THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TRB7/11

Fast SLCA and ELCA Computation for XML Keyword Query Based on Set Intersection Operation

Junfeng Zhou, Zhifeng Bao, Wei Wang, Tok Wang Ling, Ziyang Chen, Xudong Lin and Jingfeng Guo

July 2011

T e c h n i c a l R e p o r t

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

Fast SLCA and ELCA Computation for XML Keyword Query Based on Set Intersection Operation

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 on XML data based on SLCA and ELCA semantics. We have an elaborate design of the keyword inverted list to efficiently locate the nodes that directly or indirectly contain a given keyword. We propose a family of algorithms that are based on set intersection operation to accelerate SLCA and ELCA computation. In essence, the problem of SLCA and ELCA computation becomes finding a set of common nodes that appear in all inverted lists and checking their satisfiability. We show that any existing set intersection algorithms can be adopted for SLCA and ELCA computation, and our optimization techniques can be used together with any existing search method to improve the overall performance. Experiments verify that the performance of our methods outperforms existing methods by more than 2 orders of magnitude in most cases.

I. Introduction

Keyword search on XML data has been a hot research issue along with the ever increasing of XML-based applications [1]–[10]. Similar to the importance of effectiveness to keyword search, efficiency is also a key factor to keyword search, especially for applications that need to provide instant feedback to many users in Web service-based environment [11].

Over the past few years, researchers have proposed several semantics [1]–[5], [10] to define meaningful results for keyword queries, among which the most widely adopted ones are SLCA [4] and ELCA [2], [3]. As both semantics are defined based on the notion of lowest common ancestor (LCA), and the Dewey label [12] of a node n consists of a sequence of components that implicitly contain all ancestor nodes on the path from the document root to n, Dewey labeling scheme is a natural choice of the state-of-the-art algorithms [2]–[4], [7], [8], [13] for result computation. The two basic operations for existing methods are $(OP1)$ testing the document order of two nodes and $(OP2)$ computing the LCA node of two nodes.

However, as each Dewey label consists of a set of components that collectively represent a node, the cost of both $OP1$ and $OP2$ operations is linear to the height of the XML tree. In practice, as a node v could be a common ancestor of many nodes, all nodes on the path from the root to v may be repeatedly visited, which we call as common-ancestor-repetition (CAR). Obviously, query processing based on Dewey labels will result in CAR problem, which is inefficient yet difficult to make optimization.

Example 1: To verify our intuition about the CAR problem, we have implemented the IL algorithm [4] for SLCA computation, which is the one that is completely based on $OP1$ and $OP2$ operations, and was verified by existing methods [4], [7] an efficient one in many cases. For query $\{bold, increase\}$ on XMark[1] dataset of 582MB, the lengths of the two inverted Dewey label lists for "bold" and "increase" are 368544 and 304752, which contain 52013 common ancestor nodes. After processing this query, we know that $OP1$ and $OP2$ were called 6169093 times, which results in 14132239 comparison operations between two numbers. As a result, each common ancestor node is visited 272 times on average, and the time spent on $OP1$ and $OP2$ operations occupies more than 80% of the total time. □

To address the CAR problem, we propose a suite of novel and efficient algorithms for answering SLCA and ELCA queries that depart from existing approaches based on Dewey encoding. We propose to assign each node a unique ID which is its pre-order number. 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. It can be shown that SLCA and ELCA computation can be cast into a variant of set intersection problem. We also design several optimization techniques that exploit the positional relationship between nodes in XML tree to accelerate the computation. In our extensively experimental study, we not only find that our methods outperform existing approaches by 2 orders of magnitude in most cases, but also have an in-depth analysis on the major factors that drags the performance of each previous work. To the best of our knowledge, this is the first work that has comprehensive study and in-depth analysis on the behavior of all existing SLCA and ELCA algorithms implemented on various choices of labeling schemes.

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

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

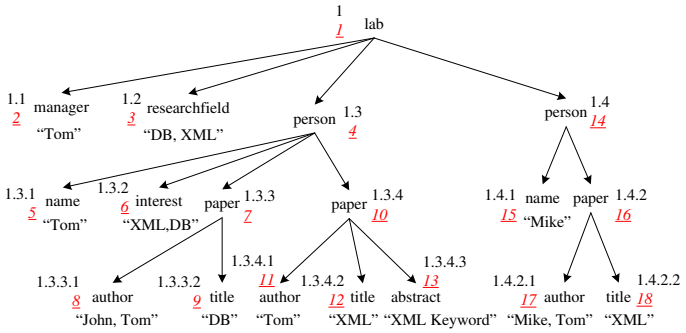


Fig. 1: A sample XML document.

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 (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 a Dewey label [12] to represent v. In Fig. 1, the Dewey label of each node is shown as the black sequence of components. In our method, we assign each node an ID (the underlined number beside it in Fig. 1) that is compatible with the document order to globally represent this node. For frequently updated XML data, we can use the encoding scheme of [14] to assign an 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 ID 1, 4, 10 and 16.

The most widely adopted variants of LCA are SLCA [4], [7] and ELCA [2], [3], [13]. 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 $SLCA(Q) = \{v | v \in LCA(L_1, L_2, ..., L_m)$ and $\nexists v' \in LCA(L_1, L_2, ..., L_m)$, such that $v \prec_a v'\}$. In Fig. 1, although node 1 and node 4 are LCAs, they are ancestors of node 10, thus the set of SLCAs for $Q = \{XML, Tom\}$ are node 10 and node 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 ID 1, 4, 10 and 16.

C. Set Intersection Methods

Finding all common components from a set of lists is a central operation in query processing on text and database. Given m lists $L_1, L_2, ..., L_m$, to efficiently find the common components that appear in all these lists, we need to repeatedly pick a component e from a list L_i, and use it as the *eliminator* to *probe* the remaining lists. If e appears in all m lists, output it as a result. Obviously, the overall performance is dominated by two orthogonal factors: (1) the number of probe operation, which is in turn dominated by *probe order*, i.e., which list should be firstly probed; and (2) the cost of each *probe operation*, i.e., how to efficiently find the *matched component*[2] for eliminator e, which is further affected by two orthogonal factors: (2.1) *search method*, such as binary, galloping [15] or interpolation search, and (2.2) *search interval*.

