we buffer the current CA node $v$ as a candidate SLCA node in $u$, then advance the cursors of all the lists (line 7) before the next iteration.

The function fwdGetCA is called to find a CA node by intersecting $m$ lists. It 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 list to the longest (lines 1 and 3-13). The probe operation will continue to the remaining lists if entries of same ID are found (line 9-11); otherwise, since we found an entry larger than the current eliminator, the eliminator will be reset and we restart the probing from $L_1$ immediately (line 12). Binary search (fwdBinSearch) is used to perform the probe – finding a matched node for a given eliminator, though other kinds of search can also be used. Function fwdEof checks whether we have exhausted a list by checking the cursor positions.

*Example 2:* For keyword query $Q = \{XML, Tom\}$ and its two IDLists in Fig. 3, FwdSLCA will find all CA nodes of $Q$, i.e., 1, 4, 10, 14 and 16, 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., qualified SLCA node, by testing their parent-child relationship. Consider node 10 and 14, node 14's Pos values is 7. Since $10 \neq L_{XML}[L_{XML}[7].PIDPos].ID = 1$, node 10 is a leaf node of $Q$'s CA tree, thus node 10 is an SLCA node. Note that node 16 is the last CA node visited in document order, it must be a leaf node of the CA tree, and therefore an SLCA node. □

*2) Analysis of the FwdSLCA Algorithm:* For a given keyword query $Q$ of $m$ keywords, the cost of line 7 in Algorithm 1 is $O(m)$. The cost of fwdBinSearch is $O(\log |L_m|)$. The maximum number of iteration is bounded by the size of the smallest IDList, i.e., $|L_1|$. Therefore, the time complexity of FwdSLCA is $O(m \cdot |L_1| \cdot \log(|L_m|))$.

As a comparison, the Stack algorithm [4] has a complexity of $O(d \cdot m \cdot \sum_i^m |L_i^D|)$, where $|L_i^D|$ is the size of the inverted list with Dewey labels, and $d$ is the depth of the XML data tree.

**Algorithm 1**: FwdSLCA($Q$)

```
/*Q = {k_1, ..., k_m}, |L_1| ≤ |L_2| ≤ ... ≤ |L_m|*/
1  u ← {−1, −1}; /*u.ID is the root node initially*/
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
2  k ← argmax_i{C_i.ID}
3  while (j < 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
11    else
12       k ← j; j ← 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]
```

Both IL [4] and IMS [7] algorithms have time complexity of $O(d \cdot m \cdot |L_1^D| \cdot \log(|L_m^D|))$. For JDewey [6], its time complexity 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, our algorithm has a comparable worst case complexity with existing algorithms. Our experimental results show that the performance of our proposed methods, together with optimization techniques to be introduced in the next sections, substantially outperforms that of existing algorithms.

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 [15], or interpolation search [16]) 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., [21], [22]).

### B. Backward Solution

Algorithm 1 computes all CA nodes in document order as with all existing algorithms except JDewey [6], therefore we have to compute potentially many CA nodes that are eventually not SLCA nodes because there are other CA nodes that are

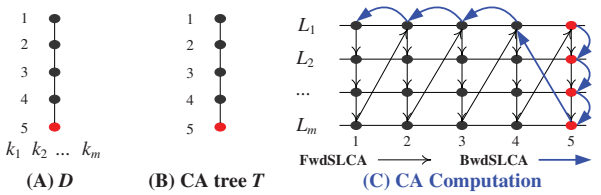


Fig. 4: Illustration of the problem of Algorithm 1.

identified afterwards and reside in the subtree of them. We give an extreme example below.

*Example 3:* Consider a keyword 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 lists, which is unnecessary but can hardly be avoided, as we cannot tell whether the current CA node is a qualified SLCA node or not unless we obtain the next CA node. □

Unlike JDewey [6] which solves this inherent inefficiency by computing SLCA results from bottom-up, we propose to compute SLCA results by the reverse document order. The basic idea of the resulting backward solution is: *whenever find a CA node $v$ from backward set intersection, we proactively* remove *all its ancestor nodes from the* shortest *inverted list; this will guarantee that every time a CA node is computed, it is 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* list. For each CA node returned from an iteration, we use the PIDPos to find all ancestors' Pos values in the shortest list, then mark them as invalid entries in the auxiliary array, such that all these nodes can be skipped when computing CA nodes. Another implementation choice is to use a hash table to "remember" all ancestor nodes of an SLCA node $v$.

