Top-Down XML Keyword Query Processing

Junfeng Zhou, Wei Wang, Ziyang Chen, Jeffrey Xu Yu, Xian Tang, Yifei Lu, Yukun Li

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 LLLists 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 LLLists. The experimental results verify the benefits of our methods according to various evaluation metrics.

Index Terms—XML, Keyword Search, LCA, LLList

1 INTRODUCTION

XML has been successfully used in many applications, such as that in scientific and business domains, as the standard format for storing, publishing and exchanging data. Compared with structured query languages, such as XPath and XQuery, keyword search is also gained popularity on XML data as it relieves users from understanding the complex query languages and the structure of the underlying data, and has received much attention [1]–[21] due to that results are not the entire documents anymore but nested fragments.

Typically, an XML document can be modeled as a node-labeled tree T. For a given keyword query Q, several semantics [1], [2], [5], [6], [10] 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) [2]–[4], [7], [14] and Smallest LCA (SLCA) [5], [7]–[9], [11]. SLCA defines a subset of LCA nodes, of which no LCA is the ancestor of any other LCA. As a comparison, ELCA tries to capture more meaningful results, it may take some LCAs that are not SLCAS as meaningful results.

Example 1: Assume that a user issues Q1 trying to find some information he may be interested in from D in Fig. 1. If the query semantics is SLCA, then the book and papers represented by nodes 6, 10 and 16 will be returned, from which we know that the three publications are written by “Tom” with titles containing “XML”. As a comparison, he will get one more result, i.e., node 1, with ELCA semantics, meaning that “Tom” is the manager of the lab, for which the research field is “XML”. Besides the above four results, node 4 is also taken as a result with LCA semantics.

Obviously, a system supporting more query semantics will facilitate users to find interesting results, since any query semantics cannot work well in all situations. However, each of existing algorithms [2]–[5], [7], [8], [11], [15]–[17] focuses on a certain query semantics. Simply implementing them to support all query semantics will result in big index size and make it unscalable to new query semantics, and more importantly, these algorithms are still inefficient due to redundant computation.

Given a keyword query Q = \{k_1, k_2, ..., k_m\}, all nodes that directly contain each query keyword is typically organized with an inverted list, where each node v is usually represented as a Dewey label [22], which consists of a set of components representing all nodes on the path from the document root to v. Therefore, if we take each inverted list as a set of components, then the key problem of processing Q is how to find fewer results from m much larger sets of components by visiting components as few as possible. Existing methods on LCA/ELCA/SLCA computation in fact made improvements following this principle, from firstly sequentially visiting all components once, such as DIL [2] and Stack [5], to skipping useless components with indexes [3]–[5], [7], [8], [11], [15]–[17], however, they still suffer from redundant computation by visiting many useless components. Generally speaking, the common problems that result in their redundancy are the CAR and VUN problems.
The CAR problem: As identified in [16], [17], existing methods [2]–[5], [7]–[9], [11], [14] assign each node \( v \) a Dewey label [22], or one of its variants [4], [7], [10], [11], based on which two basic operations are adopted to get qualified results, i.e., \((\text{OP}1)\) testing the document order of two nodes and \((\text{OP}2)\) 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 \( \text{OP}1 \) or \( \text{OP}2 \) equals visiting all CAs of the two involved nodes once. In practice, as \( v \) could be a CA of multiple nodes, frequently performing \( \text{OP}1 \) and \( \text{OP}2 \) 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 [2], [3], [5], [8], [9] that are based on \( \text{OP}1 \) and \( \text{OP}2 \) operations, because they take each Dewey label as the basic processing unit, without noticing that some components repeatedly appear in many different Dewey labels.

The VUN problem: 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 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 are UNs; (3) auxiliary nodes (ANs), each of which is a child node of some CA node and belongs to \( V = V_1 \cup V_2 \), such as \( V_3 \) in Fig. 1. As show by Fig. 1, to get qualified results for \( Q_2 \) on \( D \), none of existing methods [2]–[5], [7]–[9], [11], [16], [17] can avoid visiting useless nodes, i.e., visiting nodes in \( V_2 \), which we call as the VUN problem. The reason 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 (including UNs), rather than its child nodes (ANs).