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

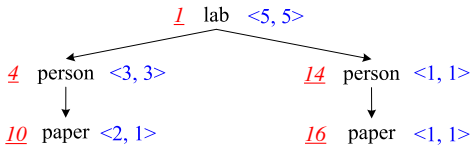


Fig. 2: A sample CA tree of $Q = \{XML, Tom\}$ on the XML document of Fig. 1, each italic number beside a node v denotes v's ID, and $< N_1, N_2 >$ is the vector associated with v denoting the number of nodes that directly contain "XML" and "Tom" in the subtree rooted at v.

For *probe order*, assume that $|L_1| \leq |L_2| \leq ... \leq |L_m|$, the SvS algorithm [16] computes the final results by processing two lists each time from the shortest to the longest, i.e., $R(L_1, L_2, ..., L_m) = R(R(L_1, ..., L_{m-1}), L_m)$, it is a blocking algorithm and needs to buffer many intermediate results when the result selectivity is low. The Quantile-based algorithm (QB) [17] will encounter the same problem when the lists exhibit *locally skewed distribution* but *globally uniform distribution*. In this case, the initial lists cannot be partitioned into sub-lists of large discrepancy in length, and QB degenerates accordingly. On the contrary, the Adaptive (Adaptive) [18], the Small Adaptive (SA) [16] and the Probabilistic Intersection (PI) [19] algorithms can dynamically adjust the probe order and output the results without buffering any intermediate results; however, the sort operation according to the special heuristic used by these methods for choosing the probe order could be a great burden in practice.

For *search interval*, assume that n components have been processed for the probed list L_i, then no matter which search method is used, the length of the search interval is $|L_i| - n$. The shorter the search interval is, the less the comparison needed for a probe operation are.

III. Overview of Our Method

For our method, the satisfiability of an SLCA or ELCA node is constructed based on an important concept, i.e., CA tree, which in turn is based on the following definition.

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 those with node ID 1, 4, 10, 14, 16. Obviously, we have the following lemma.

Lemma 1: For a given keyword query Q=$\{k_1, k_2, ..., k_m\}$ and an XML document D, $CA(Q) \supseteq ELCA(Q) \supseteq SLCA(Q)$.

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.

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 $v \in V_T^{leaf}$ do 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 ID 10 and 16.

According to the definition of ELCA [2], [3], [13], we have the following lemma, which is consistent with [3].

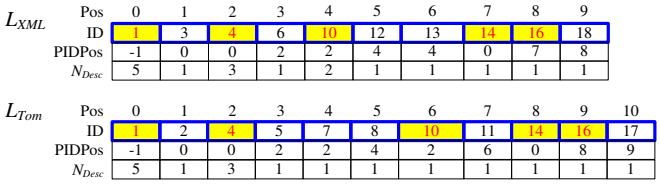
Lemma 3: For a given keyword query $Q = \{k_1, k_2, ..., k_m\}$ and its CA tree T on an XML document D, assume that for each node $v \in V_T$, $S_{child}^v = \{v_1, v_2, ..., v_x\}$ is the set of child nodes of v in T, v is associated with a vector $N =< N_1, N_2, ..., N_m >$, where $N_i (i \in [1, m])$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

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, it's easy to figure out that the qualified ELCA nodes are those with ID 1, 4, 10 and 16. Node 14 is not an ELCA since the two subtrees rooted at node 14 and 16 contain the same number of nodes for each keyword.

According to Lemma 1, 2 and 3, CA computation is the basis for SLCA and ELCA computation, and the basic idea of our method is: (1) *for SLCA, in each iteration, find a CA node and check whether it is a leaf node of the corresponding CA tree*, (2) *for ELCA, in each iteration, find a CA node and check its satisfiability according to Lemma 3*.

IV. IDList and its Properties

As shown in Fig. 1, we assign each node an ID that is compatible with the document order to globally represent this node, which is denoted as the underlined number beside each 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, which correspond to all nodes that directly contain k_i and their ancestors. Each entry



L_{XML}	Pos	0	1	2	3	4	5	6	7	8	9
	ID	1	3	4	6	10	12	13	14	16	18
	PIDPos	-1	0	0	2	2	4	4	0	7	8
	N_{Desc}	5	1	3	1	2	1	1	1	1	1

L_{Tom}	Pos	0	1	2	3	4	5	6	7	8	9	10
	ID	1	2	4	5	7	8	10	11	14	16	17
	PIDPos	-1	0	0	2	2	4	2	6	0	8	9
	N_{Desc}	5	1	3	1	1	1	1	1	1	1	1

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

corresponding to v in L_i consists of three member variables: "ID" is the node ID for 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 directly contain k_i in the subtree rooted at v, which is just used for *ELCA* computation. All entries of 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 constructed efficiently when parsing the XML document in document order.

Fig. 3 shows the two inverted IDLists of "XML" and "Tom", respectively. Take L_{XML} as an example, "Pos" denotes 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_{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". The three values at Pos 3 are 6, 2 and 1, which means that the parent of node 6 is the one at Pos 2, i.e., its parent ID is 4, and the subtree rooted at node 6 contains just one node that directly contains "XML".

In the following discussion of our methods, we do not differentiate a node, its ID and the corresponding entry of an IDList if without 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_{XML} and L_{Tom} at Pos 2. For the m IDLists of a given keyword query, we always assume that $|L_1| \leq |L_2| \leq \ldots \leq |L_m|$, 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, which can be moved to the entry (if any) next to C_i by calling $fwdAdvance(C_i)$. We use $pos(C_i)$ to denote the "Pos" value of C_i in L_i, $L_i[x]$ denotes an entry of L_i at Pos x, 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 $R(L_1, L_2, ..., L_m) = V_T$. If the set intersection operation is done in ascending order, then results of R are output in document order, equaling to visit T in document order.

The correctness of Property 1 is obvious, since each IDList L_i contains nodes that directly contain k_i and all their ancestor nodes, the result of a set intersection operation on IDLists of a keyword query must be all CA nodes of the CA tree.

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

A. Forward Solution

For a given keyword query Q, our first algorithm, i.e., FwdSLCA, computes all CA nodes in document order, and at the same time, it finds leaf nodes of the CA tree using Property 2, and outputs them as SLCA nodes according to Lemma 2 if they are leaf nodes. The basic idea of the forward solution for SLCA computation is: *for a given keyword query Q, in each iteration, find a common node v from the set of IDLists (i.e., v is CA node) in document order, then output the CA node u of the previous iteration as an SLCA node if u is a leaf node of Q's CA tree T.*

1) The Algorithm: As shown in Algorithm 1, our method tries to find a CA node v in line 3 from the m lists in each iteration (line 2-9). If no CA node exists in the remaining entries of all lists (line 4, $flag =$ FALSE), we directly break out of the loop, and output the last CA node u as an SLCA node (line 10); otherwise, it means that there are still CA nodes in all lists (line 4, $flag =$ TRUE), then we check whether the CA node u of the previous iteration is the parent of v (line 5). Note that in Algorithm 1, *parent* denotes the parent node of v. Since all CA nodes are identified in document order, if $u \not\prec_p v$(line 5, $parent.ID \neq u.ID$), 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. Otherwise, in line 6, we set u as the current CA node v, then move all cursors to the next entries in all lists (line 7) and start a new iteration.