Our backward solution can achieve the same effect that each outputted CA node is an SLCA node *without* using any additional data structure. The idea is to remove all *adjacent parent nodes* in the IDList, i.e., we iteratively skip the node $u$ preceding the current CA node $v$ in the IDList such that $u$ is the parent of $v$. This is correct due to 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** : $v \prec_d u'$. Therefore $v \prec_d u' \prec_d v'$. 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.

■

The pseudocode of our backward solution is shown in Algorithm 2. Note that we only list additional or different lines for the auxiliary functions if they are different from their counterparts in Algorithm 1. In each iteration (lines 2-10), it finds a CA node $v$ from $m$ lists. 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. Therefore, in lines 7-9, we perform such iterative skipping only on the smallest IDList $L_1$ until its current and previous entries have no parent-child relationship.

In Algorithm 2, function bwdGetCA is called in each iteration to find an ***SLCA*** node from the $m$ lists, which in turn calls bwdBinSearch to locate the matched node for $C_k$. Note that in lines 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 with Algorithm 2. The first iteration returns node 5, which is outputted as an SLCA node immediately. Then by iteratively pruning the adjacent parent node (first node 4, then 3, 2, and 1), cursor $C_1$ will be moved beyond the first entry in $L_1$, hence the algorithm stops without computing any other CA node. □

The worst case complexity of our backward algorithm is the same as that of the forward algorithm. However, since in the backward algorithm only SLCA nodes are computed and outputted as results via a simple modification to the set intersection algorithm, it is very efficient in practice.

### C. 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.

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 lists. 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 $v$,
- $y$ be the first node that precedes $v$ in $L_i$,

**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]
4.5       break
8  if (s > e) then C_j ← L_j[e]
9  return parent
```

- $C_k$ be the eliminator used in the next iteration 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 exists only if $z$ exists.

*Proof:* Since each probe operation involves nodes of two lists, we prove this result using nodes corresponding to keyword $k_1$ and $k_2$, respectively. After processing $v$, $C_2$ points to $y$ and $C_1$ points to $x$. Assume that $x \prec_d y$, according to the BwdSLCA algorithm, $C_1$, 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 cases with regard to the positional relationships between $x$ and $y$.

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

**(A.1)** $x = y = u$. In this case, $u$ and $v$ are two consecutive nodes in $L_2$, thus the 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)** $x \prec_a y$ (c.f., Fig. 5 (B)). In this case, the nodes between $u$ and $v$ in $L_2$ must be 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 $u \preceq_d w \preceq_d C_1 \prec_d z \preceq_d y \prec_d v$.

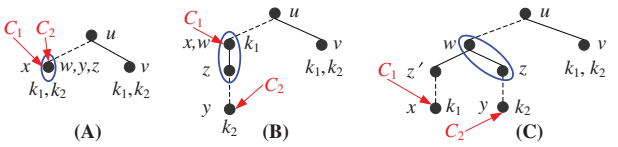


Fig. 5: Illustration of the positional relationships between nodes in an XML document, where each solid (dashed) line from $u$ to $v$ means $u \prec_p v (u \preceq_a v)$.

**(Case C)** $x \prec_d y$ and $x \not\prec_a y$ (c.f., Fig. 5 (C)). In this case, the nodes between $u$ and $v$ in $L_2$ 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 $m$ lists, if the minimum node after processing $v$ is $C_k (k \in [1, m])$, then we have the same result, that is, $u \preceq_d w \preceq_d C_k(\prec_d z) \preceq_d y \prec_d v$. ■

**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} ← setInterval(L_j, u)
```

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

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

According to Lemma 5, for each probe operation and its eliminator $C_k$, we set the search interval as $[pos(w), pos(z)]$, where $z = y$ if $C_k = y$. We implement the optimized backward algorithm, i.e., BwdSLCA$^+$, which is same as BwdSLCA except that in function bwdGetCA, bwdBinSearch is replaced by bwdBinSearch$^+$ (shown in Algorithm 3. The difference between bwdBinSearch and bwdBinSearch$^+$ lies in line 1 marked with rectangle in bwdBinSearch, which is replaced by calling function setInterval in bwdBinSearch$^+$. As shown in function setInterval, line 1 corresponds to case (A.2) of Lemma 5 and Fig. 5 (A). Lines 2-7 will iteratively compute the minimum interval, which correspond to cases (B) and (C) of Lemma 5. Note that case (A.1) of Lemma 5 is processed in lines 7-9 of Algorithm 2.

