Top-Down XML Keyword Query Processing

Junfeng Zhou[1] Xingmin Zhao[1] Wei Wang[2]
Ziyang Chen[1] Jeffrey Xu Yu[3] Xian Tang[1]

[1] Yanshan University, China
{zhoujf,zxm,zychen,txianz}@ysu.edu.cn

[2] University of New South Wales, Australia
weiw@cse.unsw.edu.au

[3] The Chinese University of Hong Kong, China
yu@se.cuhk.edu.hk

Technical Report
UNSW-CSE-TR-201321
2013-12

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Efficiently answering XML keyword queries has attracted much research effort in the last decade. The key factors resulting in the inefficiency of existing methods are the *common-ancestor-repetition* (CAR) and *visiting-useless-nodes* (VUN) problems. To address the CAR problem, we propose a *generic top-down* processing strategy to answer a given keyword query w.r.t. LCA/SLCA/ELCA semantics. By "*top-down*", we mean that we visit all *common ancestor* (CA) nodes in a depth-first, left-to-right order; by "*generic*", we mean that our method is independent of the query semantics. To address the VUN problem, we propose to use child nodes, rather than descendant nodes to test the satisfiability of a node v w.r.t. the given semantics. We propose two algorithms that are based on either traditional inverted lists or our newly proposed LLists to improve the overall performance. We further propose several algorithms that are based on hash search to simplify the operation of finding CA nodes from all involved LLists. The experimental results verify the benefits of our methods according to various evaluation metrics.

1 Introduction

Keyword search on XML data has received much attention in the literature [3–5, 7, 12–14, 17, 20, 22, 23, 27], of which one of the key issues is how to efficiently answer a given query w.r.t. the given query semantics.

Typically, an XML document can be modeled as a node-labeled tree T. For a given keyword query Q, several semantics [5, 7, 12, 13, 22] have been proposed to define meaningful results, for which the basic semantics is *Lowest Common Ancestor* (**LCA**). Based on LCA, the most widely adopted query semantics are Exclusive LCA (**ELCA**) [3, 7, 10, 23, 27] and Smallest LCA (**SLCA**) [3, 14, 17, 20, 22]. SLCA defines a subset of LCA nodes, of which no LCA is the ancestor of any other LCA. SLCA requires the resulting LCAs to be lowest, such that each query result is a tightest XML fragment containing all the query keywords. As a comparison, ELCA tries to capture more meaningful results, it may take some LCAs that are not SLCAs as meaningful results. Finding all LCA/ELCA/SLCA nodes is the core operation for XML keyword query processing, based on which subtree results meeting certain constraints [8, 10, 14] can then be generated for effectively identifying useful information from the underlying data. During the past few years, even though researchers have proposed many algorithms [3, 7, 8, 17, 20, 22–25, 27] on LCA/ELCA/SLCA computation, these methods still suffer from *redundant computation*.

Given a keyword query Q and XML document D, let V be the set of nodes of D that contain at least one query keyword in their subtrees, we can classify all nodes of V into three categories: (1) *common ancestors* (CAs), each of which contains all query keywords in its subtree, such as nodes of V_1 in Fig. 1.1 for Q_2; (2) *useless nodes* (UNs), each of which is not a child of any CA node, for Q_2, all nodes of V_2 in Fig. 1.1 are UNs; (3) *auxiliary nodes* (ANs), each of which is a child node of some CA node and belongs to $V - V_1 \bigcup V_2$, such as V_3 in Fig. 1.1. Generally speaking, the common problems that result in *inefficiency* for existing algorithms are the *CAR* and *VUN* problems.

The CAR problem: As first identified in [24, 25], existing methods [3, 7, 10, 14, 17, 20, 22, 23, 27] assign each node v a Dewey label [18], or one of its variants [3, 12, 20, 27], based on which two basic operations are adopted to compute qualified results, i.e., ($OP1$) testing the document order of two nodes and ($OP2$) computing the LCA of two nodes. However, as each Dewey label consists of a set of components that collectively represent a node v, and each component itself corresponds to a node on the path from the root r of the XML tree to v, either $OP1$ or $OP2$ operation equals visiting all CAs of the two involved nodes once. In practice, as v could be a CA of multiple nodes, frequently performing $OP1$ and $OP2$ operations will result in all CAs on the path from r to v be repeatedly visited, which is called as *common-ancestor-repetition* (CAR). The CAR problem is inherent in algorithms [7, 14, 17, 22, 23] that are based on $OP1$ and $OP2$ operations, because they take Dewey label as the basic processing unit, without noticing that some components repeatedly appear in many different Dewey labels.

Although some recently proposed methods [24, 25] address this problem by employing IDList as the basis for SLCA/ELCA computation, the flattened index structure of IDList makes them victim of the VUN problem presented below.

The VUN problem: *none* of existing methods can avoid *visiting useless nodes*, which we call as the VUN problem, as shown by Example 1.

Example 1. *Since the basic operations of Stack [22], DIL [7], IS [23], IL [22] and IMS [17] are $OP1$ and $OP2$, they need to repeatedly select two nodes and compare their Dewey labels. E.g., for query Q_2 in Fig. 1.1, to get the LCA of nodes 8 and 9, they need to visit nodes 6, 8 and 9; to get the LCA of nodes 18 and 19, they need to visit nodes 16, 18 and 19. JDewey-S [3] and JDewey-E [3][1] compute ELCA/SLCA results by performing the set intersection operation on all lists of each tree depth from the leaf to the root, they will firstly visit nodes of V_2. For FwdSLCA, BwdSLCA, BwdSLCA$^+$, FwdELCA and BwdELCA [24], since the IDList of each keyword k_i consists of all nodes that contain k_i, the set intersection operation on all IDLists may visit any node, including UNs. For hash search based algorithms, since HC [27] needs to push each component of every IDDewey [20, 27] label of the shortest inverted list into a stack, they will process UNs if some nodes of an IDDewey label are UNs. Even though HS [20] does not need to check the satisfiability of all nodes of an IDDewey label by using binary search to select processed nodes, it still needs to visit UNs if the selected nodes are UNs.* □

[1] In [3], there doesn't exist such a name, we use JDewey-S and JDewey-E to denote the algorithm on SLCA and ELCA computation based on JDewey labeling scheme, respectively.

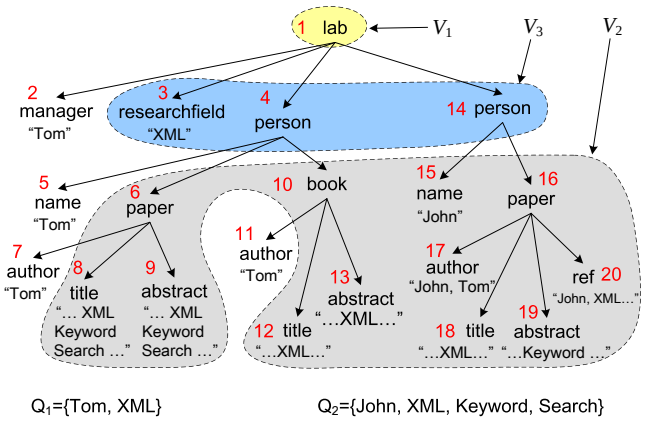

Figure 1.1: A sample XML document D.

The reason that existing methods [3, 7, 14, 17, 20, 22–25, 27] suffer from the VUN problem lies in that, essentially, the satisfiability of each node v w.r.t. LCA, SLCA or ELCA semantics is determined by v's *descendant* nodes, rather than its *child* nodes.

Considering the above problems, we propose a suite of novel algorithms to improve the overall performance. Specifically, we make the following contributions.

(1) To address the CAR problem, we propose a *generic top-down* XML keyword query processing strategy. The "*top-down*" means that our methods take the component of Dewey labels as the basic processing unit, and visit CA nodes in depth-first left-to-right order. The "*generic*" means that our methods can be used to find xLCA (xLCA can be either one of LCA, SLCA and ELCA) results.

(2) To address the VUN problem, we propose to use child nodes, rather than descendant nodes, to test the satisfiability of a node v w.r.t. xLCA semantics. E.g., for either one of LCA, SLCA and ELCA, the qualified result is node 1 for Q_2 in Fig. 1.1, our method only needs to visit nodes of $V_1 \bigcup V_3$, rather than additionally visit useless nodes in V_2. We then propose a top-down algorithm, namely TDxLCA, based on traditional inverted lists to get xLCA nodes.

(3) We propose a *labeling-scheme-independent* inverted index, namely LList, which maintains every node in each level of a traditional inverted list only once and keeps all necessary information for answering a given keyword query without any loss. Based on LLists, our second top-down algorithm, namely TDxLCA-L, further reduces the time complexity.

(4) To further improve the overall performance, we consider the existence of additional hash indexes [20, 25, 27] and propose new algorithms to accelerate xLCA computation.

(5) We conducted an extensive set of performance studies to compare our proposed algorithms with the state-of-the-art algorithms. The experimental results verify the benefits of our methods according to various evaluation metrics.

As an extension of [26], we have several major updates: (1) We propose several algorithms that are based on hash search to accelerate xLCA computation in Section 7. (2) We discuss the extension of our methods to other LCA-based semantics in Section 6. (3) We conduct additional performance study in Section 8. (4) We give an in-depth analysis to related work in Section 2.3.

The rest of the paper is organized as follows. In Section 2, we introduce preliminaries and related work. In Section 3, we introduce the basic idea of our top-down processing strategy. The algorithm for ELCA computation based on traditional inverted list is presented in Section 4. In Section 5, we introduce the LList index and its properties, and then discuss LList based algorithm. In Section 6, we discuss how to compute qualified LCA/SLCA results based on our top-down processing strategy. In Section 7, we present algorithms for xLCA computation based on hash search. In Section 8, we present experimental results and then conclude our work in Section 9.

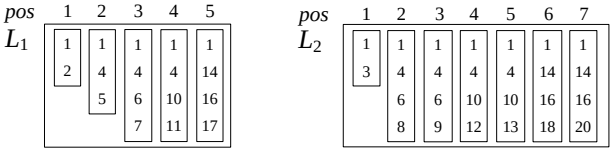

Figure 2.1: Inverted IDDewey label lists of "Tom" (L_1) and "XML" (L_2), where *pos* denotes the array subscript.

2 Preliminaries and Related Work

2.1 Data Model

We model an XML document as an ordered tree (Fig. 1.1), where nodes represent elements or attributes, while edges represent direct nesting relationship between nodes. We say node v *directly* contains k or v is a keyword node w.r.t. k, if k appears in the node name, attribute name, or text value of v; otherwise, if k is directly contained by some descendant nodes of v, we say v contains k.

Given two nodes u and v, $u \prec_d v$ means that u is located before v in *d*ocument order, $u \prec_a v$ means that u is an *a*ncestor node of v, $u \prec_p v$ denotes that u is the *p*arent node of v. If u and v represent the same node, we have $u = v$, and both $u \preceq_d v$ and $u \preceq_a v$ hold.

To accelerate the query processing, existing methods [3, 7, 14, 17, 20, 22–24, 27] usually assign each node v a uniquely label to facilitate the testing of the positional relationships between nodes. The assigned label for each node can be an ID which is compatible with the document order [24, 25], a Dewey label [18] or one of its variants [3, 12, 16, 20, 27]. In Fig. 1.1, we denote each node v by its ID, based on which its IDDewey [20, 27] label can be easily got by concatenating all IDs on the path from the root node to v. E.g., the IDDewey label of node "5" is "1.4.5". In the following discussion, we do not differentiate a node, its ID, and the corresponding IDDewey label unless there is ambiguity.