Among the FwdSLCA algorithm, function fwdGetCA is called in each iteration to find a CA node from the m lists. It always uses the cursor with the maximum ID value as the eliminator to probe the shortest list first, because *if all IDs are uniformly distributed, the difference between IDs of two consecutive nodes of a shorter list is larger than that of a longer list*, and in this way, we can avoid many unnecessary probe operations and skip more useless entries. As shown in line 1 and 12 of function fwdGetCA, our method always gives priority to the shortest list. The probe operation will spread to the remaining lists if entries of same ID are found (line 9-11). Further, our method will immediately return to probe the shortest list (line 12) if the returned ID is larger than the eliminator. For each probe operation, our method simply uses binary search

Algorithm 1: FwdSLCA(Q)/*$Q = \{k_1, ..., k_m\}$*/

```
/*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  n ← 1; j ← 1;
2  k ← maxarg_i{C_i.ID}
3  while (n < m) do
4     if (j = k) then
5        if (j = m) then j ← 1 else 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       n ← n + 1
11       if (j = m) then j ← 1 else j ← j + 1
12    else {n ← 1; 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     m ← (s+e)/2
4     if (L_j[m].ID = u.ID) then {C_j ← L_j[m]; break}
5     else if (L_j[m].ID < u.ID) then s ← m + 1
6     else e ← m − 1
7  endwhile
8  if (s > e) then C_j ← L_j[s]
```

to complete this task. Function fwdEof is used to check whether there exists one IDList, for which all nodes are processed. Function fwdBinSearch is used to find a matched node for a given eliminator.

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 $10 \neq L_{Tom}[L_{Tom}[7].PIDPos].ID = 1$, 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 both line 7 in Algorithm 1 and line 2 in fwdGetCA function is $O(m)$. Since the cost of each call of fwdBinSearch is $O(\log |L_m|)$, in the worst case, i.e., all lists have the same length and all nodes are CA nodes, the cost of line 2 in fwdGetCA can be omitted compared with the cost of calling fwdBinSearch m times, therefore the time complexity of FwdSLCA in the worst case is $O(m|L_1| \log |L_m|)$.

As a comparison, in each iteration, the Stack algorithm [4] needs to find a minimum Dewey label from m inverted Dewey label lists, the cost of such operation is $O(dm)$, where d is the depth of the given XML document. Since there are $\sum_i^m |L_i^D|$ Dewey labels, the overall time complexity of the Stack algorithm is $O(dm \sum_i^m |L_i^D|)$, where L_i^D is a Dewey label list. For IL [4] and IMS [7], the time complexity is $O(dm|L_1^D| \log |L_m^D|)$. For JDewey [6], it needs to do m set intersection operations on each set of lists corresponding to each tree depth, therefore the overall time complexity is also $O(dm|L_1^D| \log |L_m^D|)$.

Obviously, only if $|L_i^D| = 1$, we have $|L_i| = d|L_i^D|$; otherwise, $|L_i| < d|L_i^D|$, because each IDList contains each CA node only once, while each Dewey label list may implicitly contain some CA nodes many times. Although the complexity of our method equals to that of existing set intersection methods, our experimental results show that in practice, the probe order

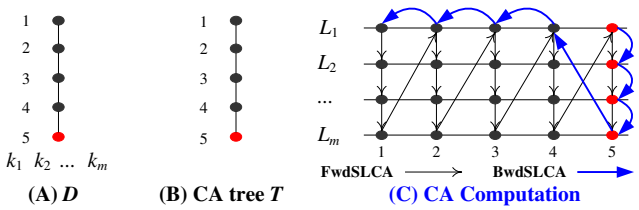


Fig. 4: Illustration of the problem of Algorithm 1.

adopted in our method and the optimization technique introduced in the following section can greatly improve the overall performance. Besides, our method does not block intermediate results.

Theorem 1: **(Generalization)** Any existing set intersection algorithm can be used in Algorithm 1 for CA computation, and in each probe operation, either binary, galloping [15] or interpolation search can be used in function fwdGetCA to find the matched node.

B. Backward Solution

Algorithm 1 computes all CA nodes in document order, therefore the computation of many CA nodes that are not SLCA nodes cannot be simplified, because *the satisfiability of the current CA node is determined by the CA node returned in the next iteration*, which makes it inflexible to utilize the semantic constraints to make further optimization.

Example 3: Consider a keyword query Q=$\{k_1, k_2, ..., k_m\}$ and 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 cannot be avoided, since before meeting the next CA node, we cannot know whether the current CA node is a qualified SLCA node. □

The basic idea of the backward solution is: *whenever finding a CA node v, we pro-actively **remove** the occurrence of the ancestor nodes of v in the **shortest** inverted list, such that the computation of these ancestor nodes can be largely simplified and they won't be in the intersection result any more.*

When implementing the *remove* operation, several methods can be adopted. One is using an auxiliary array with size equaling to that of the *shortest* list, then 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 method is using a hash table to record all ancestor nodes of v. However, both are not space efficient, as they need to use additional space to record all CA nodes that are not SLCA nodes.

Considering this problem, the backward solution introduced in this section computes CA nodes in *reverse* document order, i.e., it scans the entries in the set of lists in descending order, which equals to visiting nodes of the CA tree in reverse document order, such that the satisfiability of the current CA node is determined by the CA node returned from the previous iteration, rather than the one returned from the next iteration.

Lemma 4: Assume that all CA nodes are computed in reverse document order, let v' be the CA node returned from the previous iteration, u' the parent node of v', v the CA node of the current iteration, then v is an SLCA node if $v \neq u'$.

Proof: If $v \prec_d u'$, we have $v \prec_d u' \prec_d v'$, thus v cannot be the current CA node, because the previous CA node is v', and it is impossible to meet v before u' in reverse document order. If $u' \prec_d v \prec_d v'$ and 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 $u' \prec_d v \prec_d v_d \prec_d v'$, thus the CA node of the current iteration should be v_d, not v, which contradicts the assumption. ■

Therefore, when we meet a CA node in any list, we can immediately know whether it is a qualified SLCA node. In this way, many probe operations on other lists can be avoided.