*Example 5:* For keyword query $Q$={$XML, Tom$} and its two IDLists in Fig. 3, the BwdSLCA$^+$ algorithm will firstly find the SLCA node 16, then in lines 7-9, node 14 is skipped, then $C_{XML}$ and $C_{Tom}$ point to 13 and 11, respectively. In the second iteration, $C_{Tom}$ is chosen as the eliminator, and its search interval is set as $[4, 6]$ of Pos values for $L_{XML}$. After outputting the second SLCA node 10, both cursors are moved forward to Pos 3 and 5, respectively. In the third iteration, $C_{XML}$ pointing node 6 at Pos=3 is chosen as the eliminator

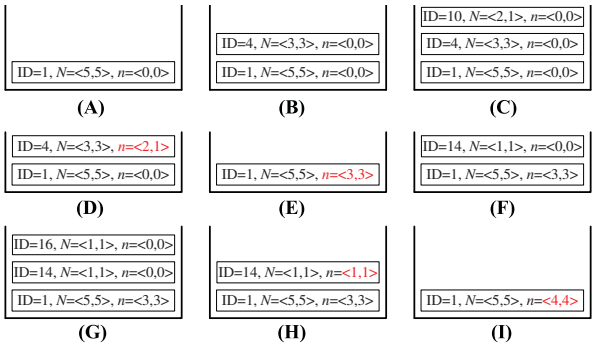


Fig. 6: Forward ELCA computation of Example 6.

to probe $L_{Tom}$, BwdSLCA$^+$ will call setInterval to set the search interval to [2,4] for $L_{Tom}$. The third iteration will skip nodes 4 and 1, and return FALSE to $flag$ to terminate the processing. □

## VI. ELCA Computation

### A. Forward Solution

Lemma 1 guarantees that $ELCA(Q) \subseteq CA(Q)$. Therefore we can still use the fwdCA function of Algorithm 1 to find all CA nodes first. Unlike SLCA, a non-leaf CA node may still be an ELCA node. Since the FwdELCA algorithm computes all CA nodes in document order, according to Lemma 3, it is not possible for us to know whether a CA node $v$ is an ELCA 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 $N_{Desc}$ values of all keywords in $Q$, the other is $n = \langle n_1, n_2, ..., n_m \rangle$ 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 the stack $S$ 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 we check whether $w$ is an ELCA according to Lemma 3.

*Example 6:* For keyword query $Q = \{XML, Tom\}$ and its two IDLists in Fig. 3, as shown in Fig. 6 (A)-(C), since node 1 is the parent of node 4, which in turn is the parent of node 10, the three CA nodes, i.e., node 1, 4 and 10, are pushed into stack $S$. The next CA node returned from fwdGetCA function is node 14, since node 10 and 4 are not the parent of node 14, they are popped 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. 6 (D) and (E); then we check the satisfiability of node 10 and node 4 according to Lemma 3, and output both of them as ELCA nodes; finally, node 14 and node 16 are pushed into $S$ (Fig. 6 (F) and (G)). After that, all elements are popped from $S$ and the algorithm stops. After each element is popped from $S$, we check its satisfiability and add its $N$ vector value to $n$ vector of the top element in $S$. Note that node 14 is not an ELCA node, since each $N_i$ of node 14 equals to its $n_i$ (see Fig. 6 (H)). □

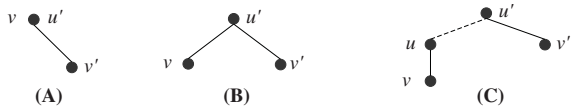


Fig. 7: The positional relationships for CA nodes, each solid (dashed) line from $u$ to $v$ means $u \prec_p v (u \prec_a v)$.

### B. 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, with its proof in [23].

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

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

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

There are three kinds of positional relationships between $v$, $u$ and $u'$, which correspond to Fig. 7 (A), (B) and (C). For the case of Fig. 7 (A), it means $v = u'$ and all descendant nodes of $v$ have been processed, thus $v$ should be popped from the stack and $v$'s parent, i.e., $u$, should be pushed into the stack if $u$ is not in the stack. For the case of Fig. 7 (B), it means that $u' \prec_p v$, thus we just need to output $v$ if it is an ELCA, then transfer its $N$ value to $u'$. For the case of Fig. 7 (C), it means $v$ is a leaf CA node and $u' \prec_a v$, thus we just need to output $v$ as an ELCA and transfer its $N$ value to its parent $u$, then push $u$ into the stack.

*Example 7:* For keyword query $Q = \{XML, Tom\}$ and its two IDLists in Fig. 3, the processing is shown in Fig. 8 (A)-(D). The first CA node is 16, which corresponds to Fig.