Considering the above problems, we propose to support different query semantics with a generic processing strategy, which is more efficient by avoiding both the CAR and VUN problems, such that to further reduce the number of visited components. 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 \( xLCA \) (i.e., \( xLCA \) 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, our method only needs to visit nodes of \( V_1 \cup V_2 \), rather than additionally visits 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 [4], [11], [17] 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 [18], we have several major updates: (1) We give an in-depth analysis to related work in Section 2.3. (2) We discuss the extension of our methods to other LCA-based semantics in Section 6. (3) We propose several algorithms that are based on hash search to accelerate \( xLCA \) computation in Section 7. (4) We conduct additional performance study in Section 8.

2 PRELIMINARIES AND RELATED WORK

2.1 Data Model

We model an XML document as an ordered tree (Fig. 1), where nodes represent elements or attributes\(^1\), while edges represent direct nesting relationship between nodes. We say node \( v \) directly contains keyword \( k \) or \( v \) is a keyword node w.r.t. \( k \), if \( k \) appears in the node

\[1\]

\(1\) In our discussion, each text data is taken as the text value of the element or attribute node. E.g., in Fig. 1, “Tom” is the text value of node 5, rather than that of nodes 4 and 1.
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 $v$, $u <_d v$ means that $u$ is located before $v$ in document order, $u <_a v$ means that $u$ is an ancestor node of $v$, $u <_p v$ denotes that $u$ is the parent node of $v$. If $u$ and $v$ represent the same node, we have $u = v$, and both $u <_d v$ and $u <_a v$ hold.

To accelerate the query processing, existing methods [2]–[5], [7]–[9], [11], [16] 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 [16], [17], a Dewey label [22] or one of its variants [4], [7], [10], [11], [23]–[25]. In Fig. 1, we denote each node $v$ by its ID, based on which its ID Dewey label [4], [11] label can be easily got by concatenating all IDs on the path from the root node to $v$. E.g., the ID Dewey label of node “5” is “1.4.5”. In the following discussion, we do not differentiate a node, its ID, and the corresponding ID Dewey label unless there is ambiguity.

2.2 Query Semantics

Given a query $Q = \{k_1, k_2, ..., k_m\}$ and XML document $D$, we use $L_i$ to denote the inverted list of IDDewey labels sorted in document order w.r.t. $k_i$. Fig. 2 shows the inverted lists of $Q_i$ in Fig. 1.

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

E.g., for query $Q_1$ and $D$ in Fig. 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 $LCA(Q) = \{v \mid v = lca(v_1,v_2, ..., v_m), v_i \in L_i(1 \leq i \leq m)\}$.

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

Based on LCA, the two mostly adopted query semantics are SLCA [5], [8] and ELCA [2]–[4]. SLCA defines a subset of LCA, of which no LCA is the ancestor of any other LCA.

Definition 3: (SLCA) Given a query $Q$ and an XML document $D$, the set of SLCAs of $Q$ on $D$ is $SLC(A(Q) = \{v \mid v \in LCA(Q) \text{ and } \forall u \in LCA(Q), \text{ such that } v <_a u\}$.

According to Definition 3, SLCA nodes of $Q_1$ on $D$ in Fig. 1 are nodes 6, 9, 10 and 16.