As shown in Algorithm 2, in each iteration (line 2-10), our method tries to find a CA node v in line 3 from the m lists. If no CA node is found (line 4, $flag$ = FALSE), we directly skip out of the loop (line 5); otherwise, we directly output the current CA node as an SLCA node. In line 6, all cursors are moved to the previous entries in all lists. For any list and its two consecutive entries u and v, assume that $u.ID < v.ID$ and u is the parent of v, then u must not be an SLCA node, and can be directly skipped without processing. In line 7-9, we repeatedly move C_1 backwardly until the two consecutive entries have not 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 line 4.2 and 4.3 of bwdBinSearch, the current CA node will be skipped without processing if its ID equals to that of the parent of the previous CA node.

Example 4: Continue Example 3. By Algorithm 2, the first iteration returns node 5, which is output as an SLCA node since it is the first returned CA node in reverse document order, so it must be an SLCA node. Because node 1 is the parent of node

2, which is in turn the parent of node 3, and node 3 is the parent of node 4, in line 6-9, cursor C_1 will be moved to the position just before the first entry in L_1, then the processing will stop. □

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($parent$) /*Based on fwdGetCA*/

```
2   k ← minarg_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($L_j, u, parent$) /*Based on fwdBinSearch*/

```
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
```

Function bwdBinSearch$^+$($L_j, u, parent$) /*Based on bwdBinSearch*/

```
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}
```

Property 3: Each CA node returned by the bwdGetCA function is an SLCA node.

Proof: As discussed in previous paragraphs, any CA node that is the parent of other CA nodes will be skipped without processing, that is, function bwdGetCA only returns leaf nodes of a CA tree. According to Lemma 2, each CA node returned by function bwdGetCA is an SLCA node. ■

C. Optimized Backward Solution

Probe operation is one of the most important factors that will greatly affect the overall performance for set intersection problem. In this section, we introduce an optimization technique that can utilize the structure relationship between nodes to reduce the number of search steps for probe operation.

Assume that all CA nodes are computed in reverse document order, v is the CA node of the current iteration, u 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, thus we have $u \preceq_d C_k \prec_d v$, and the search interval for each probe operation can be reduced from $[0, pos(C_j)]$ to $[pos(u), pos(v)]$ in line 1 of bwdBinSearch. In fact, we can further reduce the search interval to make the probe operation more efficient.

Lemma 5: Assume that all CA nodes are computed in reverse document order, let v be the CA node returned from the current iteration, u the parent node of v, y the first node that precedes v in L_i, C_k the eliminator used in the next iteration satisfying $u \preceq_d C_k \preceq_d y \prec_d v$, w the common ancestor of C_k and y, z the child node of w (if exists) on the path from w to y, then we have $u \preceq_d w \preceq_d C_k(\prec_d z) \preceq_d y \prec_d v$.

Proof: Since each probe operation involve 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 positional relationships between x and y.

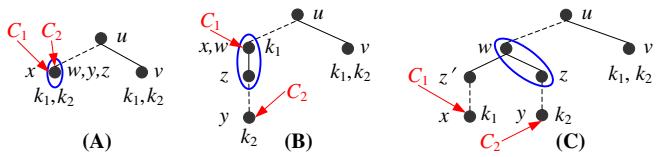


Fig. 5: Illustration of the relationship between nodes in an XML document, where each solid (dashed) line from u to v means $u \prec_p v (u \preceq_a v)$.

(A) $x = y$ (Fig. 5 (A)), which in turn consists of two 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 $\boxed{u, ..., y, v}$, 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 $u \preceq_d w \preceq_d C_1 \preceq_d y \prec_d v$.

(B) $x \prec_a y$ (Fig. 5 (B)), in this case, the nodes between u and v in L_2 must be as $\boxed{u, ..., x, ..., z, ..., y, v}$, 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 $u \preceq_d w \preceq_d C_1 \prec_d z \preceq_d y \prec_d v$.

(C) $x \prec_d y$ and $x \not\prec_a y$ (Fig. 5 (C)), in this case, the nodes between u and v in L_2 must be as $\boxed{u, ..., w, ..., z, ..., y, v}$, 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 $u \preceq_d w \preceq_d C_1(\prec_d z) \preceq_d y \prec_d v$, where "$(\prec_d z)$" denotes that z does not exist in case (A). 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, $u \preceq_d w \preceq_d C_k(\prec_d z) \preceq_d y \prec_d v$. ■

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 $x = y$. Thus the probe operation can be largely simplified in practice. We implement the optimized backward algorithm, i.e., BwdSLCA$^+$, which is same as BwdSLCA except that in function bwdGetCA, bwdBinSearch is replaced by bwdBinSearch$^+$. 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). Line 2-7 will recursively compute the minimum interval, which correspond to case (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 keyword query Q=$\{XML, Tom\}$ and its two IDLists in Fig. 3, the BwdSLCA$^+$ algorithm will firstly find the SLCA node 16, then in line 7-9, node 14 is skipped without further processing, 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 forwardly to Pos 3 and 5, respectively. In the third iteration, C_{XML} pointing node 6 at Pos=3 is chosen as the eliminator to probe L_{Tom}, BwdSLCA$^+$ will call setInterval to set the search interval as [2,4] for L_{Tom}. The third iteration will skip node 4 and 1 when meeting them, and return FALSE to $flag$ to terminate the processing. □

Obviously, the BwdSLCA and BwdSLCA$^+$ algorithms have the same time and space complexity to that of FwdSLCA.

VI. ELCA Computation

A. Forward Solution

For a given keyword query Q, according to Lemma 1, $ELCA(Q) \subseteq CA(Q)$, thus we can still use the fwdCA function of Algorithm 1 to find all CA nodes. Different with 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 are sure that all descendant CA nodes of v are processed.

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 $N =< N_1, N_2, ..., N_m >$ denoting N_{Desc} values of all keywords in Q, the other is $n =< n_1, n_2, ..., n_m >$ used to online adding up the N value of e's child nodes in the CA tree. According to Lemma 3, when e is popped out 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 N value of w to n of the top element of S, then check whether w is an ELCA according to Lemma 3.

As shown in Algorithm 3, 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 founded, then we can stop the processing (line 10) and pop out each element of S to check their satisfiability (line 13-17). If S is empty or the top element of S is the parent node of the current CA node, we directly push it into S (line 9); otherwise, if S is not empty and the top element of S is not the parent of the current CA node v, we need to pop out from S all elements that are not parent of v (line 5). For each element w popped out from S, we firstly modify n's value of the top element of S in line 6, then check whether w is an ELCA by calling function isELCA, and output

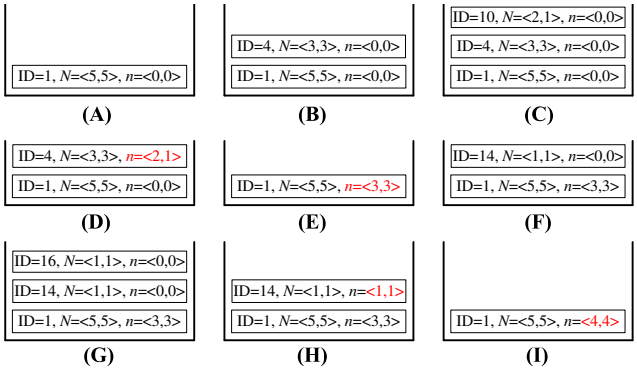


Fig. 6: Forward ELCA computation of Example 6.

$w.ID$ 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.

Algorithm 3: 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;
```