7 (C), thus 16 is output as an ELCA node, and its parent, i.e., node 14, is pushed into stack $S$ (Fig. 8 (A)). The next CA node is node 14, which corresponds to Fig. 7 (A), thus node 14 is popped out from $S$. Since node 14 is not an ELCA node, it will not be output as an ELCA node, then its parent, i.e., node 1, is pushed into $S$ (Fig. 8 (B)). The third CA node is 10, which corresponds to Fig. 7 (C), thus node 10 is a leaf node of the CA tree, and it is output as an ELCA node, then its parent, i.e., node 4, is pushed into $S$ (Fig. 8 (C)). The next CA node is 4, which corresponds to Fig. 7 (A), thus node 4 is popped from the stack and output as an ELCA node, then we add its $N$ value to node 1. The last CA node is 1, which also corresponds to Fig. 7 (A), then node 1 is output as an ELCA, and the processing stops. □

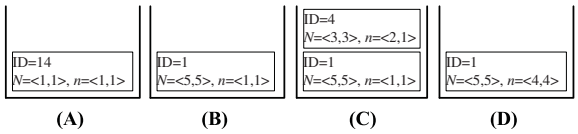


Fig. 8: Backward ELCA computation of Example 7.

## VII. Experiment

According to Lemma 1, for a given keyword query, both SLCA and ELCA nodes are subsets of CA nodes. Efficiently identifying all CA nodes will greatly impact the overall performance, based on which both SLCA and ELCA nodes can be identified easily. In this section, we present the experimental results to verify the benefits of our methods on CA, SLCA and ELCA computation.

### A. Experimental Setup

All experiments were run on a PC with Pentium4 2.8GHz CPU, 2GB memory, 500GB IDE hard disk, and Windows XP Professional OS.

We considered three groups of algorithms:

- **CA computation**. Algorithms of this group target at finding the common node from a set of IDLists, including SvS [17], Small Adaptive (SA) [17], Quantile-based (QB) [18], Probabilistic Intersection (PI) [20], FwdCA, BwdCA, and BwdCA$^+$. The last three are simplified from FwdSLCA, BwdSLCA, and BwdSLCA$^+$, respectively, where only function fwdGetCA, bwdGetCA and bwdGetCA$^+$ are called to return CA nodes[3]. All these algorithms were implemented to use binary search to make a fair comparison.
- **SLCA computation**. Algorithms of this group focus on SLCA computation, including Stack [4], IL [4], IMS [7], JDewey [6], and our three algorithms FwdSLCA, BwdSLCA and BwdSLCA$^+$.
- **ELCA computation**. Algorithms of this group focus on ELCA computation, including DIL [2], IS [12], HC [3], JDewey [6], and the our two algorithms FwdELCA and BwdELCA.

[3]We just use the parent node of current CA node to simplify the computation of non-leaf CA nodes, rather than blocking them from being output.

TABLE I: Comparison of index size.

| Dataset | Dewey | ID+PIDPos | ID+PIDPos+$N_{Desc}$ |
|---|---|---|---|
| XMark | 2.3GB | 817MB | 1.05GB |
| DBLP | 1.05GB | 673MB | 887MB |

TABLE III: Queries used in 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 | |

All algorithms were implemented using Microsoft VC++. The running time is the averaged over 100 runs with warm cache.

We used the XMark (582MB)[4] and DBLP (876MB)[5] datasets. Index sizes are listed in Table I. where each number is stored as an integer. Note that ID+Pos index is for SLCA computation and ID+Pos+$N_{Desc}$ is for ELCA computation.

We randomly selected 30 keywords for XMark dataset that falls into three categories according to their frequencies (i.e. $|L_{Dewey}|$ line in Table II): (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.

### B. Processed Nodes

Each number in line $|L_{Dewey}|$ of Table II denotes the number of Dewey labels of a keyword. The number of nodes they need to process is shown in the $N_{L_{Dewey}}$ 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 $|L_{ID}|$ 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).

### C. CA Computation

We generated two groups of queries (Table III) 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.