Note that in practice, integer is not the unique choice for storing components of Dewey labels, we can also use binary string [11] to encode them. However, the CAR problem still exists for existing methods since each Dewey label l consists of a set of binary strings, of which each one represents a component of l. If we choose to use containment labeling scheme [9], then it's difficult to compute LCA for the given two nodes. Another choice is to use prime labeling scheme [21], which uses an SC (Simultaneous Congruence) value to derive the mapping from self labels to global orders. To prevent the SC value from getting too large, a list of SC values has to be used. In practice, as an XML document could contain a very large number of nodes, the list of SC values can be very long, which results in higher cost in computing the LCA and document order between different nodes. In this paper, components of Dewey labels are represented as integers in memory for efficient computation, which does not conflict with the specific storing format in disk.

2.2 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 that consists of IDDewey labels sorted in document order. Fig. 2.1 shows the inverted lists for keywords of Q_1 in Fig. 1.1.

Definition 1. ***(CA)*** *Given a query Q and XML document D, the set of CA nodes of Q on D is $CA(Q) = \{v | v$ contains each keyword of Q at least once$\}$.*

E.g., for query Q_1 and D in Fig. 1.1, CA nodes are nodes 1, 4, 6, 10, 14 and 16.

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 are defined as follows.

Definition 2. ***(LCA)*** *Given a query Q=$\{k_1, k_2, ..., k_m\}$ and XML document D, the set of LCAs of Q on D is $LCA(Q)$=$\{v | v = lca(v_1, v_2, ..., v_m), v_i \in L_i (1 \leq i \leq m)\}$.*

According to Definition 2, LCA nodes of Q_1 on D in Fig. 1.1 are nodes 1, 4, 6, 10 and 16.

SLCA [17, 22] is one of the widely adopted query semantics, which defines a subset of $LCA(Q)$, of which no LCA is the ancestor of any other LCA, as shown by Definition 3.

Definition 3. ***(SLCA)*** *Given a query Q and an XML document D, the set of SLCAs of Q on D is $SLCA(Q) = \{v | v \in LCA(Q)$ and $\nexists v' \in LCA(Q)$, such that $v \prec_a v'\}$.*

According to Definition 3, SLCA nodes of Q_1 on D in Fig. 1.1 are nodes 6, 10 and 16.

Another widely adopted query semantics is ELCA [7, 23, 27]. Intuitively, a node v is an ELCA if the subtree rooted at v contains at least one occurrence of all query keywords, after excluding the occurrences of the keywords in each subtree rooted at a descendant LCA node of v, as shown by Definition 4.

Definition 4. ***(ELCA)*** *Given a keyword query $Q = \{k_1, k_2, ..., k_m\}$ and XML document D, the set of ELCAs of Q on D is $ELCA(Q)$=$\{v|\exists v_1 \in L_1^D, ..., v_m \in L_m^D (v = LCA(v_1, ..., v_m) \wedge \forall i \in [1, m], \nexists x (x \in LCA(Q) \wedge child(v, v_i) \preceq_a x))\}$, where $child(v, v_i)$ is the child of v on the path from v to v_i.*

According to Definition 4, ELCA nodes of Q_1 on D in Fig. 1.1 are nodes 1, 6, 10 and 16.

2.3 Related Work

Here we only focus on algorithms on ELCA computation, algorithms on SLCA and LCA computation possesses similar problems as ELCA. We refer readers to [4, 15, 25] for a complete coverage. Existing algorithms on ELCA computation can be generally classified into two categories: (1) algorithms that cannot avoid the CAR problem, including DIL [7], IS [23], JDewey-E [3] and HC [27], (2) algorithms that do not suffer from the CAR problem, including FwdELCA and BwdELCA [24, 25].

Among the first kind of algorithms, DIL [7] sequentially processes all involved Dewey labels in document order, its performance is linear to the number of involved Dewey labels. IS [23] sequentially processes all Dewey labels of the shortest list L_1 one by one. In each iteration, it picks from L_1 a Dewey label l and uses it to probe other lists to get a candidate ELCA node. As the basic operations of the two algorithms are $OP1$ and $OP2$, they heavily suffer from the CAR problem. JDewey-E [3] computes ELCA results by performing set intersection operation on all lists of each tree depth from the leaf to the root. For all lists of each level, after finding the set of common nodes, it needs to recursively delete all ancestor nodes in all lists of higher levels. As a node could be a parent node of many other CA nodes, and the deletion operation needs to process each parent-child relationship separately, JDewey-E also suffers from the CAR problem. As some node IDs are repeatedly appearing in many different IDDewey labels of the same inverted list, and HC [27] processes each IDDewey label of the shortest list separately, it still suffers from the CAR problem.

As a comparison, the second kind of algorithms, including FwdELCA and BwdELCA [24, 25], adopt inverted lists of node IDs, i.e., IDList, as the basis of ELCA computation. For each keyword k_i, the corresponding IDList L_i^{ID} consists of IDs of all nodes that contain k_i, where IDs are sorted in ascending order. Based on IDList, CA computation followed by an appropriate pruning is used to implement ELCA computation. Therefore, the performance of FwdELCA and BwdELCA is dominated by the cost of CA computation. For a given keyword query $Q = \{k_1, k_2, ..., k_m\}$, [24, 25] use the set intersection operation on IDLists to get all CA nodes, as shown by Formula 2.1.

$$CA(Q) = \bigcap_{i=1}^{m} L_i^{ID} \tag{2.1}$$

As show in [24, 25], BwdELCA is more efficient than FwdELCA by reducing search interval of binary search operations used in CA computation. Since each ID appears only once in an inverted list, both FwdELCA and BwdELCA avoid the CAR problem. Even so, the flattened structure of IDList makes them still suffer from the VUN problem and higher cost, i.e., $O(\log |L_m^{ID}|)$, in checking the appearance of each node in another IDList (see Formula 2.1), while the cost of our methods is $O(\log N)$ (see Section 5.2 for detailed explanation), where N is usually much less than $|L_m|$, which in turn is much less than $|L_m^{ID}|$. Further, [24, 25] considered the existence of additional hash indexes on IDLists and proposed three algorithms to accelerate ELCA computation, including FwdELCA-HS, BwdELCA-HS and HybELCA-HS. All these algorithms avoid the CAR problem but cannot avoid the VUN problem. The reason why all other existing algorithms cannot avoid the VUN problem has been explained in Example 1.

3 Overview of Our Method

As LCA/SLCA/ELCA nodes are subsets of CA nodes, the basic idea of our method is: *directly compute all CA nodes in a **top-down** way, and check the satisfiability of each CA node w.r.t the given query semantics.*

Assume that for a given query $Q = \{k_1, k_2, ..., k_m\}$, each keyword appears at least once in the given XML document. Intuitively, to get all CA nodes of Q, our method takes all nodes in the set of inverted IDDewey

label lists as leaf nodes of an XML tree T_r rooted at node r, and checks whether each node of T_r contains all keywords of Q in a "*top-down*" way. The "*top-down*" means that if T_r contains all keywords of Q, then r must be a CA node. We then remove r and get a forest $\mathcal{F}_{T_r} = \{T_{v_1}, T_{v_2}, ..., T_{v_n}\}$ of subtrees rooted at the n child nodes of r. Based on $\mathcal{F}_{T_r}$, we further find the set of subtrees $\mathcal{F}^{CA}_{T_r} \subseteq \mathcal{F}_{T_r}$, where each subtree $T_{v_i} \in \mathcal{F}^{CA}_{T_r}$ contains every keyword of Q at least once, i.e., node v_i, is a CA node. If $\mathcal{F}^{CA}_{T_r} = \emptyset$, it means that for T_r, only r is a CA node, then we can safely skip all nodes of T_r from being processed; otherwise, for each subtree $T_{v_i} \in \mathcal{F}^{CA}_{T_r}$, we *recursively* compute its subtree set $\mathcal{F}^{CA}_{T_{v_i}}$ until $\mathcal{F}^{CA}_{T_{v_i}} = \emptyset$.

Let $S^{k_i}_{ch}(v)$ denote, for node v of T_r, the set of child nodes that contain k_i in T_v, $S^{CA}_{ch}(v)$ the set of child CA nodes of v, and $CA(T_v)$ the set of CA nodes in T_v. Formula 3.1 means that the set of CA nodes of Q equals the set of CA nodes in T_r, which can be recursively computed according to Formula 3.2. Formula 3.2 means that for a given CA node v, the set of CA nodes in T_v is equal to the union of $\{v\}$ and the set of CA nodes in the subtree rooted at each of v's child CA nodes, which can be further computed by Formula 3.3.

$$CA(Q) = CA(T_r) \tag{3.1}$$

$$CA(T_v) = \{v\} \bigcup (\bigcup_{u \in S^{CA}_{ch}(v)} CA(T_u)) \tag{3.2}$$

$$S^{CA}_{ch}(v) = \bigcap_{i=1}^{m} S^{k_i}_{ch}(v) \tag{3.3}$$

Example 2. *Consider Q_1 and D in Fig. 1.1. According to Formula 3.1, $CA(Q_1) = CA(T_1)$. Since nodes 4 and 14 are child CAs of node 1 in Fig. 1.1, i.e., $S^{CA}_{ch}(1)$= $\{4, 14\}$, we have $CA(T_1)$=$\{1\} \bigcup CA(T_4) \bigcup CA(T_{14})$ according to Formula 3.2. As $S^{CA}_{ch}(4) = \{6, 10\}$, $S^{CA}_{ch}(14) = \{16\}$ and $S^{CA}_{ch}(6) = S^{CA}_{ch}(10) = S^{CA}_{ch}(16)$= $\emptyset$, we know that $CA(Q_1) = \{1, 4, 6, 10, 14, 16\}$.* □

Corollary 1. *Either one of Dewey [18], JDewey [3], IDDewey [20, 27], MDC [12] and Extended Dewey [16] labeling schemes can be used for top-down CA computation.*

The correctness of Corollary 1 lies in that each child node of a given node v is represented by a distinct ID value compared with its sibling nodes w.r.t. either one of Dewey-like labeling schemes, thus the set intersection operation can correctly return all child CA nodes of v by intersecting v's m sets of child nodes. Based on all CA nodes got by Formula 3.1, our method can efficiently check the satisfiability of each CA node w.r.t. the given query semantics. In the following sections, we focus on *ELCA* computation to illustrate the benefits of our *top-down* processing strategy. We will return to LCA and SLCA computation from Section 6.

4 The Baseline Algorithm

Our baseline algorithm uses inverted IDDewey label lists (Fig. 2.1) for ELCA computation. To recursively get all ELCA nodes in a top-down way, we need to solve two problems: ($P1$) identify the set of child CA nodes for each CA node v, ($P2$) check v's satisfiability w.r.t. ELCA semantics.

For $P1$, according to Formula 3.3, for query Q with m keywords, we know that $\forall i \in [1, m]$, $S^{CA}_{ch}(v) \subseteq S^{k_i}_{ch}(v)$. Therefore, for a given CA node v and its subtree set $\mathcal{F}_{T_v} = \{T_{v_1}, T_{v_2}, ..., T_{v_n}\}$, to get $S^{CA}_{ch}(v)$, we do not need to check whether each $T_{v_x}(x \in [1, n], v \prec_p v_x)$ contains all keywords of Q; instead, we just need to check whether each node of $S^{k_{min}}_{ch}(v) = \{S^{k_i}_{ch}(v) | i \in [1, m], \forall j \in [1, m], j \neq i, |S^{k_i}_{ch}(v)| \leq |S^{k_j}_{ch}(v)|\}$ appears in every $S^{k_j}_{ch}(v)(j \in [1, m], j \neq min)$.

E.g., for Q_1 and D of Fig. 1.1, $S^{Tom}_{ch}(4)$=$\{5, 6, 10\}$, $S^{XML}_{ch}(4) = \{6, 10\}$. Since $|S^{Tom}_{ch}(4)| > |S^{XML}_{ch}(4)|$, we know $S^{k_{min}}_{ch}(4) = S^{XML}_{ch}(4)$. To compute $S^{CA}_{ch}(4)$ for Q_1, we just need to check whether each node of $S^{XML}_{ch}(4)$ is contained by $S^{Tom}_{ch}(4)$, then we know $S^{CA}_{ch}(4) = \{6, 10\}$.