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 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. 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 out from S and the processing stops. After each element is popped from S, we check its satisfiability and add its N value to n of the top element in S. Note that node 14 is not an ELCA node, since according to Fig. 6 (H), each N_i of node 14 equals to its n_i. The last popped node, i.e., node 1, is an ELCA node, because there is no N_i of node 1 equals to its n_i. □

B. Backward Solution

BwdELCA computes all CA nodes in 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

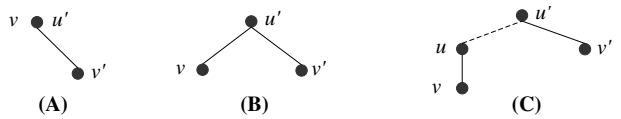


Fig. 7: The positional relationship for CA nodes, each solid (dashed) line from u to v means $u \prec_p v (u \prec_a v)$.

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 is found 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' the parent of v', v the CA node of the current iteration, then $u' \preceq_d v \prec_d v'$.

Proof: As shown in Fig. 7, there are three kinds of positional relationships for u', v' and v: Case 1 (Fig. 7 (A)), u' has not other descendant CA nodes except v', thus the next CA node must be u' itself after visiting v', i.e., $u' = v \prec_d v'$. Case 2 (Fig. 7 (B)), v is a leaf node of the CA tree and the first child node of u' before v' in document order, thus $u' \prec_p v \prec_d v'$. Case 3 (Fig. 7 (C)), v is a leaf node of the CA tree and the first descendant node of u' before v' in document order, where u is the parent node of v, thus $u' \prec_d v \prec_d v'$. By summarizing the above three 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 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 relationship between v, u and u', where u' is the parent of the previous CA node.*

Consider the three positional relationships between v, u and u' in 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 out 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.

As shown in Algorithm 4, 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 are 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 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. 7 (A)), it means that all CA descendant nodes of v has been processed, then we need to pop v from S (line 10), then 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 the parent of v into S in line 14; (2) If the top element of S equals to v's parent, we need to add up the N value of v to the top element of S in line 15; (3) the parent of v is an descendant of the top element of S, we directly push v's parent into S in line 16. If v does not equal to the top element of S (line 18-20), it means that v is 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. 7 (B), we add up the N value of v to the top element of S in line 19; otherwise (the case of Fig. 7 (C)), we directly push v's parent to S in line 20. Note that in the BwdELCA algorithm, we use the optimization technique proposed in Section V-C to accelerate the locating of a node in other lists as shown in function bwdBinSearch.

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. □

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. Therefore in this section, we mainly present the experimental results to verify the benefits of our methods on CA, SLCA and ELCA computation.

Algorithm 4: BwdELCA(Q)

```
1  while (¬ bwdEof(L_i)) do
2    {flag, parent, v} ← bwdGetCA();
3    if (flag = TRUE) then
4      if(isEmpty(S)) then
5        output v.ID as an answer
6        if(parent.ID = −1) then break
7        pushParent(S, parent, v)
8      else
9        if(v.ID = top(S).ID) then
10         w ← pop(S)
11         if(isELCA(w)) then output w.ID as an answer
12         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 ∈ [1, m]) do bwdAdvance(C_i)
25 endwhile
```

Function pushParent$(S, parent, v)$

```
1  foreach(i ∈ [1, m]) do parent.n_i ← v.N_i
2  push(S, parent)
```

Function bwdGetCA() /*Based on fwdGetCA*/

```
2  k ← 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} ← setInterval(L_j, u)
8  if (s > e) then C_j ← L_j[e]
```

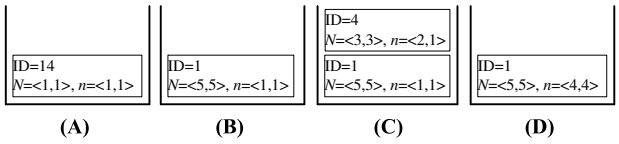


Fig. 8: Backward ELCA computation of Example 7.

A. Experimental Setup

Our experiments were implemented on a PC with Pentium4 2.8 GHz CPU, 2 G memory, 500 GB IDE hard disk, and Windows XP professional as the operating system.

The algorithms used for comparison consist of three groups: (1) **CA computation**. Algorithms of this group target at finding the common node from a set of IDLists, including SvS [16], Small Adaptive (SA) [16], Quantile-based (QB) [17], Probabilistic Intersection (PI) [19], FwdCA, BwdCA and BwdCA$^+$, among which the last three algorithms are the simplified versions of FwdSLCA, BwdSLCA and BwdSLCA$^+$, respectively, where only function fwdGetCA, bwdGetCA and bwdGetCA$^+$ are called to return CA nodes[3]. All these algorithms were implemented based on binary search to make a fair comparison.

(2) **SLCA computation**. Algorithms of this group focusing on SLCA computation, including Stack [4], IL [4], IMS [7], JDewey [6], and the three algorithms proposed in this paper, i.e., FwdSLCA, BwdSLCA and BwdSLCA$^+$.

(3) **ELCA computation**. Algorithms of this gorup focusing on ELCA computation, including DIL [2], IS [13], HC [3], JDewey [6], and the two algorithms proposed in this paper for ELCA computation, i.e., FwdELCA and BwdELCA.