*1) Impacts of Different Probe Order:* As shown in Fig. 9 (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

[4]http://www.monetdb.org/Home

[5]http://www.informatik.uni-trier.de/∼ley/db/

TABLE II: Statistics of keywords used in our experiment.

| Keyword | tissue | baboon | necklace | arizona | cabbage | hooks | shocks | patients | cognition | villages |
|---|---|---|---|---|---|---|---|---|---|---|
| $\|L_{Dewey}\|$ | 384 | 725 | 200 | 451 | 366 | 461 | 596 | 382 | 495 | 829 |
| $N_{L_{Dewey}}$ | 3,252 | 6,013 | 1,704 | 2,255 | 3,016 | 3,869 | 4,823 | 3,306 | 4,192 | 6,981 |
| $\|L_{ID}\|$ | 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 |
| $\|L_{Dewey}\|$ | 18,441 | 17,129 | 16,797 | 23,561 | 36,304 | 35,257 | 19,902 | 33,520 | 31,232 | 34,065 |
| $N_{L_{Dewey}}$ | 113,081 | 100,338 | 116,429 | 143,894 | 213,449 | 187,843 | 113,081 | 173,900 | 129,455 | 177,450 |
| $\|L_{ID}\|$ | 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 |
| $\|L_{Dewey}\|$ | 299,018 | 304,969 | 352,121 | 368,544 | 535,268 | 313,398 | 457,232 | 350,560 | 411,575 | 304,752 |
| $N_{L_{Dewey}}$ | 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 |
| $\|L_{ID}\|$ | 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 |

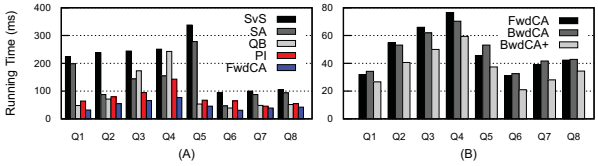


Fig. 9: Comparison of running time for CA computation.

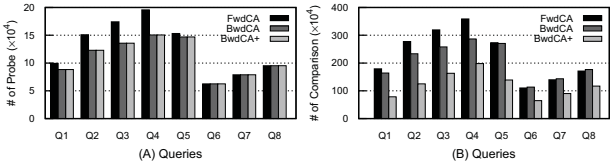


Fig. 11: Number of probe and comparison operations.

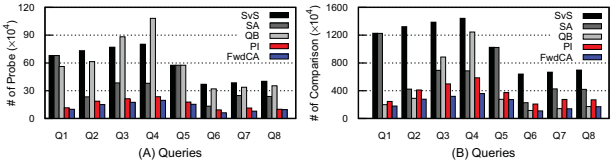


Fig. 10: Number of probe and comparison operations.

TABLE IV: The composition of the comparison operation for the BwdCA$^+$ algorithm.

| # | 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 | 76,138 | 88,036 | 50,706 | 32,710 | 49,055 | 65,420 |

cannot be well partitioned into sublists, such as Q3 and Q4, it degenerates to SvS. PI has slightly worse performance than QB on average. Our FwdCA beats all existing methods for all queries in running time.

This excellent performance can be explained according to Fig. 10, which shows the number of comparisons and probes. Fig. 10 (A) shows that the number of probe operations of our method is much less than those 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. 9 (A), QB performs similarly to or more efficient than PI in many cases, the reason can be further explained by referring to Fig. 10 (B). 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. 10 (B), our method achieves the best performance, the reason lies in the least number of comparison operations.

*2) Impacts of Reducing Search Interval:* Fig. 9 (B) presents the running time of FwdCA, BwdCA and BwdCA$^+$. We can find: (1) BwdCA does not always beat FwdCA, (2) BwdCA$^+$ outperforms FwdCA and BwdCA in all the cases.

According to Fig. 11 (A), BwdCA reduces many probe operations for Q1 to Q4, which can be verified according to Fig. 11 (B), 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. 11 (B), we find 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 IV, the number of comparison operations presented in Fig. 11 (B) 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 equals to the number of CA nodes. For FwdCA and BwdCA, the number of comparison is $N_1 + N_3$.

Form the above discussion, we conclude the following results: (1) always using the maximum cursor as the eliminator to probe the shortest list can improve the overall performance in most cases; (2) the optimization technique used by BwdSLCA$^+$ can improve the overall performance remarkably, as shown in Fig. 9 (B) and Fig. 11.

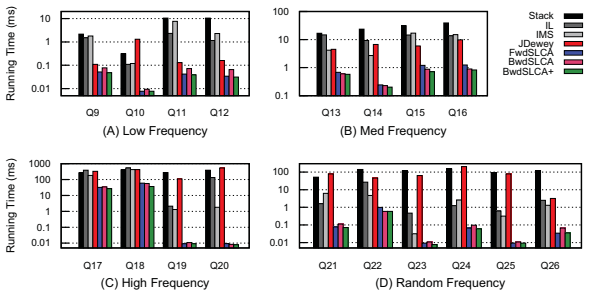


Fig. 12: Comparison of running time for SLCA computation.