### Table 1: Comparison of worst-case time complexities for algorithms on ELCA computation, where $|L_1|(|L_m|)$ is the length of the shortest (longest) inverted list consisting of Dewey/IDDewey labels, $|L_1|(|L_m|)$ is the number of distinct node IDs in the shortest (longest) ID List, $m$ is the number of keywords of the given query, and $d$ is the depth of the given XML tree.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
<th>CAR</th>
<th>VUN</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIL [2]</td>
<td>$O(m \cdot \log</td>
<td>L_m</td>
<td>)$</td>
</tr>
<tr>
<td>IS [3]</td>
<td>$O(m \cdot \log</td>
<td>L_m</td>
<td>)$</td>
</tr>
<tr>
<td>[Dewey-E] [7]</td>
<td>$O(m \cdot \log</td>
<td>L_m</td>
<td>)$</td>
</tr>
<tr>
<td>BwdELCA-HS [17]</td>
<td>$O(</td>
<td>L_m</td>
<td>)$</td>
</tr>
<tr>
<td>FwdELCA-LS [17]</td>
<td>$O(</td>
<td>L_m</td>
<td>)$</td>
</tr>
<tr>
<td>FwdELCA-HS [17]</td>
<td>$O(</td>
<td>L_m</td>
<td>)$</td>
</tr>
<tr>
<td>BwdELCA-HS [17]</td>
<td>$O(</td>
<td>L_m</td>
<td>)$</td>
</tr>
<tr>
<td>FwdELCA-LS [17]</td>
<td>$O(</td>
<td>L_m</td>
<td>)$</td>
</tr>
<tr>
<td>FwdELCA-HS [17]</td>
<td>$O(</td>
<td>L_m</td>
<td>)$</td>
</tr>
<tr>
<td>TDELCA</td>
<td>$O(m \cdot \log</td>
<td>L_m</td>
<td>)$</td>
</tr>
<tr>
<td>TDELCA-L</td>
<td>$O(m \cdot \log</td>
<td>L_m</td>
<td>)$</td>
</tr>
<tr>
<td>TDELCA-H</td>
<td>$O(m \cdot \log</td>
<td>L_m</td>
<td>)$</td>
</tr>
<tr>
<td>TDELCA-HS</td>
<td>$O(m \cdot \log</td>
<td>L_m</td>
<td>)$</td>
</tr>
</tbody>
</table>

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 $ELCA(Q) = \{v \mid v \exists v_i \in L_1, ..., v_m \in L_m(v = LCA(v_1, ..., v_m) \land \forall i \in [1, m], \exists x(x \in LCA(Q) \land child(v, v_i) \leq_a x))\}$, where $child(v, v_i)$ is the child of $v$ on the path from $v$ to $v_i$, and $[1, m] = \{1, 2, ..., m\}$.

Intuitively, ELCA may take some LCAs that are not SLCA as results that users may be interested in. According to Definition 4, ELCA nodes of $Q_1$ on $D$ in Fig. 1 are nodes 1, 6, 10 and 16.

2.3 Related Work

Here we only focus on algorithms on ELCA computation, we refer readers to [12], [17], [19] for a complete coverage. Existing algorithms on ELCA computation can be generally classified into two categories: (1) algorithms that cannot avoid the CAR problem, and (2) algorithms that do not suffer from the CAR problem. Table 1 summarizes different ELCA algorithms.

2.3.1 Algorithms with the CAR Problem

DIL [2] sequentially processes all involved Dewey labels in document order, its performance is linear to the number of involved Dewey labels. IS [3] 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 both the CAR and VUN problems.

JDewey-E [7] 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 suffers
from the CAR problem. Meanwhile, as it performs set intersection on all lists of each tree depth from the leaf to the root, they will firstly visit nodes of V_2 in Fig. 1 for Q_2, thus it also suffers from the VUN problem.

As some node IDs appear in many different IDDewey labels of the same inverted list, and HC [4] processes each IDDewey label of the shortest list separately, it still suffers from the CAR problem. Moreover, HC needs to push each component of every IDDewey label into a stack, it also suffers the VUN problem when the pushed components are UNs.

2.3.2 Algorithms without the CAR Problem

FwdELCA and BwdELCA [16], [17] 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 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\}, [16], [17] use the set intersection operation on IDLists to get all CA nodes, as shown by Formula 1.

\[
CA(Q) = \bigcap_{i=1}^{m} L_i
\]

As shown in [16], [17], 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 IDList, both FwdELCA and BwdELCA avoid the CAR problem. However, as 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, thus both the two algorithms suffer from the VUN problem. Moreover, the flattened structure of IDList makes them suffer from higher cost, i.e., \(O(\log |L_m|)\), in checking the appearance of each node in another IDList (see Formula 1), while the cost of our methods is \(O(\log |S|)\) (see Section 5.2 for detailed explanation), where \(|S| \leq |L_m| \leq |L_1|\).