In Fig. 2.1, all nodes of $S^{XML}_{ch}(4) = \{6, 10\}$ appear in L_2 as $\boxed{6,6,10,10}$, which we call as the *child list* of node 4 w.r.t. XML. Obviously, even though all nodes of $S^{k_i}_{ch}(v)$ are different with each other, in the *child list* of v w.r.t. k_i, each node of $S^{k_i}_{ch}(v)$ may repeatedly appear many times. Therefore, even if we know the lengths of all child lists, it's difficult to know which one contains the least number of child nodes. Fortunately, as all

node IDs in each child list of v are sorted in ascending order, our newly proposed set intersection algorithm guarantees that the number of processed child nodes for each CA node v is bounded by $|S_{ch}^{k_{min}}(v)|$.

For *P2*, we use the following Lemma to check the satisfiability of v, which is similar to [24, 25, 27].

Lemma 1. *Given a query $Q = \{k_1, k_2, ..., k_m\}$ and CA node v, let $T_v.N_i$ be the number of occurrences of k_i in T_v, v is an ELCA node iff $\forall i \in [1, m], T_v.N_i - \sum_{u \in S_{ch}^{CA}(v)} T_u.N_i > 0$.*

Proof. Assume that u is a child CA node of v. If u is an LCA node, than the operation of removing all occurrences of $k_i (i \in [1, m])$ from T_u equals removing all occurrences of k_i from every subtree rooted at an LCA node in T_u. If u is not an LCA node, then among all u's child nodes in the original XML tree, according to Definition 1 and 2, there exists a unique child CA node u_c, and all other child nodes of u do not contain any query keyword, which means that the number of occurrences of each k_i in T_u equals that in T_{u_c}. If u_c is not an LCA node either, we can recursively get the closest descendant LCA node of u_c, such that both nodes contain the same number of occurrences for each keyword of Q. Therefore, the operation of excluding the occurrences of the query keywords in each subtree rooted at a descendant LCA node of v equals removing the keyword occurrences from each subtree rooted at a child CA node of v. Since $T_v.N_i$ denotes the number of occurrences of k_i in T_v, if $\forall i \in [1, m], T_v.N_i - \sum_{u \in S_{ch}^{CA}(v)} T_u.N_i > 0$, it means that after excluding the occurrences of all query keywords in the subtree rooted at each of v's child CA nodes, T_v still contains at least one occurrence of all query keywords. According to Definition 4, v is an ELCA node. Similarly, if v is an ELCA node, we have $\forall i \in [1, m], T_v.N_i - \sum_{u \in S_{ch}^{CA}(v)} T_u.N_i > 0$. □

According to Fig. 2.1, $T_1.N_1 = 5$, $T_1.N_2 = 7$ and $S_{ch}^{CA}(1)=\{4, 14\}$. Since $T_4.N_1 = 3$, $T_4.N_2 = 4$, $T_{14}.N_1 = 1$ and $T_{14}.N_2 = 2$, we know $T_1.N_1 - (T_4.N_1 + T_{14}.N_1) = 1 > 0$ and $T_1.N_2 - (T_4.N_2 + T_{14}.N_2) = 1 > 0$, thus node 1 is an ELCA node of Q_1. Compared with [24, 25, 27], our method does not need to pre-compute and maintain $T_v.N_i$'s value, which can be got on the fly.

4.1 The Algorithm

In Algorithm 1, each inverted list L_i is associated with a cursor C_i pointing to some IDDewey label of L_i. C_i will refer to the IDDewey label that C_i points to. We use $\text{pos}(C_i)$ to denote C_i's position in L_i, and $L_i[x]$ to denote the x^{th} IDDewey label of L_i. If C_i points to the x^{th} IDDewey label of L_i, i.e., $\text{pos}(C_i) = x$, we have the assertion that $L_i[x] = C_i$. For a given IDDewey label l, $|l|$ denotes the length of l, i.e., the number of components of l, $l[j]$ denotes the j^{th} component of l. If C_i points to the x^{th} IDDewey label l, then $L_i[x][j] = C_i[j] = l[j]$. Besides, we use T to denote the subtree rooted at a CA node. T has five member variables: (1) $T.r$ is the root node of T; (2) $T.l$ is the IDDewey label of r; (3) $T.s_i$ is the position value of the *first* appearance of l in L_i; (4) $T.e_i$ is the position value of the *last* appearance of l in L_i; and (5) $T.N_i$ is the number of occurrences of k_i in T. Obviously, $T.N_i = T.e_i - T.s_i + 1$. Note that $T.s_i$ and $T.e_i$ denote an interval of L_i, in which all IDDewey labels share the same common prefix, i.e., $T.l$, from which we can get the child list of $T.r$ w.r.t. k_i.

As shown in Algorithm 1, our method firstly initializes the subtree rooted at the root node of the given XML tree in lines 1-4, then calls the procedure processSubTree() to recursively get all ELCA nodes in line 5.

The procedure processSubTree() works as follows. It firstly gets the depth of $T.r$'s child nodes in line 1, then makes each C_i point to the first IDDewey label l in interval $[T.s_i, T.e_i]$, such that l's length is greater than $T.l$ (lines 2-4), which equals making C_i point to the first descendant node of $T.r$ that directly contains k_i. In lines 6-12, it repeatedly gets all child CA nodes of $T.r$. For each child CA node id_{ch} got in line 7, it firstly gets the subtree T' rooted at id_{ch} in line 9; in line 10, it excludes the occurrences of all query keywords in T'. In line 11, it calls the procedure processSubTree() to recursively process T'. After processing all subtrees rooted at $T.r$'s child CA nodes, if $\forall i \in [1, m], T.N_i > 0$, according to Lemma 1, we know that $T.r$ is an ELCA node, then we output it in line 14.

Given a subtree T, the function getNextChildCA() is called in line 7 of processSubTree() to find a child CA node of $T.r$ by intersecting m *child lists* of $T.r$. It always uses the cursor with the maximum chL^{th} ID value as the eliminator (line 1, and $C_x[chL]$ is the eliminator), and uses the static probing order from the shortest list to the longest (lines 1-8). The probe operation will continue to the remaining lists if IDDewey labels of same ID are found (line 6); otherwise, since we found an IDDewey label with ID larger than the current eliminator, the eliminator will be reset and we restart the probing from L_1 immediately (line 7).

Algorithm 1: TDELCA($Q = \{k_1, k_2, ..., k_m\}$)

```
1  T.r ← 1; T.l ← 1
2  for each (i ∈ [1, m]) do
3      T.s_i ← 1; T.e_i ← |L_i|; T.N_i ← |L_i|
4  end for
5  processSubTree(T)
```

Procedure processSubTree(T)

```
1  chL ← |T.l| + 1
2  for each (i ∈ [1, m]) do
3      if (|L_i[T.s_i]| = |T.l|) then C_i ← L_i[T.s_i + 1]
4      else C_i ← L_i[T.s_i]
5  end for
6  while (¬ eof(T)) do
7      id_ch ← getNextChildCA(T, chL)
8      if (id_ch = −1) then break
9      T' ← getSubTree(T, chL, id_ch)
10     T.[N_1, ..., N_m] ← T.[N_1, ..., N_m] − T'.[N_1, ..., N_m]
11     processSubTree(T')
12 end while
13 if (∀i ∈ [1, m], T.N_i > 0) then
14     output T.r as an ELCA node
15 end if
```

Function getNextChildCA(T, chL)

```
1  j ← 1; n ← 1; x ← argmax_i{C_i[chL]}
2  while (n < m) do
3      if (j = x) then j ← j + 1
4      binSearch(T, L_j, chL, C_x[chL])
5      if (pos(C_j) > T.e_j) then return −1
6      if (C_x[chL] = C_j[chL]) then j ← j + 1; n ← n + 1
7      else x ← j; j ← 1; n ← 1
8  end while
9  return C_x[chL]
```

Function getSubTree(T, chL, id_{ch})

```
1  T'.r ← id_ch; T'.l ← T.l.id_ch
2  for each(i ∈ [1, m]) do
3      T'.s_i ← pos(C_i);
4      binSearch(T, L_i, chL, id_ch + 1);
5      T'.e_i ← pos(C_i) − 1; T'.N_i ← T'.e_i − T'.s_i + 1
6  end for
7  return T'
```

Procedure binSearch(T, L_j, chL, id_{ch})

```
1  s ← pos(C_j); e ← T.e_j
2  while (s < e) do
3      mid ← (s+e)/2
4      if (L_j[mid][chL] ≥ id_ch) then e ← mid − 1
5      else s ← mid + 1
6  end while
7  C_j ← L_j[s]
```

Function eof(T)

```
1  if (∃i ∈ [1, m], such that pos(C_i) > T.e_i) then
2      return TRUE
3  end if
4  return FALSE
```

In processSubTree(), after finding a CA node id_{ch} in line 7 by calling getNextChildCA(), we call getSubTree() in line 9 to get the subtree rooted at id_{ch}. Note that binSearch() is used to locate the position of the *first* IDDewey label l in L_j satisfying $l[chL] \geq id_{ch}$. Function eof(T) checks whether we have exhausted a list by checking the cursor positions.

Example 3. *Consider Q_1 and D in Fig. 1.1. Algorithm 1 firstly processes T_1. It finds node 1's child CA nodes from its two child lists, i.e., the second level of the two inverted lists in Fig. 2.1. Since node 1's first child CA node is node 4, we get T_4 and then find node 4's child CA nodes from its child lists in the third level of the two inverted lists. Note that for node 4, $T_4.s_1 = 2$, $T_4.e_1 = 4$, $T_4.s_2 = 2$ and $T_4.e_2 = 5$, that is, we find node 4's child CA nodes from two shortened child lists, i.e., 5,6,10 and 6,6,10,10. As node 6 does not have child CA nodes, and $T_6.N_1 = 1$, $T_6.N_2 = 2$, node 6 is an ELCA node. Similarly, we know that node 10 is an ELCA node. After that, we check the satisfiability of node 4. Since $T_4.N_1 = 3$, $T_4.N_2 = 4$, after excluding the occurrences of all query keywords in T_6 and T_{10}, we have $T_4.N_2 = 0$, and according to Lemma 1, node 4 is not an ELCA node. The following processing is similar, we firstly find the second child CA node of node 1, i.e., node 14, then find node 14's child CA node, i.e., node 16, then the processing stops. Finally, the outputted ELCA nodes by Algorithm 1 for Q_1 on D in Fig. 1.1 are nodes 1, 6, 10 and 16.* □

4.2 Analysis

As each CA node is visited only once, our method does not suffer from the **CAR** problem [24]. Besides, according to Algorithm 1, we have the following result.

Corollary 2. (***Independence***) *The satisfiability of each CA node v w.r.t. ELCA semantics can be determined by v's child nodes.*

The correctness of Corollary 2 can be easily confirmed according to Lemma 1 and Algorithm 1. The importance of Corollary 2 lies in that compared with existing methods [7, 23, 24, 27], our method *avoids* the VUN problem.

Assume that for the given query $Q = \{k_1, k_2, ..., k_m\}$, $0 \leq |L_1| \leq |L_2| \leq ... \leq |L_m|$. Since the length of the child node list of each CA node w.r.t k_i is bounded by $|L_m|$, the cost of checking whether a node is a CA node is $O(m \log |L_m|)$.

If we always process nodes of $S_{ch}^{k_1}(v)$ for each CA node, the number processed nodes of TDELCA is $\sum_{v \in CA(Q)} |S_{ch}^{k_1}(v)|$. As shown by getNextChildCA(), our method always uses the *maximum* ID to probe other lists to find the next CA node, it may skip many useless nodes in $S_{ch}^{k_1}(v)$, thus the total number of processed nodes is bounded by $\sum_{v \in CA(Q)} |S_{ch}^{k_1}(v)|$. Therefore, the time complexity of the TDELCA algorithm is $O(m \cdot (\sum_{v \in CA(Q)} |S_{ch}^{k_1}(v)|) \cdot \log |L_m|)$. Note that in any case, $\sum_{v \in CA(Q)} |S_{ch}^{k_1}(v)| < \sum_{l \in L_1} |l| \leq d \cdot |L_1|$, where d is the depth of the given XML tree.