TABLE V: Queries used in our experiment, $R_S(R_E)$ 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 | 34136 | 34189 | High |
| Q18 | date,listitem,emph | 43777 | 43792 | |
| 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 | |

### D. SLCA Computation

Based on the keywords in Table II, we generated four groups of queries as shown in Table V: (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.

*1) Evaluation Metrics:* The metrics for evaluating algorithms for SLCA and ELCA computation include: (1) running time, and (2) number of comparison operations, which is computed based on $N_1$, $N_2$ and $N_3$ in Table IV. For FwdSLCA and BwdSLCA, $N_C = N_1 + N_3$, and for BwdSLCA$^+$, $N_C = N_1 + N_2 + N_3$.

*2) Performance Comparison and Analysis:* Fig. 12 shows the running time of different algorithms on queries Q9 to Q26, from which we have the following observations: (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 can achieve better performance when there exists huge difference in the lengths of the set of Dewey label lists, e.g., Q21 to Q26 in Fig. 12 (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 may not be as efficient as that of Stack, e.g., Q17 and Q18, which can be further verified according to the number of comparison operations they have done. 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.

From Fig. 13 we have several observations: (1) our methods always need the least comparison operations, therefore achieves 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. 13, 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 IV, 15% to 30% comparison operations exists in the setInterval function, which can be done much more efficient than the comparison operations in 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.

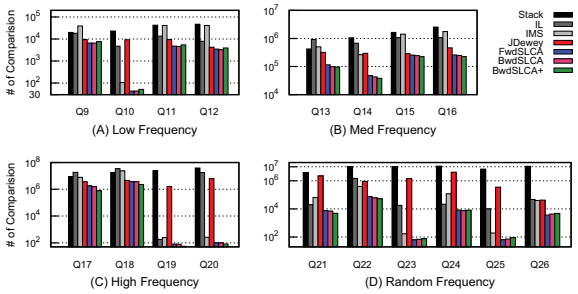


Fig. 13: Number of comparison operations.

Besides the 18 queries in Table V, we generated 174,406 queries by combining all 2, 3, 4 and 5 keywords from the 30 keywords listed in Table II.

Fig. 14 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. We can see that (1) the speedup of the IL, IMS, JDewey and our BwdSLCA$^+$ algorithms[6] over Stack is more significant when the result selectivity is low, (2)

[6]Since FwdSLCA, BwdSLCA and BwdSLCA$^+$ have similar performance, we only present the results for the BwdSLCA$^+$ algorithm. In Fig. 14, we take the result selectivity of existing methods as that of our method to make a fair comparison.

our method outperforms all existing algorithms in all cases. Note that in Fig. 14 (A) and (B), with the increase of the result selectivity, the average time used by all methods for result selectivity in [40,100] is less than that in [30,40), which can be explained by Fig. 15, 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].

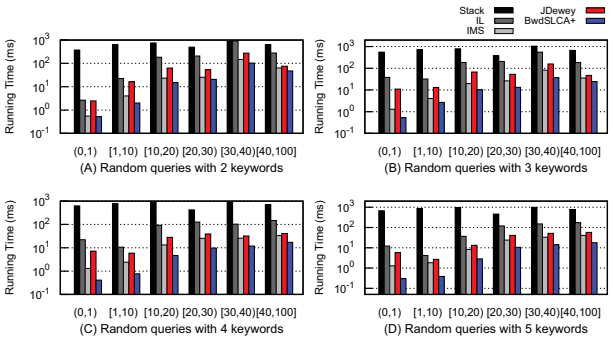


Fig. 14: Comparison of running time with different result selectivities (log-scaled).

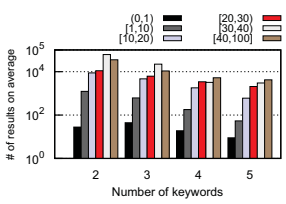


Fig. 15: Average number of results for each selectivity.

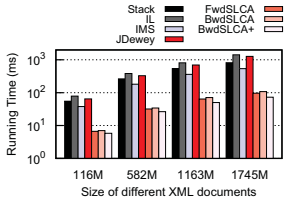


Fig. 16: Running time of Q17 on different XML documents.

Experimental results of 10 sample queries on DBLP is shown in Fig. 17, from which we know that our methods are more efficient than existing methods on these queries. Note that compared with IL and IMS, the speedup of our methods is not as significant as that of Fig. 12. This is because the DBLP document is very shallow, and the cost of testing the document order and computing the LCA for two nodes on DBLP dataset is not as expensive as that on XMark dataset.