Further, [17] 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 but not the VUN problem due to the same reason as FwdELCA and BwdELCA.

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 keyword 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 \(T_v\) rooted at node \(v\), and checks whether each node of \(T_v\) contains all keywords of Q in a “top-down” way. The “top-down” means that if \(T_v\) contains all keywords of Q, then \(v\) must be a CA node. We then remove \(v\) and get a forest \(F_v = \{T_{v_1}, T_{v_2}, ..., T_{v_n}\}\) of subtrees rooted at the n child nodes of \(v\). Based on \(F_v\), we further find the set of subtrees \(F_{vCA} \subseteq F_v\) where each subtree \(T_{v_i} \in F_{vCA}\) contains every keyword of Q at least once, i.e., node \(v_i\) is a CA node. If \(F_{vCA} = \emptyset\), it means that for \(T_{v_i}\), only \(v\) is a CA node, then we can safely skip all nodes of \(T_v\) from being processed; otherwise, for each subtree \(T_{v_i} \in F_{vCA}\), we recursively compute its subtree set \(F_{vCA}^i\) until \(F_{vCA} = \emptyset\).

Let \(S_i(v)\) denote, for \(v\), the set of child nodes that contain \(k_i\), \(S_{ca}(v)\) the set of child CA nodes of \(v\), and \(CA(T_v)\) the set of CA nodes in \(T_v\). Formula 2 means that the set of CA nodes of Q equals the set of CA nodes in \(T_r\), where \(r\) is the document root node. \(CA(T_v)\) can be recursively computed according to Formula 3. Formula 3 means that for a given CA node \(v\), the set of CA nodes in \(T_v\) is equal to the union of \(S_i(v)\) and the set of CA nodes in subtrees rooted at \(v\)’s child CA nodes, which can be further computed by Formula 4.

\[
CA(Q) = CA(T_r)
\]

\[
CA(T_v) = \{v\} \bigcup \left( \bigcup_{u \in S_{ca}(v)} CA(T_u) \right)
\]

\[
S_{ca}(v) = \bigcap_{i=1}^{m} S_i(v)
\]

Example 2: Consider \(Q_1\) and \(D\) in Fig. 1. According to Formula 2, \(CA(Q_1) = CA(T_1)\). Since nodes 4 and 14 are child CAs of node 1 in Fig. 1, i.e., \(S_{ca}(1) = \{4, 14\}\), we have \(CA(T_1) = \{1\} \bigcup CA(T_4) \bigcup CA(T_{14})\) according to Formula 3. As \(S_{ca}(4) = \{6, 10\}\), \(S_{ca}(14) = \{16\}\) and \(S_{ca}(6) = S_{ca}(10) = S_{ca}(16) = \emptyset\), we know that \(CA(Q_1) = \{1, 4, 6, 10, 14, 16\}\).

Corollary 1: Either one of Dewey [22], JDewey [7], IDDewey [4], [11], MDC [10] and Extended Dewey [23] labeling schemes can be used for top-down CA computation.

Proof: For a given CA node \(v\) which is assigned a Dewey-like label, i.e., either one of Dewey, JDewey, IDDewey, MDC and Extended Dewey labels, since the common property for these labeling schemes is that each child node of \(v\) is represented by a distinct ID value compared with its sibling nodes, the set intersection operation can correctly return all child CA nodes of \(v\) by intersecting \(v\)’s m sets of child nodes (Formula 4). Assume that the root node is a CA node (otherwise, we don’t need to do anything), we can recursively get all
CA nodes in the top-down way by getting child CA nodes each time (Formulas 2 and 3).