We also note that any set intersection algorithm [6, 19] can be used in our method for CA computation, and any of the search methods (e.g., binary, galloping [2], or interpolation search [1]) can be used to complete the set intersection in our getNextChildCA() function.

5 The LList-based Algorithm

Given a keyword query Q, since any node may repeatedly appear many times in each inverted list, for each processed node v, Algorithm 1 needs to use two probe operations (binSearch() is called two times) to find the first and last IDDewey labels containing v in each L_i, such that if v is a CA node, we can recursively get its child CA nodes.

Moreover, as shown in Fig. 2.1, the closer v to the root node, the more repeated times for v in each inverted list on average, thus the longer the child lists of v. Since we perform set intersection operation on the set of child lists of v, one of the most important factors that affects its performance is the length of the search interval. The longer the child lists of v, the more cost we need to pay to process each of its child nodes.

Considering the above problems, we propose a new index, namely LList, based on which we can reduce both the cost and calling times of binSearch().

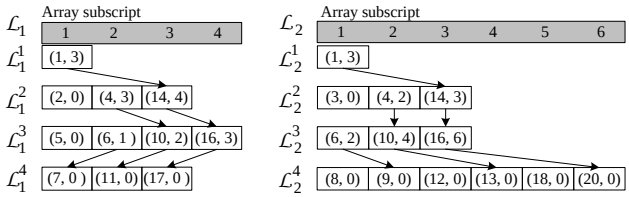

Figure 5.1: LLists of "Tom" ($\mathcal{L}_1$) and "XML" ($\mathcal{L}_2$).

5.1 The LList Index

Let L_i be the inverted IDDewey label list of k_i, in our method, the corresponding inverted list $\mathcal{L}_i$ of k_i consists of at most d_i sublists, i.e., $\mathcal{L}_i = \{\mathcal{L}_i^1, \mathcal{L}_i^2, ..., \mathcal{L}_i^{d_i}\}$, where d_i denotes the number of components of the *longest* IDDewey label in L_i. Each $\mathcal{L}_i^j (j \in [1, d_i])$ consists of entries representing the set of distinct nodes in the j^{th} level of L_i. Each entry $e_v \in \mathcal{L}_i^j$ corresponding to a node v consists of two numeric values:

- "id" is the last component of v's IDDewey label, i.e., the ID value of v,
- "p_{lc}" is, in $\mathcal{L}_i^{j+1}$, the position of the entry corresponding to v's *last* child that contains k_i. When $S_{ch}^{k_i}(v) = \emptyset$, if e_v is the *first* entry of $\mathcal{L}_i^j$, $e_v.p_{lc} = 0$; otherwise, $e_v.p_{lc} = e_u.p_{lc}$, where e_u is the last entry that precedes e_v in $\mathcal{L}_i^j$.

We call each $\mathcal{L}_i$ an LList, denoting that all nodes in the same *level* are maintained together. Fig. 5.1 shows the two LLists of "Tom" and "XML" based on D in Fig. 1.1. For simplicity, we do not differentiate a node and its entry in the following discussion, if without ambiguity. We use $|\mathcal{L}_i^j|$ to denote the number of entries in $\mathcal{L}_i^j$, $|\mathcal{L}_i|$ the sum of lengths of all sublists of $\mathcal{L}_i$, $\mathcal{L}_i^j[x] (x \in [1, |\mathcal{L}_i^j|])$ the x^{th} entry of $\mathcal{L}_i^j$. Take $\mathcal{L}_1$ for example, "(1,3)" of $\mathcal{L}_1^1$ means that the child list of node 1 contains the first three nodes of $\mathcal{L}_1^2$, and "(14,4)" of $\mathcal{L}_1^2$ means that the *last* node in the child list of node 14 is the 4^{th} node of $\mathcal{L}_1^3$, i.e., node 16 or $\mathcal{L}_1^3[4]$. As "(4,3)" is the last entry that proceeds "(14,4)" in $\mathcal{L}_1^2$, we know that the child list of node 14 w.r.t. "Tom" contains just one node, i.e., node 16.

Compared with the inverted list L_i of IDDewey labels (Fig. 2.1), each distinct node of L_i is maintained only once in $\mathcal{L}_i$. Compared with IDList L_i^{ID} that also consists of distinct nodes [24] of L_i, all nodes of $\mathcal{L}_i$ are maintained by level, and each entry of $\mathcal{L}_i$ has only two numerical values, while each entry of L_i^{ID} consists of three numerical values. LList can be efficiently constructed when parsing the XML document in document order.

Property 1. *For each node of $\mathcal{L}_i$, all nodes in each of its child lists are sorted in ascending order by their ID values.*

Property 1 guarantees that for any node v, $S_{ch}^{CA}(v)$ can still be computed based on anyone of existing *ordered set intersection algorithms*. Based on LList, we have the following lemma to check whether a given CA node v is an ELCA node.

Lemma 2. *For a query $Q = \{k_1, k_2, ..., k_m\}$ and CA node v, let $T_v.N_i$ be the number of nodes in the child list of v w.r.t. k_i in $\mathcal{L}_i$, v is an ELCA node iff $S_{ch}^{CA}(v) = \emptyset$ or $\forall i \in [1, m], T_v.N_i - |S_{ch}^{CA}(v)| > 0$.*

Proof. As v is a CA node, T_v contains at least one occurrence of each keyword of Q. Therefore, if $S_{ch}^{CA}(v) = \emptyset$, according to Definition 4, v is an ELCA node. Consider the case where $S_{ch}^{CA}(v) \neq \emptyset$. As all nodes in each child list of v have different ID values, $\forall i \in [1, m], T_v.N_i - |S_{ch}^{CA}(v)|$ equals removing subtrees that are rooted at v's child CA (or descendant LCA) nodes. Therefore, if $\forall i \in [1, m], T_v.N_i - |S_{ch}^{CA}(v)| > 0$, it means after removing all subtrees that are rooted at v's child CA (or descendant LCA) nodes, T_v still contains each keyword of Q at least once. According to Definition 4 and Lemma 1, v is an ELCA node. □

Example 4. *Consider Q_1 and D in Fig. 1.1. From Fig. 5.1 we know that $T_1.N_1 = T_1.N_2 = 3$. Since $|S_{ch}^{CA}(1)| = |\{4, 14\}| < 3$, according to Lemma 2, node 1 is an ELCA node. For node 4, since $T_4.N_1 = 3$, $T_4.N_2 = 2$ and $|S_{ch}^{CA}(4)| = |\{6, 10\}| = 2$, we know $T_4.N_2 - |S_{ch}^{CA}(4)| = 0$, thus according to Lemma 2, node 4 is not an ELCA node.* □

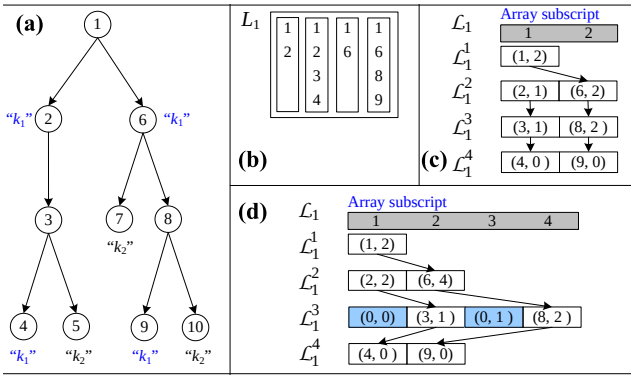

Figure 5.2: Illustration of the problem of non-leaf keyword nodes. (a) is a sample XML document; (b) is the inverted list L_1 of IDDewey labels for k_1, (c) is the corresponding LList $\mathcal{L}_1$, which cannot process the case where keyword nodes are non-leaf nodes; (d) is the revised LList $\mathcal{L}_1$, which can correctly process non-leaf keyword nodes.

5.2 The Algorithm

We call the algorithm that is based on the LList index as TDELCA-L, which finds all CA nodes and checks their satisfiability in the same order as that of TDELCA. The improvements of TDELCA-L over TDELCA lie in the following aspects: (1) binSearch() does not need to be called by getSubTree() anymore; (2) binSearch() is performed on a much shorter child list on average; (3) binSearch() stops immediately when it spots a value that equals the eliminator. Note that in TDELCA, each node may repeatedly appear in a child list many times, and binSearch() is used to find the *first* IDDewey label containing the *eliminator*. We omit the detailed description of TDELCA-L due to limited space.

Example 5. *Consider Q_1 and D in Fig. 1.1. To get child CAs of node 1, we need to compute the intersection of node 1's child lists, i.e.,* $\boxed{2,4,14}$ *and* $\boxed{3,4,14}$ *in $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively. As a comparison, the corresponding child lists are* $\boxed{2,4,4,4,14}$ *and* $\boxed{3,4,4,4,4,14,14}$ *in L_1 and L_2 for TDELCA. Obviously, the intersection operation is performed on much shorter lists by calling binSearch(). Moreover, as each node in a child list of LList is distinct from others, binSearch() can stop immediately when it spots a value that equals the eliminator. Further, in TDELCA, we need to call binSearch() in getSubTree() to get the position of the* last *IDDewey label containing the ID of the current node v, such that to get the child list of v w.r.t. each query keyword. As a comparison, we do not need to call binSearch() to know each child list, which can be directly got by p_{lc}'s value in the corresponding entry of v.* □

As Lemma 2 is still based on the set of child nodes to check the satisfiability of each CA node, Corollary 1 and 2 still hold for the TDELCA-L algorithm.

Let N be the number of nodes in the longest child list for all nodes in the set of LLists of $Q = \{k_1, k_2, ..., k_m\}$, obviously $N \leq |L_m|$. As the number of processed nodes of TDELCA-L equals that of TDELCA, the time complexity of TDELCA-L is $O(m \cdot (\sum_{v \in CA(Q)} |S_{ch}^{k_1}(v)|) \cdot \log N)$.

5.3 Processing Non-leaf Keyword Nodes

Even though each IDDewey label in L_i implicitly contains all its ancestor nodes, only the last component represents the node that *directly* contains k_i, and directly removing all repeated appearances of each node of L_i to construct $\mathcal{L}_i$ may result in losing the information that some *non-leaf* nodes may *directly* contain k_i, since in $\mathcal{L}_i$, we are only sure that leaf nodes are keyword nodes w.r.t. k_i. E.g., for query $Q = \{k_1, k_2\}$ and the XML document in Fig. 5.2 (a), the inverted list L_1 of IDDewey labels for k_1 is shown in Fig. 5.2 (b), from which we know that k_1 appears four times in the two subtrees rooted at nodes 2 and 6. However, according to LList $\mathcal{L}_1$ in Fig. 5.2 (c), both nodes 2 and 6 have only one child node that contains k_1 in their subtrees. Therefore, according to Lemma 2, node 6 is not an ELCA node. In fact, node 6 is an ELCA node.

A straight-forward way to solve the above problem is using a flag in each entry to denote whether it directly contains a keyword. However, this method will make LList suffer from the same space cost as IDList [24, 25] does. Different with the above method, only when *both* $v \in \mathcal{L}_i^j$ and some of its descendant nodes directly

contain k_i, we maintain a *dummy* entry e_v^d in $\mathcal{L}_i^{j+1}$ with $e_v^d.id = 0$ denoting that v directly contains k_i. e_v^d is inserted before e_v's first child entry.