All algorithms were implemented using Microsoft VC++. The running time is the average time by executing each algorithm 100 times on hot cache.

We use XMark dataset of 582MB for our experiments, which contains 8.35 million nodes, the maximum depth and the average depth of the XML tree are 12 and 5.5 respectively. The comparison of the index sizes is shown in Table I, where "Naive" means that each number is stored as an integer, using 4 Bytes. For SLCA computation, we store ID and PIDPos for each node, and the initial index size is smaller than that of Dewey, however, the compression ratio is not as significant as that

[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.

Storage	Dewey	ID+PIDPos	ID+PIDPos+N_{Desc}
Naive	1.5GB	1.17GB	1.76GB
UTF-8	587MB	887MB	1.19GB

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
$N_{L_{Dewey}}$	3252	6013	1704	2255	3016	3869	4823	3306	4192	6981
$\|L_{ID}\|$	2281	4014	1198	1355	2071	2674	3302	2309	2857	4867
Keyword	male	takano	order	school	check	education	female	province	privacy	gender
$\|L_{Dewey}\|$	18441	17129	16797	23561	36304	35257	19902	33520	31232	34065
$N_{L_{Dewey}}$	113081	100338	116429	143894	213449	187843	113081	173900	129455	177450
$\|L_{ID}\|$	62162	53324	68049	91138	106974	115318	70747	105795	66244	108098
Keyword	bidder	listitem	keyword	bold	text	time	date	emph	incategory	increase
$\|L_{Dewey}\|$	299018	304969	352121	368544	535268	313398	457232	350560	411575	304752
$N_{L_{Dewey}}$	1196072	2353169	3072825	3218152	4086735	1616385	2463062	3057771	2057875	1542891
$\|L_{ID}\|$	353220	574800	1236016	1270061	1670362	730709	1112961	1229698	520333	683370

of Dewey, since each component of a Dewey label is smaller than IDs of our methods. The last column shows the index size for ELCA computation.

We have selected 30 keywords classified into three categories according to their frequencies (i.e. $|L_{Dewey}|$ line in Table II): (1) low frequency (100-1000), (2) median frequency (10000-40000), (3) high frequency (300000-600000). The queries in our experiment are generated from these keywords.

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 line $N_{L_{Dewey}}$, which is the sum of the lengths of all Dewey labels. The number of nodes our methods need to process is shown in line $|L_{ID}|$. By comparing the numbers in line $N_{L_{Dewey}}$ and $|L_{ID}|$, we know that the ratio of processed nodes for each keyword between our methods and existing methods is at most 0.7 (necklace), and with the increase of keyword frequency, the ratio decreases dramatically, the minimum value is 0.24 (listitem).

C. CA Computation

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

1) Evaluation Metrics: The metrics for evaluating CA computation include: (1) running time; (2) number of probe operation; (3) number of comparison operation, which is affected by both the search interval and search method. To make a fair comparison, we use binary search across all methods, since the optimization technique introduced in Section V-C is orthogonal to search methods.

2) 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 it processes lists that cannot be partitioned into sublists, such as Q3 and Q4, it degenerates to SvS. PI is also efficient on average, but when QB meets lists that can be partitioned into sublists of uneven distribution, QB beats PI. Our FwdCA beats existing methods for all queries in running time.

The reason can be explained according to Fig. 10. As shown in Fig. 10 (A), the number of probe operation of our method is much less than existing methods. Note that except Q1, Q3 and Q4, the number of probe operation of QB is much larger than that of PI, but as shown in Fig. 9 (A), QB is similar to or more efficient than PI in many cases, the reason can be further

TABLE III: Queries used in CA computation.

ID	Keywords	# of Result	Group
Q1	bold, increase	34136	G1
Q2	bold, increase, text	34136	
Q3	bold, increase, text, keyword	25346	
Q4	bold, increase, text, keyword,emph	20766	
Q5	data, listitem	50706	G2
Q6	data, listitem, bidder	16355	
Q7	data, listitem, bidder, time	16355	
Q8	data, listitem, bidder, time, text	16355	

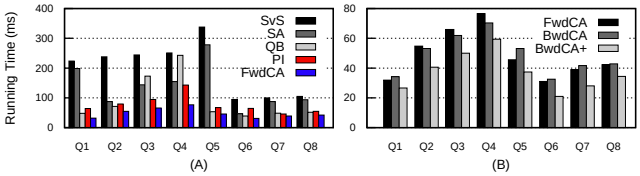


Fig. 9: Comparison of running time for CA computation.

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

#	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
N_1	534767	865241	1169748	1450364	1026313	512245	719006	915088
N_2	213617	318335	385424	439540	311322	102162	132571	189292
N_3	34136	68272	76138	88036	50706	32710	49055	65420

explained according to Fig. 10 (B), from which we know 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 because of the least number of comparison operation.

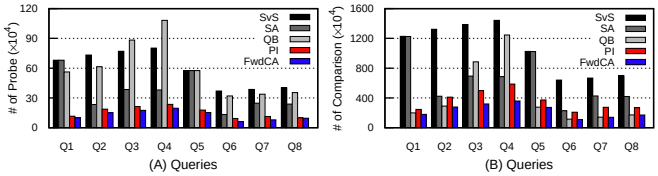


Fig. 10: Number of probe and comparison operation.

3) Impacts of Reducing Search Interval: Fig. 9 (B) presents the running time of FwdCA, BwdCA and BwdCA$^+$, and we find: (1) BwdCA does not always beat FwdCA, (2) BwdCA$^+$ beats FwdCA and BwdCA for all queries.

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 operation when compared with FwdCA for Q1 to Q4. However, when the number of probe and comparison operation 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 operation, FwdCA may be more efficient than BwdCA. Although BwdCA$^+$ processes the same number of probe operation as that of BwdCA, from Fig. 11 (B), we find BwdCA$^+$ needs much less comparison operation than BwdCA, because BwdCA$^+$ can reduce the search interval for each probe operation.

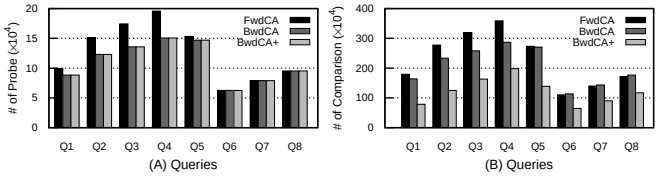


Fig. 11: Number of probe and comparison operation.