According to Corollary 1, inverted lists can be generated based on Dewey labeling scheme or either one of its variants. Hereafter, IDDewey labels are used for clearer description, as each ID can be uniquely represented by intersecting inverted lists of keywords. If a node is not an LCA node, than the operation of removing all occurrences of all query keywords in the 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 $L_i(v)$ denotes the number of occurrences of $k_i$ in $T_v$, if $\forall i \in [1, m], |L_i(v)| > \sum_{u \in S_{ca}(v)} |L_i(u)|$, 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.

4 THE BASELINE ALGORITHM

Our baseline ELCA algorithm recursively gets all CA nodes in a top-down way, then checks the satisfiability of each CA node, which works on the traditional inverted lists of labels w.r.t. Dewey or one of its variants as shown in Fig. 2. To do so, it needs 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 $P_1$, given a query $Q$ with $m$ keywords, we know that $\forall i \in [1, m], S_{ca}(v) \subseteq S_i(v)$. Thus given a CA node $v$ and its subtree set $\mathcal{F}_v = \{T_{v_1}, T_{v_2}, ..., T_{v_n}\}$, to get $S_{ca}(v)$, we do not need to check whether each subtree contains all query keywords; instead, we just need to check whether each node in $S_{min}(v)$, which contains least number of child nodes of $v$ w.r.t. $k_{min}$, appears in $S_i(v)(i \in [1, m], i \neq min)$.

E.g., for $Q_1$ and $D_1$ of Fig. 1, $S_{XML}(4) = \{5, 6, 10\}$, $S_{XRM}(4) = \{6, 10\}$. Since $|S_{XML}(4)| > |S_{XRM}(4)|$, we know $S_{min}(4) = S_{XML}(4)$. To compute $S_{ca}(4)$, we just need to check whether each node of $S_{XML}(4)$ is contained by $S_{XRM}(4)$, then we know $S_{ca}(4) = \{6, 10\}$.

However, in Fig. 2, nodes of $S_{XML}(4)$ appear in $L_2$ as $L_2(4) = \{6, 6, 10, 10\}$, which we call as the child list of node 4 w.r.t. the second query keyword, i.e., XML. Obviously, nodes of $S_i(v)$ are different with each other, while in $L_i(v)$, each node of $S_i(v)$ may repeatedly appear many times. Even if we know the lengths of all child lists, it's difficult to know which one is $S_{min}(v)$. 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 $|S_{min}(v)|$.

For $P_2$, we use the following Lemma to check the satisfiability of $v$, which is similar to [4], [16], [17].

Lemma 1: Given a query $Q = \{k_1, k_2, ..., k_m\}$ and CA node $v$, let $L_i(v)$ be the child list of $v$ w.r.t. $k_i$. $v$ is an ELCA node iff $\forall i \in [1, m], |L_i(v)| > \sum_{u \in S_{ca}(v)} |L_i(u)|$.

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 CA nodes in the original XML tree, according to Definition 1 and 2, there exists a unique child CA node $u_{c_r}$, and all other child nodes of $u$ do not contain any query keyword. If $u_{c_r}$ is not an LCA node either, we can recursively get the closest descendant LCA node of $u_{c_r}$, 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 $v$.

4.1 The Algorithm

Based on the above description, Algorithm 1 recursively gets all CA nodes in a top-down way. For each CA node $v$, it finds out the number of occurrences of each query keyword in its subtree, i.e., the length of each of its child list, then gets $v$'s child CA nodes by intersecting $v$'s child lists using binary search operation. After that, it checks the satisfiability of $v$ by Lemma 1.

To do so, each inverted list $L_i$ is associated with a cursor $C_i$ pointing to some IDDewey label of $L_i$, $C_i[x]$ denotes the $x$th component of the IDDewey label that $C_i$ points to, and pos($C_i$) is used to denote $C_i$'s position in $L_i$. Given a node $v$, we use $l(v)$ to denote the IDDewey label of $v$, $v.N_i$ denotes the number of keyword occurrences w.r.t. $k_i$ in the subtree rooted at $v$.

As shown in Algorithm 1, it firstly initializes the subtree rooted at the root node of the given XML tree in line 1, then calls the procedure processCANode() to recursively get all CA nodes in line 2.