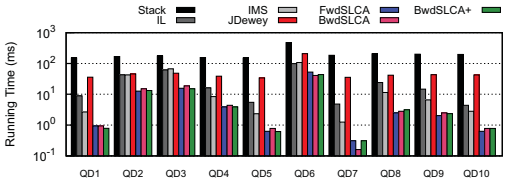


Fig. 17: Comparison of SLCA computation on DBLP dataset.

From the above discussion, we conclude that our methods outperform all existing methods, especially for queries with low result selectivity (see Table V) and Fig. 14. The reasons lie in that (1) the number of processed nodes (see Section VII-B) is largely reduced; (2) any kind of positional relationships between nodes can be determined with $O(1)$ cost; and (3) compared with JDewey, we put all node IDs in one list, while JDewey needs to do a set intersection operation for all lists of each tree depth, and it needs to afford more additional time on deleting ancestor nodes from all lists of each tree depth.

*3) Scalability:* We investigate the scalability from two aspects based on Fig. 14: (1) fixing the number of keywords and varying the result selectivity, which is just explained in the previous paragraphs, (2) fixing the result selectivity and varying the number of keywords, which can be obtained from the four sub-figures of Fig. 14 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 and BwdSLCA$^+$ 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. 16 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.

### *E. ELCA computation*

Fig. 18 shows the running time of different algorithms on queries Q9 to Q26 for ELCA computation.

As shown in Fig. 18, since DIL processes Dewey labels in document order, its performance is mainly affected by the number of processed Dewey labels. IS can utilize the positional relationships to skip many useless Dewey labels, thus it achieves better performance when there exists huge difference in the length of the set of Dewey label lists, e.g., Q21 to Q26 in Fig. 18 (D); however, for queries with similar lengths, it may not be as efficient as DIL, e.g., Q15, Q17, Q19 and Q20. HC uses a hash table to record for each node $v$, the number of nodes that directly contain each keyword in the subtree rooted at $v$, thus can beat other existing methods in most cases except when the shortest list is very long, such as Q17, Q18 and Q20. In contrast, our methods always need the least comparison operations, thus achieves the best performance for all queries. From Fig. 18, we also know that BwdELCA is more efficient than FwdELCA, because BwdELCA uses the optimization techniques introduced in Section V to reduce the search interval for each probe operation.

## VIII. Related Work

Many query semantics [1], [4], [5], [10] have been proposed to define query results for an XML keyword query. We refer readers to the recent tutorials for a complete coverage [24], [25]. Two widely adopted definitions are SLCA [4], [7] and ELCA [2], [3], [12]. To facilitate computing the SLCA and ELCA results on XML data, existing methods [2]–[4], [7]–[9] assign to each node $v$ a Dewey label [13] for computing the final matched results. The basic operation is computing the LCA node for a set of nodes and the efficiency is largely influenced by testing the positional relationships between two

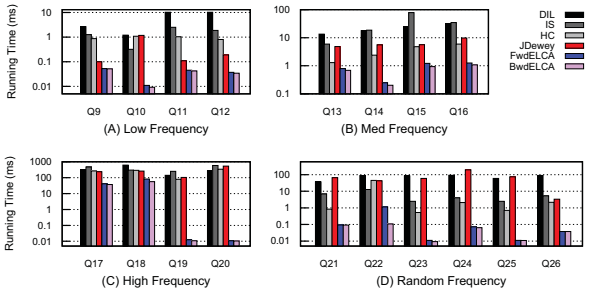


Fig. 18: Comparison of running time for ELCA computation.

Dewey labels. Unfortunately, for an XML tree of depth $d$, the cost of testing any of their positional relationships is $O(d)$ and difficult to make further improvement. Existing methods [2]–[4], [7] mainly focus on using $B^+$ tree to reduce the number of test operations by skipping some useless Dewey labels. For each test itself, the cost is still unchanged, i.e., $O(d)$. [6] studied how to support efficient top-$K$ XML keyword query processing based on the JDewey labeling scheme, where each component of a JDewey label is a unique identifier among all the nodes at the same depth. According to this property, the algorithms proposed in [6] perform set intersection operation on all lists of each tree depth from the leaf to the root.