Let $e_{u_{ch}}$ be the last entry that precedes e_v's first child entry, the value of e_v^d can be got by Formula 5.1.

$$e_v^d = \begin{cases} (0,0), & \nexists e_{u_{ch}} \\ (0, e_{u_{ch}}.p_{lc}), & \text{otherwise.} \end{cases} \tag{5.1}$$

According to Formula 5.1, as shown in Fig. 5.2 (d), we add a dummy node "(0,0)" for node 2 in $\mathcal{L}_1^3$, and add a dummy node "(0,1)" for node 6 in $\mathcal{L}_1^3$. Note that for a *non-leaf* node v, its dummy node e_v^d is in the first position of its child list. Therefore, Corollary 2, Lemma 2 and Property 1 still hold. And TDELCA-L still works correctly, if we do not take the dummy node as anyone's child node when using Lemma 2 to check CA nodes' satisfiability.

6 Extending to Other Semantics

6.1 SLCA Computation

To get SLCA results, we can still use our top-down processing strategy to get all CA nodes, then check the satisfiability of each CA node v according to Lemma 3.

Lemma 3. *Let v be a CA node of Q, then v is an SLCA node of Q iff $S_{ch}^{CA}(v) = \emptyset$.*

Proof. Since $S_{ch}^{CA}(v) = \emptyset$ is equivalent to v does not have descendant LCA nodes, according to Definition 3, v is an SLCA node of Q iff $S_{ch}^{CA}(v) = \emptyset$. □

6.2 LCA Computation

Our top-down processing strategy can also be used to get LCA results. If the underlying index is inverted lists of Dewey labels or one of its variants, then we can check the satisfiability of each CA node v according to Lemma 4.

Lemma 4. *For a query $Q = \{k_1, k_2, ..., k_m\}$ and CA node v, let $T_v.N_i$ be the number of occurrences of k_i in T_v, v is an LCA node iff v does **not** satisfy $S_{ch}^{CA}(v) = \{u\} \wedge (\forall i \in [1,m], T_v.N_i = T_u.N_i)$.*

Proof. $S_{ch}^{CA}(v) = \{u\} \wedge (\forall i \in [1,m], T_v.N_i = T_u.N_i)$ means that v has just one child CA node u, and all occurrences of each keyword of Q are in T_u. Therefore, if this condition holds, it means v is not an LCA node; otherwise, it means we can find a set of nodes from T_v that collectively contain all keywords of Q, such that their LCA is v, and in this case, v does not satisfy $S_{ch}^{CA}(v) = \{u\} \wedge (\forall i \in [1,m], T_v.N_i = T_u.N_i)$. □

If the underlying index is LList, we can check the satisfiability of each CA node v according to Lemma 5.

Lemma 5. *Let v be a CA node of $Q = \{k_1, k_2, ..., k_m\}$, then v is an LCA node of Q iff $S_{ch}^{CA}(v) = \emptyset$ or $\exists i \in [1,m]$, such that $|S_{ch}^{k_i}(v)| > 1$.*

Proof. If $S_{ch}^{CA}(v) = \emptyset$, it means that v is an SLCA node, therefore an LCA node. If $\exists i \in [1,m]$, such that $|S_{ch}^{k_i}(v)| > 1$, it means that v has at least two different child nodes containing some keyword of Q in their subtrees. And in this case, we can always find a set of nodes from different subtrees rooted at v's child nodes, such that they collectively contain all keywords of Q and v is their LCA node. On the contrary, if v is an LCA node, then either v does not have child CA nodes, i.e., $S_{ch}^{CA}(v) = \emptyset$, or v must have at least two different child nodes that contain some keyword of Q, which is equal to $\exists i \in [1,m]$, such that $|S_{ch}^{k_i}(v)| > 1$. Therefore, v is an LCA node of Q iff $S_{ch}^{CA}(v) = \emptyset$ or $\exists i \in [1,m]$, such that $|S_{ch}^{k_i}(v)| > 1$. □

For both SLCA and LCA, the time complexity of each one based on our top-down processing strategy is same as that of our algorithms introduced in previous sections in terms of the underlying index, and Corollary 1 and 2 still hold for all algorithms.

(a) H_F

Key:	k_1	k_2	...
Value:	11	13	...

(b) H_1

Key:	1	2	4	5	6	7	10	11	14	16	17
Value:	3	0	3	0	1	0	1	0	1	1	0

(c) H_2

Key:	1	3	4	6	8	9	10	12	13	14	16	18	20
Value:	3	0	2	2	0	0	2	0	0	1	2	0	0

Figure 7.1: Illustration of the hash tables used in our methods corresponding to the XML document D in Fig. 1.1.

6.3 Other Semantics

Besides LCA, ELCA and SLCA, researchers have proposed several other query semantics, including XSEarch [5], MLCA [13] and VLCA [12], all are based on LCA semantics.

The common feature of XSEarch, MLCA and VLCA is that their validation criteria need to check the names of all nodes on the path from an LCA node to each of its descendant nodes that directly contain some query keywords. Therefore, in their methods, if additional indexes that are used to get node names are constructed in advance, our algorithms can also easily support these semantics with minor adaption. We omit the detailed discussion in this paper, since no other technical problems need to be solved.

7 The Hash Search Based Algorithms

Even though TDELCA-L reduces the time complexity compared with TDELCA, it relies on the probe operation (implemented by binary search) to align the cursors of inverted lists. To further improve the overall performance, we consider the existence of *additional* hash indexes [20, 25, 27] on inverted lists, such that each probe operation takes $O(1)$ time.

As shown in Fig. 7.1, the first hash table H_F records the number of nodes in each $\mathcal{L}_i$, which is used to choose the *shortest* LList. For each $\mathcal{L}_i$, another hash table H_i records, for each *valid* node of $\mathcal{L}_i$ (*dummy* nodes are not taken as valid nodes), the number of its *child* nodes (*including* the dummy node if exists) that contain k_i. Note that H_i in our methods is different with that of [20, 25, 27], where H_i records, for each node v, the number of v's *descendant* nodes that *directly* contain k_i in T_v.

According to Fig. 7.1 (a), we know that the number of nodes of $\mathcal{L}_1$ is 11, which can be denoted as $H_F[k_1] = 11$. According to Fig. 7.1 (b), node 1 has three child nodes containing k_1("Tom"), which can be denoted as $H_1[1] = 3$. Node 5 does not have *child* nodes containing k_1, thus $H_1[5] = 0$. Similarly, node 3 does not contain k_1, which is denoted as $3 \notin H_1$.

7.1 The Baseline Hash Search Algorithm

Assume that $|\mathcal{L}_1| \leq |\mathcal{L}_2| \leq ... \leq |\mathcal{L}_m|$, the main idea of our baseline hash search algorithm is: *take the shortest LList $\mathcal{L}_1$ as the working list and recursively process all CA nodes in top-down way. For each CA node v, sequentially check whether each of its child nodes in $\mathcal{L}_1$ is a CA node, then output v if it is an ELCA result.*

Algorithm 2 shows the detailed description of the TDELCA-H algorithm. Compared with TDELCA and TDELCA-L, for a given query Q, TDELCA-H only needs to process all CA nodes and their child nodes in $\mathcal{L}_1$. For each processed node v in $\mathcal{L}_1$, TDELCA-H checks whether v is a CA node by hash probe operations, rather than set intersection operations on a set of child lists.

Example 6. *Consider Q_1 and D in Fig. 1.1 again. According to Fig. 7.1 (a), $H_F[Tom] = 11 < H_F[XML] = 13$, our method processes nodes of $\mathcal{L}_1$ to get all ELCA nodes. Take node 4 for example, since $4 \in H_2$, we know that node 4 is a CA node. We then recursively process T_4. As the first child node of node 4 in $\mathcal{L}_1^3$ is node 5, and $5 \notin H_{k_2}$, we know that node 5 is not a CA node. Similarly, we know that nodes 6 and 10 are child CAs of node 4. As both nodes 6 and 10 do not have child CA nodes, they are ELCA nodes. After processing all child nodes of node 4, we know that node 4 is not an ELCA node, since $T_4.N_2 - |S_{ch}^{CA}(3)| = 0$. After processing all CA nodes and their child nodes in $\mathcal{L}_1$, we get all ELCA results, i.e., nodes 1, 6, 10 and 16.* □

Algorithm 2: TDELCA-H($Q = \{k_1, k_2, ..., k_m\}$)

1 $T.r \leftarrow 1; T.l \leftarrow 1; T.s_1 \leftarrow 1; T.e_1 \leftarrow |\mathcal{L}_1^2|; T.N_1 \leftarrow |\mathcal{L}_1^2|$
2 **for each** $(i \in [1, m], i \neq j)$ **do**
3 $T'.N_i \leftarrow H_{k_i}[1]$
4 **end for**
5 processSubTree(T)

Procedure processSubTree(T)

1 $chL \leftarrow |T.l| + 1$; $N_{chCA} \leftarrow 0$
2 **if** $(\mathcal{L}_1^{chL}[T.s_1].id = 0)$ **then** $C_1^{chL} \leftarrow \mathcal{L}_1^{chL}[T.s_1 + 1]$
3 **else** $C_1^{chL} \leftarrow \mathcal{L}_1^{chL}[T.s_1]$
4 **while** $(\text{pos}(C_1^{chL}) \leq T.e_1)$ **do**
5 $id_{ch} \leftarrow C_1^{chL}.id$
6 $bCA \leftarrow \text{isCA}(id_{ch}, 1)$
7 **if** $(bCA = \text{TRUE})$ **then**
8 $N_{chCA} \leftarrow N_{chCA} + 1$
9 $T'.r \leftarrow id_{ch}; T'.l \leftarrow \underline{T.l}.id_{ch}$
10 getChildList($T', \mathcal{L}_1^{chL}, C_1^{chL}$)
11 processSubTree(T')
12 **end if**
13 advance(C_1^{chL})
14 **end while**
15 **if** $(N_{chCA} = 0$ or $\forall i \in [1, m], T.N_i - N_{chCA} > 0)$ **then**
16 output $T.r$ as an ELCA node
17 **end if**

Function isCA(id, j)

1 **for each** $(i \in [1, m], i \neq j)$ **do**
2 **if** $(id \notin H_{k_i})$ **then return** FALSE
3 $T'.N_i \leftarrow H_{k_i}[id]$
4 **end for**
5 **return** TRUE

Procedure getChildList($T', \mathcal{L}_1^{chL}, C_1^{chL}$)

1 $T'.e_1 \leftarrow C_1^{chL}.p_{lc}$
2 **if** $(\text{pos}(C_1^{chL}) = 1)$ **then**
3 $T'.s_1 \leftarrow 1; T'.N_1 \leftarrow T'.e_1$; **return**
4 **end if**
5 $v \leftarrow \mathcal{L}_1^{chL}[\text{pos}(C_1^{chL}) - 1]$
6 **if** $(C_1^{chL}.p_{lc} = v.p_{lc})$ **then** $T'.s_1 \leftarrow T'.e_1 + 1$
7 **else** $T'.s_1 \leftarrow v.p_{lc} + 1$
8 $T'.N_1 \leftarrow C_1^{chL}.p_{lc} - v.p_{lc}$

Note that since H_i uses node IDs as the key, Corollary 1 does not hold for the TDELCA-H algorithm, that is, the LList can only be constructed based on IDDewey labeling scheme for the TDELCA-H algorithm.

As the number of processed nodes of TDELCA-H is same as that of TDELCA and TDELCA-L, the time complexity of TDELCA-H is $O(m \cdot \sum_{v \in CA(Q)} |S_{ch}^{k_1}(v)|)$.

Note that we can easily extend TDELCA-H to LCA and SLCA semantics based on Lemmas 3 and 5, which we call as TDLCA-H and TDSLCA-H, respectively.

7.2 The Optimized Algorithm for SLCA Semantics

Even though the TDELCA-H algorithm removes the log factor from its time complexity, according to Algorithm 2, TDELCA-H needs to process all CA nodes and their child nodes in $\mathcal{L}_1$ with hash probe operations. As the performance of TDELCA-H is dominated by the number of hash probe operations, the overall performance can be improved if we can save as many hash probe operations as possible.