The procedure processCANode() works as follows. It firstly gets the depth of $v$'s child nodes in line 1. In lines 2-7, it repeatedly gets all child CA nodes of $v$. For each child CA node $u$ got in line 3, it firstly gets the values of variables associated with $v$ in line 5; in line 6, it excludes the occurrences of all query keywords under $u$ from that under $v$. In line 7, it calls processCANode() to recursively process $u$. After processing all child CA nodes of $v$, if $\forall i \in [1, m], v.N_i > 0$, according to Lemma 1, $v$ is an ELCA node and outputted in line 9.

Given a CA node $v$, the function getNextChildCA() is called to find a child CA node of $v$ by intersecting $m$ child lists of $v$. It always uses the cursor with the maximum $chL.1^{th}$ ID value as the eliminator (line 1, and $C_v[chL]$ is the eliminator), and uses the static probing
As node 6, we restart the probing from node 6, 6, 10, 10, we have that for node\( N \) that node 10 is an ELCA node. After that, we check the inverted lists in Fig. 2. Since node 1's first child CA from its two child lists, i.e., the second level of the two

order from the shortest list to the longest (lines 2-7). 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 IDDewey labels of same ID are found (line 6); otherwise, since we found an IDDewey label with ID larger than

4.2 Analysis

As each CA node is visited only once, our method does not suffer from the CAR problem [16]. Besides, according to Algorithm 1, we have the following result.

Corollary 2: (Independence) In Algorithm 1, the satisfiability of each CA node \( v \) w.r.t. ELCA semantics can be determined by \( v \)'s child nodes.

Proof: According to Lemma 1, Corollary 2 holds if we know \( v.N_i \) and \( u.N_i \), where \( u \in S_{CA}(v) \). Since our method does not pre-compute and maintain the values of \( v.N_i \) and \( u.N_i \), we need to get them during the processing. According to Procedure initializeChildCA(), to get \( v.N_i \)'s value, our method only needs to check the length of the corresponding interval consisting of \( v \)'s child CA node, i.e., computing \( v.N_i \)'s value does not need to visit any of its descendant nodes. Therefore, based on Lemma 1 and Algorithm 1, Corollary 2 holds.

The importance of Corollary 2 is that compared with existing methods, our method avoids the VUN problem. E.g., consider \( Q_2 \) in Fig. 1, since the unique CA node is node 1, our method only needs to visit the child nodes of node 1 w.r.t. the query keywords, i.e., nodes 3, 4, and 14. As a comparison, existing algorithms may visit UNs as shown in Section 2.3.

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 list of each CA node w.r.t. \( k_i \) is bounded by \( |L_{m_i}| \), the cost of checking whether a node is a CA node is \( O(m \log |L_{m_i}|) \). If we always process nodes of \( S_1(v) \) for each CA node, the number of processed nodes of TDELCA is \( \sum_{v \in CA(Q)} |S_1(v)| \). 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 \( S_1(v) \), thus the total number of processed nodes is bounded by \( \sum_{v \in CA(Q)} |S_1(v)| \). Therefore, the time complexity of the TDELCA algorithm is \( O(m \cdot \sum_{v \in CA(Q)} |S_1(v)| \cdot \log |L_{m_i}|) \), which is better than existing methods without using hash indexes. Note that in any case,

\[ \sum_{v \in CA(Q)} |S_1(v)| \leq |L_1| \leq d \cdot |L_1| \text{ and } |L_m| \leq |L_m| \]

where \( d \) is the depth of the given XML tree (see Table 1 for a detailed comparison).

Similar to [16], [17], any set intersection algorithm [26], [27] can be used in our method for CA computation, and any of the search methods (e.g., binary, galloping [28], or interpolation search [29]) can be used to complete the set intersection in our getNextChildCA() function.
5 The LLIST-based Algorithm

Algorithm 1 needs two probe operations (binary search is performed twice) to find the first and last IDDewey labels of \( L_i(v) \), and as shown in Fig. 2, 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. Considering the above problems, we propose a new index, namely LList, based on which we can reduce both the cost and calling times of binary search operation.