Note that as shown in Table IV, the number of comparison operation presented in 11 (B) for BwdCA$^+$ consists of three parts: (1) N_1, the number of actual comparison operation in bwdBinSearch function, (2) N_2, the number of comparison operation in setInterval function to reduce the search interval, and (3) N_3, the number of comparison in line 2 of 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, except that the maximum (minimum) eliminator picked from line 2 of the fwdGetCA (bwdGetCA) function always equals to the one chosen in random manner as SA does,

which is not the case frequently occurred in practice, as shown in Fig. 9 (A) and Fig. 10; (2) the optimization technique used by BwdSLCA$^+$ can improve the overall performance remarkably, as shown in Fig. 9 (B) and Fig. 11.

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 operation.

The reason we use the second metric is that for existing methods except for JDewey, most time is spent on computing the positional relationship and LCA for two Dewey labels by each time comparing two numbers; for our methods and JDewey, most time is spent on binary search operation in comparing two numbers. Therefore, we take the number of comparison operation as an objective metric, which can largely explain the results of the first metric.

For Dewey based methods, the total number of comparison operations, 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 document order for two Dewey labels, M the number of LCA and document order operations. For JDewey, N_C can be computed using the same formula, where n_i is the number of search step of a binary search operation, M the number of binary search operations. For our methods, the number of comparison operation 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 query 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.

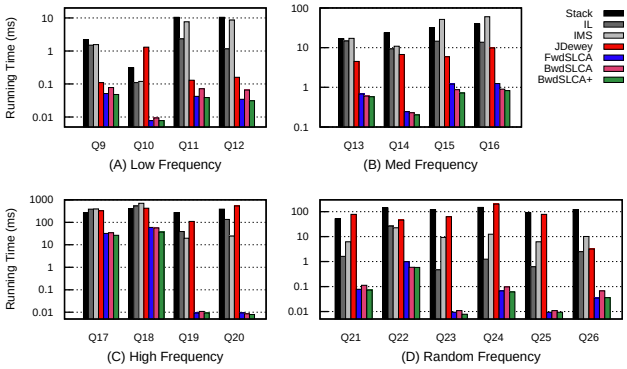


Fig. 12: Comparison of running time for SLCA computation.

For existing methods, since Stack processes Dewey labels in document order, its performance is mainly affected by the number of processed Dewey labels. IL and IMS algorithms can use the positional relationship between nodes to avoid processing many useless Dewey labels, 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, such as Q9 to Q20, the chances of skipping useless elements are largely reduced, therefore 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 operation they have done. From Fig. 12, we also know that IMS performs better when the result selectivity is very high, such as Q10, Q19 and Q20, but it usually performs worse than Stack and IL when the result selectivity becomes low, such as Q13, Q17 and Q18. Although JDewey is also 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. As shown in Fig. 13, JDewey suffers from less comparison operation than other existing methods in many cases, but its performance is not always more efficient than other existing methods.

From Fig. 13 we have several observations: (1) our methods always need the least comparison operation, therefore achieves the best performance for all queries; (2) BwdSLCA is not always more efficient than FwdSLCA, when the cost of the saved probe operation is less than the additional cost for skipping useless probe operations, FwdSLCA is more efficient; (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 operation of BwdSLCA$^+$ is less than that of FwdSLCA and BwdSLCA, (3.2) even in some cases, such as Q9 to Q12 of low frequency, Q23, Q25 and Q26 of random frequency, BwdSLCA$^+$ needs more

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	

comparison operations, as shown in Table IV, 15% to 30% comparison operation exists in the setInterval function, which can be done much more efficient than the comparison operation in the probe operation. Even though it is not remarkably significant according to Fig. 12 in log scale, BwdSLCA$^+$ actually saves 5% to 45% cost compared with FwdSLCA and BwdSLCA for Q9 to Q26.

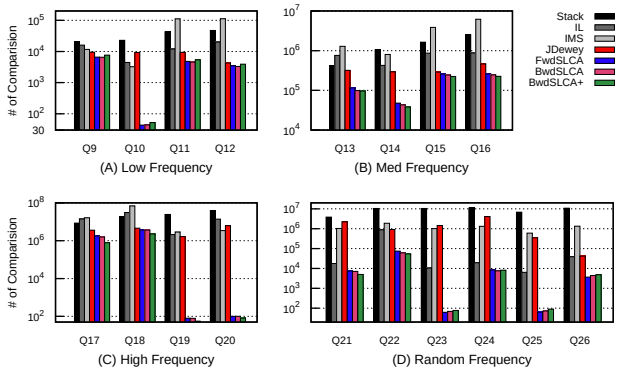


Fig. 13: Number of comparison operation.

From the above discussion, we conclude that our methods outperform all existing methods, especially for queries with high result selectivity (see Table V). The reasons lie in that (1) the number of processed nodes (see Section VII-B) is largely reduced; (2) the positional relationship between nodes can be determined with $O(1)$ cost; and (3) compared with JDewey, which is also a set intersection based method, 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 until the root.

3) Scalability: Fig. 14 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 result and therefore omitted.

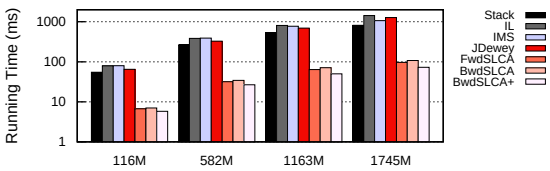


Fig. 14: Comparison of query running time for Q17 on XML documents of different sizes.

E. ELCA computation

Fig. 15 shows the running time of different algorithms on query Q9 to Q26 for ELCA computation, from which we have similar observations to that of SLCA, i.e., (1) for existing algorithms, no one can beat others for all queries, (2) our methods perform much better than existing methods for all queries.

As shown in Fig. 15, 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 relationship 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. 15 (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. For ELCA computation, JDewey can be more efficient than that of SLCA computation, the reason lies in that less nodes need to be deleted from all lists of higher levels, however, it still cannot be more efficient than HC in many cases since the cost of deleting nodes from lists of higher levels could be large in practice.