Unfortunately, it looks impossible to save hash probe operations for ELCA and LCA semantics, since the hash value for each node v in H_i is the number of its child nodes in $\mathcal{L}_i$, which is used to test the satisfiability of v according to Lemmas 2 and 5. However, according to Lemma 3, the hash value for each node in Fig. 7.1 (b)

(a) H_1

Key:	1	2	4	5	6	7	10	11	14	16	17
Value:	2	∞	5	∞	7	∞	11	∞	16	17	∞

(b) H_2

Key:	1	3	4	6	8	9	10	12	13	14	16	18	20
Value:	3	∞	6	8	∞	∞	12	∞	∞	16	18	∞	∞

Figure 7.2: Illustration of the second kind of hash tables used for SLCA computation. H_1 and H_2 are the two hash tables corresponding to the two LLists of "Tom(k_1)" and "XML(k_2)". H_i records for each node v of $\mathcal{L}_i$, the ID value of v's first child node (excluding dummy node), where "∞" means that the corresponding node does not have child node.

and (c) is useless for SLCA computation, since in this case, for each processed node v in $\mathcal{L}_1$, we only need to know whether v appears in hash tables of other LLists.

Considering this problem, we propose the second hash indexes to accelerate SLCA computation. As shown in Fig. 7.2, we maintain in H_i, for each *valid* node v of $\mathcal{L}_i$, the ID value of its *first* child node (dummy node is *not* taken as v's child node). If v does not have child nodes in $\mathcal{L}_i$, then its hash value in H_i is ∞. Note that for SLCA computation, the H_F index is same as that of Fig. 7.1 (a). Based on the new hash indexes, when processing a node v of $\mathcal{L}_1$, we have the chance to avoid using hash probe operations if v is not a CA node, as shown by Lemma 6.

Lemma 6. *Let u be a CA node of $Q = \{k_1, k_2, ..., k_m\}$, $id_u^{max} = \max\{id_{v_i} | v_i$ is the first child node of u in $\mathcal{L}_i, i \in [2, m]\}$. For each node $v \in S_{ch}^{k_1}(u)$, if $id_v < id_u^{max}$, then v is not a CA node.*

Proof. Suppose v is a CA node, then v must appear in each LList as one of u's child node. Since v_i is u's first child node in $\mathcal{L}_i (i \in [2, m])$, we know that $id_v \geq id_u^{max}$. Therefore, if $id_v < id_u^{max}$, v is not a CA node. □

Based on Lemma 6 and the new hash indexes in Fig. 7.2, the algorithm for SLCA computation can be easily devised. Compared with Algorithm 2, we only need to get the max hash value for each CA node u in isCA() function, and take it as u's id_u^{max} value. When processing u's child nodes in $\mathcal{L}_1$, we can then safely skip those ones with ID values less than u's id_u^{max} value. We call this algorithm as TDSLCA-HO, which has the same time complexity as TDxLCA-H, but usually issues less number of hash probe operations in practice.

Example 7. *Consider Q_1 and D in Fig. 1.1 again. As $H_F[Tom] = 11 < H_F[XML] = 13$, our method processes nodes of $\mathcal{L}_1$ to get all SLCA nodes. Take node 1 as an example, since $S_{ch}^{k_1}(1) = \{2, 4, 14\}$ and $id_1^{max} = 3$, according to Lemma 6, we do not need to check whether node 2 appears in H_2 of Fig. 7.2 using hash probe operation as TDxLCA-H does. Similarly, for node 4, since $S_{ch}^{k_1}(4) = \{5, 6, 10\}$ and $id_4^{max} = 6$, we do not need to process node 5 with hash probe operation. For the same reason, node 7, 11 and 17 also do not need to be processed with hash probe operations. As a comparison, to process Q_1 on D in Fig. 1.1, TDxLCA-H needs 11 hash probe operations, while TDSLCA-HO only needs 6 hash probe operations.* □

8 Experiment

8.1 Experimental Setup

All experiments were run on a PC with AMD Athlon^{TM}II X2 270 3.4GHz CPU, 2GB memory, 500GB IDE hard disk, and Windows XP Professional OS.

We considered three groups of algorithms:

(Group 1) Algorithms of this group target at **ELCA** computation, which consist of two sub-groups: (Group 1.1) algorithms that are not based on hash search, including BwdELCA [24, 25], TDELCA and TDELCA-L; (Group 1.2) algorithms that are based on hash search, including HybELCA-HS [25] and TDELCA-H.

(Group 2) Algorithms of this group target at **SLCA** computation, which also consist of two sub-groups: (Group 2.1) algorithms that are not based on hash search, including BwdSLCA+ [24, 25], TDSLCA

Table 8.1: Queries on XMark dataset. $\sum_{i=1}^{m} |L_i|$ denotes the sum of the lengths of all inverted lists of Dewey labels, $|L_m|(|L_1|)$ denotes the length of the *longest* (*shortest*) inverted list of Dewey labels, $|L_m^{ID}|(|L_1^{ID}|)$ denotes the length of the *longest* (*shortest*) IDList, N_{L_1} denotes the number of integers for all Dewey labels of L_1, $N_{total} = \sum_{v \in CA(Q)} |S_{ch}^{k_1}(v)|$ is the upper bound of the number of processed nodes by our methods, N is the number of nodes in the longest child list (see Section 5.2), N_E is the number of ELCA results, $R_E = N_E/|L_1|$ denotes the result selectivity, and *Freq* denotes the category of the keywords according to the length of the corresponding inverted lists of Dewey labels: (1) low frequency (100-1,000), (2) median frequency (10,000-40,000), (3) high frequency (300,000-600,000).

Query	Keywords	$\sum_{i=1}^{m} \|L_i\|$	$\|L_m\|$	$\|L_m^{ID}\|$	$\|L_1\|$	$\|L_1^{ID}\|$	N_{L_1}	N_{total}	N	N_E	$R_E(\%)$	Freq.
QX1	villages,hooks	1,290	829	4,300	461	2,439	3,634	471	211	9	1.95	Low
QX2	baboon,patients,arizona	1,575	742	3,689	382	1,355	2,943	1	4	1	0.26	
QX3	cabbage,tissue,shocks,baboon	2,088	742	3,689	366	1,809	2,754	376	178	9	2.46	
QX4	shocks,necklace,cognition,cabbage,tissue	2,041	596	2,996	200	1,091	1,597	210	149	9	4.5	
QX5	female,order	36,594	19,894	68,140	16,700	62,165	109,835	17,980	5,560	579	3.47	Med
QX6	privacy,check,male	85,960	36,300	101,095	18,428	60,598	98,832	2,354	30,260	34	0.185	
QX7	takano,province,school,gender	108,187	34,061	106,786	17,129	53,324	100,338	16,291	31,981	108	0.631	
QX8	school,gender,education,takano,province	143,444	35,257	113,228	17,129	53,324	100,338	16,291	31,981	108	0.631	
QX9	bold,increase	674,824	370,118	1,265,885	304,706	679,688	1,538,902	249,844	54,322	34,189	11.22	High
QX10	date,listitem,emph	1,112,760	457,231	1,229,698	304,969	574,800	2,353,169	126,507	54,294	43,792	14.36	
QX11	incategory,text,bidder,date	1,696,631	528,807	1,662,791	299,018	353,200	1,196,072	1	5	1	0.0003	
QX12	bidder,date,keyword,incategory,text	2,048,752	528,807	1,662,791	299,018	353,200	1,196,072	1	5	1	0.0003	
QX13	takano,province	50,649	33,520	104,659	17,129	53,324	100,338	18,159	31,662	1,810	10.567	Random
QX14	cabbage,male,female	38,688	19,894	68,140	366	1,809	2,754	376	1,118	9	2.459	
QX15	male,female,keyword,incategory	802,018	411,575	1,236,016	18,428	60,598	98,832	1,394	50,000	75	0.407	
QX16	female,keyword,incategory,cabbage,male	802,384	411,575	1,236,016	366	1,809	2,754	234	50,000	6	1.639	

and TDSLCA-L[1]; (Group 2.2) algorithms that are based on hash search, including HybSLCA-HS [25], TDSLCA-H and TDSLCA-HO.

(Group 3) Algorithms of this group target at **LCA** computation, including ALCA [22], TDLCA, TDLCA-L and TDLCA-H.

Note that as we have made detailed performance comparison between existing methods in [25], we refer readers to [25] for more details, here we only make comparison with the *best one* of each kind of existing algorithms, that is, for ELCA computation, the compared algorithms include BwdELCA (without hash indexes) and HybELCA-HS (with hash indexes); for SLCA computation, the compared algorithms include BwdSLCA$^+$ (without hash indexes) and HybSLCA-HS (with hash indexes); and for LCA computation, the compared algorithm is the ALCA algorithm.

All algorithms were implemented using Microsoft VC++. To make a fair comparison, we use the same configuration with that of [24, 25], i.e., (1) use the same datasets, including XMark (582MB)[2] and DBLP (876MB)[3]. After parsing the two datasets, Oracle Berkeley DB[4] is used to store the keyword inverted lists by a hash file, where each key is a keyword k and the value associated with k is the inverted list of k; (2) test the same set of queries on the two datasets (see Table 1 and 3); (3) evaluate the performance of all algorithms on main-memory resident data. When processing a given query Q, the set of inverted lists are firstly loaded into memory, and the running time is the averaged one over 1000 runs with warm cache.

Further, as SLCA nodes are usually fewer and deeper than ELCA and LCA, and all our algorithms need to traverse all CA nodes in top-down way, we also tested 10 queries (Table 8.4) on Treebank dataset (84MB)[5] to show the performance difference of these algorithms when the document tree becomes deeper. The maximum/average document depths of XMark, DBLP and Treebank are 12/5, 6/2.9 and 36/7.8, respectively.

8.2 Evaluation Metrics

For algorithms that are based on *set intersection* operation, such as BwdSLCA$^+$, BwdELCA, TDxLCA and TDxLCA-L, the evaluation metrics include: (1) running time, (2) the number of comparison operations between integers, which helps us understand the performance variance in an in-depth way. The total number of comparisons, i.e., N_C, can be computed as $N_C = \sum_{i=1}^{M} n_i$, where n_i is the number of search steps of a binary search operation, and M is the number of binary search operations.

For algorithms that are based on *hash probe* operation, such as HybSLCA-HS, HybELCA-HS, TDxLCA-H and TDSLCA-HO, their performance is mainly affected by the number of hash probe operations. Thus the evaluation metrics for these algorithms: (1) running time; (2) the number of hash probe operations.

For the ALCA algorithm that is based on $OP1$ and $OP2$ operations, the evaluation metrics include: (1) running time, (2) the number of comparison operations between integers, i.e., N_C. N_C can also be computed as $N_C = \sum_{i=1}^{M} n_i$, where n_i is the number of compared integer pairs to perform either $OP1$ or $OP2$ once, M the number of $OP1$ and $OP2$ operations.

8.3 ELCA Computation

Comparison of Algorithms in Group 1

Fig. 8.1 shows the normalized time[6] of all algorithms in Group 1 on queries QX1 to QX16, from which we have two observations for algorithms of ***Group 1.1***: (1) TDELCA and TDELCA-L are more efficient than BwdELCA. The reason is that TDELCA and TDELCA-L avoid both the CAR and VUN problems, thus achieve significant performance gains; (2) TDELCA-L usually works better than TDELCA, because TDELCA-L reduces both the cost and the number of binary search operations.

The above results can be further verified according to the number of comparison operations shown in columns 2-4 of Table 8.2, from which we know that TDELCA-L always needs the least comparison operations.

[1] TDSLCA and TDSLCA-L correspond to TDELCA and TDELCA-L, respectively, by replacing the checking condition from Lemma 1 and 2 to Lemma 3. For the same reason, we have two algorithms on LCA computation, namely TDLCA and TDLCA-L.

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

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

[4] http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html

[5] http://www.cs.washington.edu/research/xmldatasets/www/repository.html

[6] Normalized time of each algorithm is defined as $(t1 \times 100)/t2$, where t_1 is the running time of this algorithm, t_2 is the running time of the first algorithm in Figs. 8.1-8.2, 8.4-8.8.