For the set intersection problem, a simple yet efficient method is SvS [17], which intersects two sets at a time in increasing order of set sizes, starting with the two smallest sets. The Adaptive algorithm [19] computes the intersection by repeatedly cycling through the sets in a round-robin fashion. However, it does not always exhibit the best performance due to the overhead of adaptivity [17]. To address this problem, the Small Adaptive algorithm [17] was proposed by adopting galloping search [15], limiting the form of adaptivity, and changing the join order according the number of unprocessed components in each list. The Probabilistic Intersection algorithm [20] uses a probabilistic probing policy to determine which list to examine next based on historical data called "average jump". The Quantile-based algorithm [18] uses interpolation search to accelerate the probe operation, targeting at processing situations in which the distribution of components is highly non-uniform. [21] uses partitioning and hashing to quickly eliminate parts of the sets that do not have overlap and hence is suitable for cases where the relative results size is small.

## IX. 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 propose to use IDLists, rather than inverted lists of Dewey labels, for SLCA and ELCA computation. We show that both SLCA and ELCA computations can be cast into a set intersection problem among IDLists of query keywords. Several optimization methods based on the query semantics and structural relationship between nodes have been devised to further speed up the query processing. Experimental results demonstrate superior performance of our proposed methods over existing ones.

## Acknowledgment

This research was partially supported by grants from the Natural Science Foundation of China (No. 61073060, 61040023, 61103139) and the Fundamental Research Funds of Hebei Province (No. 10963527D). 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] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, "Xsearch: A semantic search engine for xml," in *VLDB*, 2003.
[2] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "Xrank: Ranked keyword search over xml documents," in *SIGMOD Conference*, 2003.
[3] R. Zhou, C. Liu, and J. Li, "Fast elca computation for keyword queries on xml data," in *EDBT*, 2010.
[4] Y. Xu and Y. Papakonstantinou, "Efficient keyword search for smallest lcas in xml databases," in *SIGMOD Conference*, 2005.
[5] Y. Li, C. Yu, and H. V. Jagadish, "Schema-free xquery," in *VLDB*, 2004.
[6] L. J. Chen and Y. Papakonstantinou, "Supporting top-k keyword search in xml databases," in *ICDE*, 2010.
[7] C. Sun, C. Y. Chan, and A. K. Goenka, "Multiway slca-based keyword search in xml data," in *WWW*, 2007.
[8] Z. Liu and Y. Chen, "Identifying meaningful return information for xml keyword search," in *SIGMOD Conference*, 2007.
[9] Z. Bao, T. W. Ling, B. Chen, and J. Lu, "Effective xml keyword search with relevance oriented ranking," in *ICDE*, 2009.
[10] G. Li, J. Feng, J. Wang, and L. Zhou, "Effective keyword search for valuable lcas over xml documents," in *CIKM*, 2007.
[11] G. Li, S. Ji, C. Li, and J. Feng, "Efficient type-ahead search on relational data: a tastier approach," in *SIGMOD Conference*, 2009.
[12] Y. Xu and Y. Papakonstantinou, "Efficient lca based keyword search in xml data," in *EDBT*, 2008.
[13] 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.
[14] C. Li, T. W. Ling, and M. Hu, "Efficient updates in dynamic xml data: from binary string to quaternary string," *VLDB J.*, vol. 17, no. 3, 2008.
[15] J. L. Bentley and A. C.-C. Yao, "An almost optimal algorithm for unbounded searching," *Inf. Process. Lett.*, vol. 5, no. 3, 1976.
[16] J. Barbay, A. Lpez-Ortiz, and T. Lu, "Faster adaptive set intersections for text searching," in *WEA*, 2006, pp. 146–157.
[17] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Experiments on adaptive set intersections for text retrieval systems," in *ALENEX*, 2001.
[18] D. Tsirogiannis, S. Guha, and N. Koudas, "Improving the performance of list intersection," *PVLDB*, vol. 2, no. 1, 2009.
[19] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Adaptive set intersections, unions, and differences," in *SODA*, 2000.
[20] 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.
[21] B. Ding and A. C. König, "Fast set intersection in memory," *PVLDB*, vol. 4, no. 4, pp. 255–266, 2011.
[22] H. Yan, S. Ding, and T. Suel, "Inverted index compression and query processing with optimized document ordering," in *WWW*, 2009, pp. 401–410.
[23] J. Zhou, Z. Bao, W. Wang, and T. W. Ling, "Efficient processing of keyword queries on xml data based on set intersection operation," *Technical report TRB7/11, School of Computing, National University of Singapore, July 2011*.
[24] Y. Chen, W. Wang, Z. Liu, and X. Lin, "Keyword search on structured and semi-structured data," in *SIGMOD Conference*, 2009, pp. 1005–1010.
[25] Y. Chen, W. Wang, and Z. Liu, "Keyword-based search and exploration on databases," in *ICDE*, 2011.