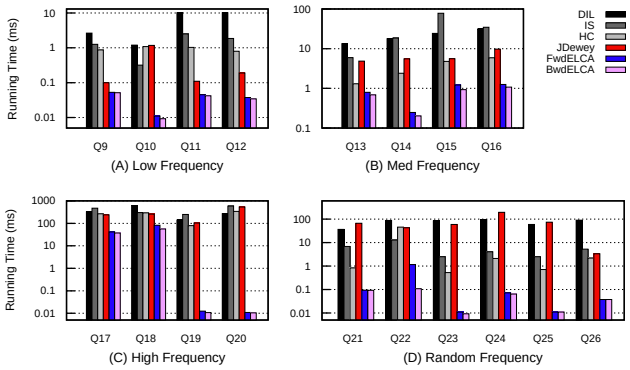


Fig. 15: Comparison of running time for ELCA computation.

In contrast, our methods always need the least comparison operation (similar to that of SLCA computation and therefore omitted), thus achieves the best performance for all queries. From Fig. 15, we can also see that BwdSLCA is more efficient than FwdELCA, because BwdELCA uses the optimization techniques introduced in Section V to reduce the search interval for each probe operation.

F. Implementing Disk-based Index

It is difficult to find from an IDList the parent node by PIDPos if IDList is organized as a B^+ tree. A naive way is storing the Pos value together with ID, and the operation of finding the parent node is thus a lookup operation on the B^+ tree, which is space inefficient and is time consuming when doing lookup operation directly.

Considering this problem, we choose to implement a disk-based B^+ tree index by replacing the PIDPos value with the ID of the parent node, i.e., PID, rather than the Pos value of the parent node. In this way, we do not need to use PIDPos to find the parent ID, and each probe operation now becomes a lookup operation on the B^+ tree, and we can still compute SLCA and ELCA nodes without changing the forward and backward solution. The only problem is that the optimization technique of Section V-C cannot be used to accelerate the probe operation.

Fig. 16 shows the running time of FwdSLCA, BwdSLCA, BwdSLCA$^+$, and the two algorithms utilizing disk-based index, i.e., FwdSLCA-PID and BwdSLCA-PID. We can see that although FwdSLCA-PID and BwdSLCA-PID cannot utilize the optimization technique of Section V-C, their performance is still very efficient, just a little slower than BwdSLCA$^+$.

VIII. Related Work

In the past years, many query semantics [1], [4], [5], [10] have been proposed to define matched results for an XML keyword query, of which the two most widely followed variants are SLCA [4], [7] and ELCA [2], [3], [13], which have been described in Section II. Considering that returning all answers is not necessary in some cases, [2], [6], [9], [20] investigated the problem of assigning to each result a score and return to users results with the highest score, among which [9] focuses on how to infer users' search intention.

To facilitate computing the SLCA and ELCA results on XML data, existing methods [2]–[4], [7]–[9] assign to each node v a Dewey [12] label 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 position relationships between two Dewey labels. Unfortunately,

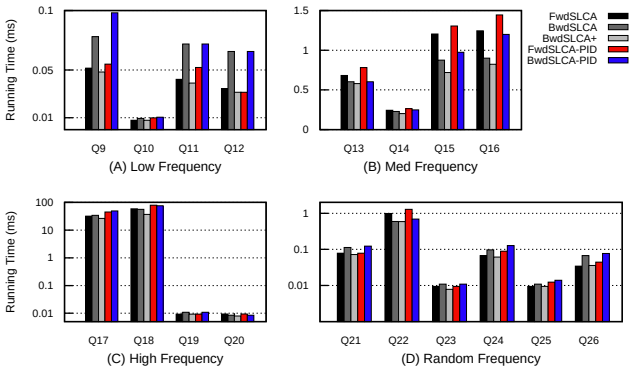


Fig. 16: Comparison of running time for SLCA computation.

for an XML tree of depth d, the cost of testing their position 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 operation 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 in the same tree depth. According to this property, the algorithms proposed in [6] do set intersection operation on all lists of each tree depth from the leaf to the root. The problem of these methods lies in the complexity of implementation, that is, after finding a common node from a set of lists corresponding to a tree depth, the ancestor information needs to be deleted from each set of lists of tree depth less than the current depth, which is costly in practice and difficult to make optimization. The closer the tree depth to the root, the more cost to delete ancestor information is

The problem of intersecting a set of sorted lists (or sets) has also been extensively studied in the past years. A simple yet efficient method is SvS [16], which intersects two sets at a time in increasing order by size, starting with the two smallest sets. It is a blocking algorithm and not efficient when the two joining lists have similar lengths. The Adaptive algorithm [18] 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 [16]. To address this problem, the Small Adaptive algorithm [16] 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 [19] is based on the Adaptive algorithm. It uses a probabilistic probing policy to determine which list to examine next based on historical data called "average jump". Both the Dynamic Probes algorithm [17] and Quantile-based algorithm [17] use interpolation search to accelerate the probe operation. The Dynamic Probes algorithm dynamically decides the join order on the lists exploiting information from previous probes at runtime. The Quantile-based algorithm [17] targets at processing situations in which the distribution of components is highly non-uniform, which deduces in advance a good join order by first partitioning all lists into sub-partitions, thus avoiding the overhead of adaptivity in choosing the join order. For each sub-partition, the join order is same as SvS.

IX. Conclusions

Considering that the CAR problem is the bottleneck for existing Dewey based algorithms for SLCA and ELCA computation, we propose to use IDLists, rather than inverted Dewey label lists, for SLCA and ELCA computation. An immediate benefit is that the total number of processed nodes is at most reduced to 24% of that of Dewey based methods. For SLCA computation, we firstly propose the FwdSLCA algorithm that reduces the number of probe operation by always using the largest ID to probe the shortest list; then we propose the BwdSLCA algorithm that utilizes the fact that non-leaf CA nodes are not SLCA nodes to reduce the number of probe operation; finally, we propose the BwdSLCA$^+$ algorithm that utilizes the positional relationship between XML nodes to reduce the search interval for probe operation. Similar to our SLCA algorithms, we propose the FwdELCA and BwdELCA algorithms for ELCA computation. Experiments verify that the performance of our methods outperforms existing methods by more than 2 orders of magnitude in most cases.

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] 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.
[13] Y. Xu and Y. Papakonstantinou, "Efficient lca based keyword search in xml data," in *EDBT*, 2008.
[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] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Experiments on adaptive set intersections for text retrieval systems," in *ALENEX*, 2001.
[17] D. Tsirogiannis, S. Guha, and N. Koudas, "Improving the performance of list intersection," *PVLDB*, vol. 2, no. 1, 2009.
[18] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Adaptive set intersections, unions, and differences," in *SODA*, 2000.
[19] 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.
[20] V. Hristidis, Y. Papakonstantinou, and A. Balmin, "Keyword proximity search on xml graphs," in *ICDE*, 2003.