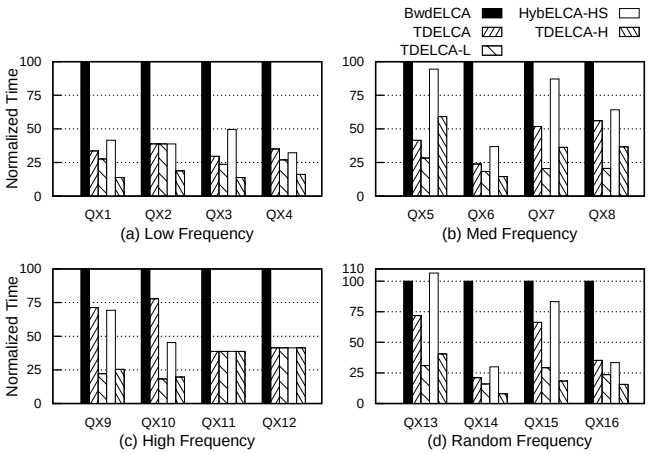

Figure 8.1: Running time of ELCA computation on XMark

Table 8.2: Number of comparison operations (for BwdELCA, TDELCA and TDELCA-L) and hash probe operations (for HybELCA-HS and TDELCA-H).

Query	BwdELCA	TDELCA	TDELCA-L	HybELCA-HS	TDELCA-H
QX1	7,646	3,601	3,367	522	471
QX2	50	33	3	7	1
QX3	5,425	2,665	2,179	840	396
QX4	3,904	2,001	1,416	296	240
QX5	97,939	69,866	55,017	19,670	17,753
QX6	38,147	18,977	17,426	2,707	2,651
QX7	224,593	140,456	133,149	34,650	18,317
QX8	226,308	145,195	135,808	18,864	18,435
QX9	806,274	1,857,448	608,443	293,311	241,253
QX10	2,226,710	4,453,949	1,508,678	246,404	185,599
QX11	53	37	1	7	1
QX12	77	38	1	15	3
QX13	178,907	185,901	135,255	36,358	18,159
QX14	8,335	4,397	3,915	829	386
QX15	26,496	18,918	14,396	3,085	1,671
QX16	5,193	3,233	2,460	337	306

We also know that TDELCA may need more comparison operations than BwdELCA for some queries, e.g., QX9, QX10 and QX13, but it still works better than BwdELCA (see Fig. 8.1), this is because the binary search of TDELCA is done on a much shorter search interval than that of BwdELCA on average, which means that the buffer hit ratio of TDELCA is higher than that of BwdELCA, thus can be done more efficiently. As a comparison, the cost of processing each node by TDELCA (TDELCA-L) is $O(m \cdot \log |L_m|)$ ($O(m \cdot \log N)$), while the cost of processing each node by BwdELCA is $O(m \cdot \log |L_m^{ID}|)$ (see Table 8.1 to get $|L_m|$, N and $|L_m^{ID}|$ for each query).

For algorithms of ***Group 1.2***, from Fig. 8.1 we know that TDELCA-H works much better than HybELCA-HS on most queries, which can also be verified according to the number of hash probe operations shown in columns 5-6 of Table 8.2. The reason that TDELCA-H uses less hash probe operations lies in that TDELCA-H can avoid the VUN problem, while HybELCA-HS may waste hash probe operations on useless nodes.

By comparing all algorithms of ***Group 1***, we know that TDELCA and TDELCA-L (TDELCA-H) work better than BwdELCA (HybELCA-HS) by avoiding both the CAR and VUN problems. For our methods, TDELCA-L and TDELCA-H beat TDELCA for all queries. Note that TDELCA-H may not be as efficient as TDELCA-L for some queries, because TDELCA-L can utilize nodes in other LLists to skip more nodes, while TDELCA-H can only utilize the positional relationship between nodes of $\mathcal{L}_1$ to achieve pruning.

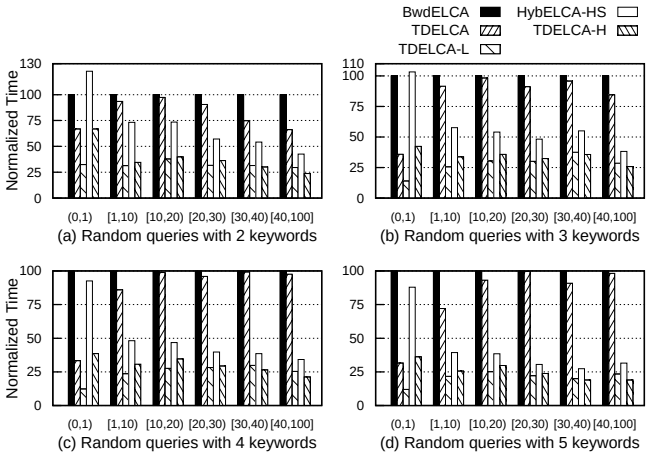

Figure 8.2: Running time with different result selectivities.

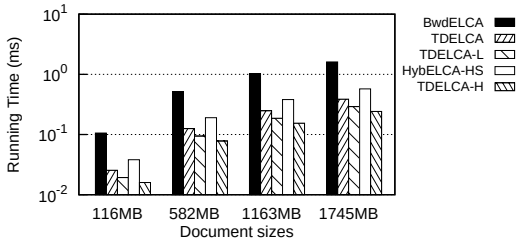

Figure 8.3: Running time of QX6 on XML documents with different sizes (log-scaled).

Scalability

Besides the 16 queries in Table 8.1, we generated 174,406 queries by combining the set of keywords we selected from XMark dataset. Fig. 8.2 shows the impacts of result selectivity on the overall performance of these algorithms grouped into different selectivity ranges. The *result selectivity* of a query Q is defined as the number of results over the length of the shortest inverted Dewey label list for all algorithms.

We investigate the scalability from two aspects: (A1) fix the number of keywords and vary the result selectivity; (A2) fix the result selectivity and vary the number of keywords.

For A1, we have the following observations according to Fig. 8.2: (1) TDELCA works much better than BwdELCA when the result selectivity is low, because BwdELCA needs to visit more UNs in this case; (2) HybELCA-HS works much better than BwdELCA when the result selectivity is high because of using hash probe operations; (3) TDELCA-L works best when the result selectivity is low; when the result selectivity becomes high, TDELCA-H works best. Generally speaking, the lower (higher) the selectivity, the more significant the performance gain of TDELCA (HybELCA-HS and TDELCA-H) on BwdELCA; and both TDELCA-L and TDELCA-H work best for all cases *on average*.

For A2, we have the following observation by comparing all results of Fig. 8.2 (a) to (d): the performance gains of TDELCA-L, HybELCA-HS and TDELCA-H over BwdELCA will increase with the increase of keywords, while the performance gain of TDELCA only increases when the result selectivity is low.

Fig. 8.3 shows the scalability when executing QX6 on XML documents with different sizes, from which we find that our methods achieve better scalability. For other queries, we have similar results, which are omitted due to space limit.

Experimental Results on DBLP Dataset

Table 8.3 shows the 10 queries we used on DBLP dataset. The experimental results are shown in Fig. 8.4, from which we know that for algorithms of ***Group 1.1***, our methods are much more efficient than BwdELCA; for algorithms of ***Group 1.2***, TDELCA-L beats HybELCA-HS for all queries. The reason also lies in that our methods avoid both the CAR and VUN problems. For our methods, from Fig. 8.4 we have similar result as

Table 8.3: Queries on DBLP dataset. $|L_m|(|L_1|)$ denotes the length of the *longest* (*shortest*) inverted list of Dewey labels, $N_E(N_S)$ is the number of ELCA(SLCA) results.

Query	Keywords	$\|L_1\|$	$\|L_m\|$	N_S	N_E
QD1	article, book	10,183	691,464	1,456	1,457
QD2	algorithm, article	42,695	691,464	18,349	18,350
QD3	data, article	66,317	691,464	26,611	26,612
QD4	article, database	18,630	691,464	5,753	5,754
QD5	xml, article	5,323	691,464	1,033	1,034
QD6	year, 2001	59,663	1,695,239	59,355	59,355
QD7	book, article, mining	10,183	691,464	6	7
QD8	algorithm, article, 2001	42,695	691,464	521	522
QD9	article, data, mining	11,853	691,464	1,563	1,564
QD10	data, xml, article	5,323	691,464	209	210

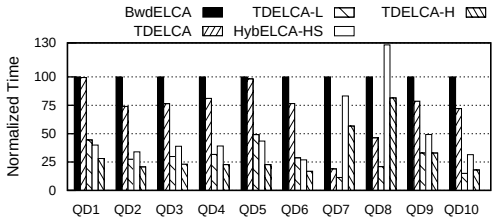

Figure 8.4: Running time of ELCA computation on DBLP.

Table 8.4: Queries on Treebank dataset. $|L_m|(|L_1|)$ denotes the length of the *longest* (*shortest*) inverted list of Dewey labels, $N_E(N_S)$ is the number of ELCA(SLCA) results.

Query	Keywords	$\|L_1\|$	$\|L_m\|$	N_S	N_E
QT1	nnp, nns	84,329	131,277	29,545	30,085
QT2	dt, vp	115,863	154,298	61,876	66,644
QT3	nnp, pp, vb	37,078	135,771	16,228	16,355
QT4	adjp, vbz, prp_dollar_	11,709	35,367	2,195	2,199
QT5	_hash_, _comma_, _period_	190	69,075	97	98
QT6	sbarq, advp, vbz, vbd	2,351	43,232	70	71
QT7	_questionmark_, _none_, _dollar_, whnp	6,996	54,091	10	11
QT8	_colon_, sbar, vbg, vbn	6,546	35,219	397	398
QT9	cd, dt, in, jj, nn	49,033	186,597	14,211	14,384
QT10	vbg, vbp, _quotes_, nnp, np	10,229	435,689	890	891

that of Section 8.3, i.e., TDELCA-L works better than TDELCA for all queries, and for TDELCA-L and TDELCA-H, no one can beat the other for all queries.

Experimental Results on Treebank Dataset

Table 8.4 shows the 10 queries we used on Treebank dataset. The experimental results are shown in Fig. 8.5, from which we have the same observations as that of Section 8.3. However, by comparing Figs. 8.4 and 8.5, we have an interesting observation: the performance gain of all algorithms over BwdELCA will increase with the increase of the document depth. This is because, BwdELCA uses the flattened index, i.e., IDList, it is unaware of the changes on document depth; while for TDELCA and TDELCA-L, the larger the document depth, the shorter the search interval for each binary search operation on average, therefore the more performance gain for TDELCA and TDELCA-L. For the same reason, the hash probe operation of HybELCA-HS and TDELCA-H can be done more efficiently compared with the binary search of BwdELCA on longer flattened IDList index.

8.4 SLCA Computation

As each SLCA node is a CA node that does not have other descendant CA nodes, and both BwdSLCA$^+$ and HybSLCA-HS can use a CA node to prune all its ancestor CA nodes, they are more efficient than BwdELCA and HybELCA-HS, respectively. On the other hand, all our algorithms on SLCA computation, i.e., TDSLCA,

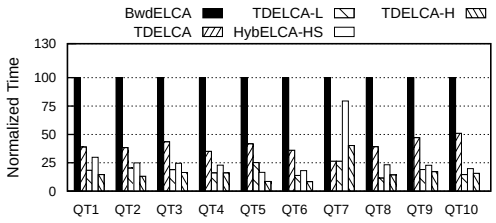

Figure 8.5: Running time of ELCA computation on Treebank.

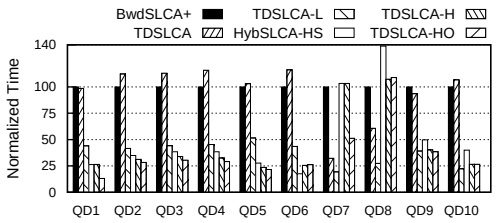

Figure 8.6: Running time of SLCA computation on DBLP.

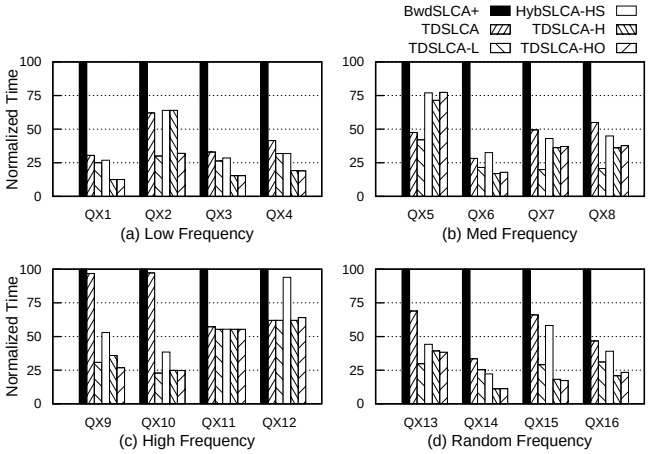

Figure 8.7: Running time of SLCA computation on XMark.

TDSLCA-L, TDSLCA-H and TDSLCA-HO, process all CA nodes in the same top-down manner as that on ELCA computation, they (except TDSLCA-HO) have similar overall performance as that of TDELCA, TDELCA-L and TDECA-H, respectively. Therefore, as shown in Figs. 8.6, 8.7 and 8.8, the performance gains of TDSLCA and TDSLCA-L (TDSLCA-H) on BwdSLCA$^+$ (HybSLCA-HS) are not as significant as their counterparts on ELCA computation (Figs. 8.1, 8.4 and 8.5).

From Figs. 8.6, 8.7 and 8.8 we have the following observations for algorithms of ***Group 2.1***: (1) for BwdSLCA$^+$ and TDSLCA, no one can beat each other for all queries; (2) TDSLCA-L works best due to reducing the cost and the number of binary search operations. For algorithms of ***Group 2.2***, we also have two observations: (1) TDSLCA-H and TDSLCA-HO usually perform better than HybSLCA-HS due to avoiding the VUN problem, and (2) even though TDSLCA-H and TDSLCA-HO have similar performance for most queries, for some queries, such as QD1 to QD5, QD7 in Fig. 8.6, QX2 and QX9 in Fig. 8.7, TDSLCA-HO performs better than TDSLCA-H due to avoiding useless hash probe operations. E.g., for QX9, the numbers of hash probe operations of TDELCA-H and TDELCA-HO are 241,253 and 77,358, respectively.

From Fig. 8.8 we find that even when the document depth becomes large, TDSLCA-L still works better than BwdSLCA$^+$, and TDSLCA-H and TDSLCA-HO have similar performance compared with HybSLCA-HS. This can be further verified according to the number of comparison and hash probe operations shown in Table 8.5. On the other hand, even though the number of comparison operations for TDSLCA is greater than that of BwdSLCA$^+$ for some queries, e.g., QT1 to QT3, QT5 and QT9, from Fig. 8.8 we know that they have similar

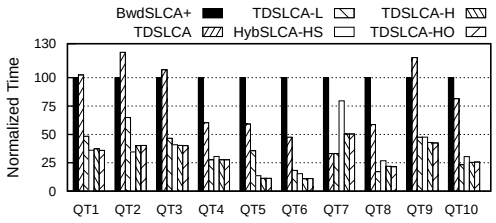

Figure 8.8: Running time of SLCA computation on Treebank.

Table 8.5: Number of comparison operations (for BwdSLCA$^+$, TDSLCA and TDSLCA-L) and hash probe operations (for HybSLCA-HS, TDSLCA-H and TDSLCA-HO, which are denoted as HybS-HS, TDS-H and TDS-HO, respectively).

Query	BwdSLCA$^+$	TDSLCA	TDSLCA-L	HybS-HS	TDS-H	TDS-HO
QT1	1,018,486	1,970,086	579,477	194,897	194,892	194,892
QT2	1,420,632	3,589,894	106,4158	363,253	363,291	363,291
QT3	1,041,003	1,897,183	672,022	178,529	178,522	178,522
QT4	365,910	347,439	206,261	30,606	30,593	30,593
QT5	5,386	7,830	3,270	491	485	485
QT6	49,008	35,727	30,041	3,097	3,091	3,091
QT7	4,671	3,941	2,655	1,587	1,583	1,502
QT8	164,598	131,523	97,276	11,429	11,428	11,428
QT9	1,564,190	2,912,962	1,097,418	309,615	309,607	309,575
QT10	304,843	287,861	183,146	22,436	22,425	22,425

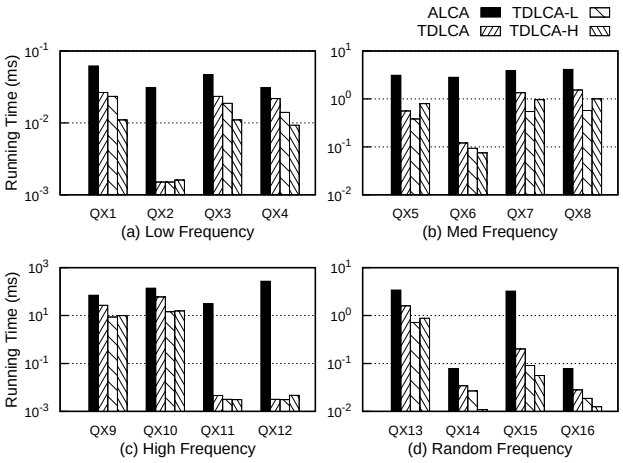

Figure 8.9: Running time of LCA computation on XMark (log-scaled).

performance, the reason has been explained in Section 8.3. The reason that TDSLCA-H and TDSLCA-HO have almost the same number of hash probe operations (Table 8.5) is that when the document depth becomes large, each node has very few child nodes, and almost all child nodes of each CA node are also CA nodes, thus TDSLCA-HO cannot pruning many useless child nodes in this case.

8.5 LCA computation

Fig. 8.9 shows the running time of different algorithms on LCA computation, from which we know that TDLCA, TDLCA-L and TDLCA-H perform much better than ALCA. The reason lies in that ALCA firstly computes SLCA nodes, then visits their ancestor nodes to find LCA nodes. Therefore, no matter which algorithm is adopted for SLCA computation, it visits all CA nodes at least two times. As a comparison, our methods visit each CA node only once, thus work better for all queries. For our methods, we have the similar conclusions as that of ELCA and SLCA computation, i.e., TDLCA-L and TDLCA-H work better than TDLCA, and for TDLCA-L and TDLCA-H, no one can beat the other for all queries. The experimental results for LCA computation on other datasets are omitted due to limited space.

9 Conclusions

Considering that the key factors resulting in the *inefficiency* for existing XML keyword search algorithms are the CAR and VUN problems, we proposed a *generic top-down* processing strategy that visits all CA nodes only *once*, thus *avoids* the CAR problem. We proved that the satisfiability of a node v w.r.t. the given semantics can be determined by v's child nodes, based on which our methods *avoid* the VUN problem. Another salient feature is that our approach is *independent* of query semantics. We proposed two efficient algorithms that are based on either traditional inverted lists or our newly proposed LLists to improve the overall performance. Further, we proposed two hash search-based methods to reduce the time complexity. The experimental results demonstrate the performance advantages of our proposed methods over existing ones.

One of our future work is studying disk-based index to facilitate XML keyword query processing when the size of indexes becomes too large to be completely loaded into memory.

10 Acknowledgments

This research was partially supported by the grants from the Natural Science Foundation of China (No. 61073060, 61040023, 61272124, 61103139, 61303040), and the Research Funds from Education Department of Hebei Province (No. Y2012014). Wei Wang was partially supported by ARC DP130103401 and DP130103405. Jeffrey Xu Yu was partially supported by RGC of the Hong Kong SAR, China, No. 418512.

Bibliography

[1] Jrmy Barbay, Alejandro Lpez-Ortiz, and Tyler Lu. Faster adaptive set intersections for text searching. In *WEA*, pages 146–157, 2006.

[2] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–87, 1976.

[3] Liang Jeff Chen and Yannis Papakonstantinou. Supporting top-k keyword search in xml databases. In *ICDE*, pages 689–700, 2010.

[4] Yi Chen, Wei Wang, and Ziyang Liu. Keyword-based search and exploration on databases. In *ICDE*, pages 1380–1383, 2011.

[5] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. Xsearch: A semantic search engine for xml. In *VLDB*, pages 45–56, 2003.

[6] Bolin Ding and Arnd Christian König. Fast set intersection in memory. *PVLDB*, 4(4):255–266, 2011.

[7] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *SIGMOD Conference*, pages 16–27, 2003.

[8] Vagelis Hristidis, Nick Koudas, Yannis Papakonstantinou, and Divesh Srivastava. Keyword proximity search in xml trees. *IEEE Trans. Knowl. Data Eng.*, 18(4):525–539, 2006.

[9] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed xml documents. In *VLDB*, pages 273–284, 2003.

[10] Lingbo Kong, Rémi Gilleron, and Aurélien Lemay. Retrieving meaningful relaxed tightest fragments for xml keyword search. In *EDBT*, pages 815–826, 2009.

[11] Changqing Li, Tok Wang Ling, and Min Hu. Efficient updates in dynamic xml data: from binary string to quaternary string. *VLDB J.*, 17(3):573–601, 2008.

[12] Guoliang Li, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. Effective keyword search for valuable lcas over xml documents. In *CIKM*, pages 31–40, 2007.

[13] Yunyao Li, Cong Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, pages 72–83, 2004.

[14] Ziyang Liu and Yi Chen. Reasoning and identifying relevant matches for xml keyword search. *PVLDB*, 1(1):921–932, 2008.

[15] Ziyang Liu and Yi Chen. Processing keyword search on xml: a survey. *World Wide Web*, 14(5-6):671–707, 2011.

[16] Jiaheng Lu, Tok Wang Ling, Chee Yong Chan, and Ting Chen. From region encoding to extended dewey: On efficient processing of xml twig pattern matching. In *VLDB*, pages 193–204, 2005.

[17] Chong Sun, Chee Yong Chan, and Amit K. Goenka. Multiway slca-based keyword search in xml data. In *WWW*, pages 1043–1052, 2007.

[18] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD Conference*, pages 204–215, 2002.

[19] Dimitris Tsirogiannis, Sudipto Guha, and Nick Koudas. Improving the performance of list intersection. *PVLDB*, 2(1):838–849, 2009.

[20] Weiyan Wang, Xiaoling Wang, and Aoying Zhou. Hash-search: An efficient slca-based keyword search algorithm on xml documents. In *DASFAA*, pages 496–510, 2009.

[21] Xiaodong Wu, Mong-Li Lee, and Wynne Hsu. A prime number labeling scheme for dynamic ordered xml trees. In *ICDE*, pages 66–78, 2004.

[22] Yu Xu and Yannis Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD Conference*, pages 537–538, 2005.

[23] Yu Xu and Yannis Papakonstantinou. Efficient lca based keyword search in xml data. In *EDBT*, pages 535–546, 2008.

[24] Junfeng Zhou, Zhifeng Bao, Wei Wang, Tok Wang Ling, Ziyang Chen, Xudong Lin, and Jingfeng Guo. Fast slca and elca computation for xml keyword queries based on set intersection. In *ICDE*, pages 905–916, 2012.

[25] Junfeng Zhou, Zhifeng Bao, Wei Wang, Jinjia Zhao, and Xiaofeng Meng. Efficient query processing for xml keyword queries based on the idlist index. *The VLDB Journal*, 10.1007/s00778-013-0313-2.

[26] Junfeng Zhou, Xingmin Zhao, Wei Wang, Ziyang Chen, and Jeffrey Xu Yu. Top-down keyword query processing on xml data. In *CIKM*, pages 2225–2230, 2013.

[27] Rui Zhou, Chengfei Liu, and Jianxin Li. Fast elca computation for keyword queries on xml data. In *EDBT*, pages 549–560, 2010.