5.1 The LLIST Index

Let \( L_i \) be the inverted IDDewey label list of \( k_i \), the corresponding inverted list \( L_i \mid k_i \) consists of at most \( d_i \) subtlists, i.e., \( L_i = \{ L_i^1, L_i^2, \ldots, L_i^{d_i} \} \), where \( d_i \) denotes the number of components of the longest IDDewey label in \( L_i \). Each \( L_i^j (j \in [1, d_i]) \) consists of entries representing the set of distinct nodes in the \( j \)th level of \( L_i \). Each entry \( e_v \) \( \in L_i^j \) consists of two numeric values:

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

We call each \( L_i \) an LList, denoting that all nodes in the same level are maintained together. Fig. 3 shows the two LLists of “Tom” (\( L_1 \)) and “XML” (\( L_2 \)).

![Fig. 3: LLists of “Tom” (\( L_1 \)) and “XML” (\( L_2 \)).](image)

5.2 The Algorithm

We call the algorithm based on the LList index as TDELCA-L. As each node appears in \( L \) only once, binary search can be done more efficiently. Based on LList index, the satisfiability of a node can be determined rather simple based on the following lemma.

**Lemma 2:** For a query \( Q = \{ k_1, k_2, \ldots, k_m \} \) and CA node \( v \), \( v \) is an ELCA node iff \( S_{ca}(v) = \emptyset \) or \( \forall i \in [1, m], |S_i(v)| - |S_{ca}(v)| > 0 \).

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

**Example 4:** Consider \( Q \) again. From Fig. 3 we know \( |S_{1}(1)| = |S_{2}(1)| = 3 \). Since \( |S_{ca}(1)| = |\{4,14\}| < 3 \), according to Lemma 2, node 1 is an ELCA node. For node 4, since \( |S_{1}(4)| = 3, |S_{2}(4)| = 2 \) and \( |S_{ca}(4)| = |\{6,10\}| = 2 \), we know \( |S_{2}(4)| - |S_{ca}(4)| = 0 \), thus according to Lemma 2, node 4 is not an ELCA node.

TDELCA-L finds all CA nodes and checks their satisfiability in the same order as that of TDELCA, but works based on LList index. Specifically, the improvements lie in the following aspects: (1) binary search operation does not need to be called by initializeChildCA() anymore due to that we do not need to locate the start and end position of a child list as TDELCA does; (2) binary search operation is performed on a much shorter child list on average due to that each node appears in \( L \) only once; (3) binary search operation stops immediately when it spots a value that equals the eliminator due to the same reason as (1).

**Example 5:** Consider \( Q_1 \) and \( D \) in Fig. 1. To get child CAs of node 1, we need to know node 1’s child lists. Given \( L_1 \) and \( L_2 \), we have \( L_1(1) = S_{1}(1) = [2,4,14] \) and \( L_2(1) = S_{2}(1) = [3,4,14] \). As a comparison, given \( L_1 \) and \( L_2 \) in Fig. 2, the corresponding child lists are
$L_1(1) = 2,4,4,4,14$ and $L_2(1) = 3,4,4,4,14,14$ for TDELCA. Obviously, the intersection operation is performed on much shorter lists when performing binary search operation. Moreover, as each node in a child list of LList is distinct from others, binary search operation can stop immediately when it spots a value that equals the eliminator. Further, in TDELCA, we need to perform binary search operation in initializeChildCA() to get the position of the last IDDevey label 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 do binary search operation to know each child list, which can be directly got by $p_{i\epsilon}$’s value in the entry of $v$. \hfill $\square$

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 $S$ be the set that contains the maximum number of child nodes for all nodes, obviously $|S| \leq |L_m|$. As the number of processed nodes of TDELCA-L equals that of TDELCA, the time complexity of TDELCA-L is $O(m \times (\sum_{i\epsilon A(Q)} |S_i(v)|) \cdot \log |S|)$.

It is worth noting that simply removing all repeated appearances of each node of $L_i$ to construct $L_{i+1}$ may result in losing the information that some non-leaf nodes may directly contain $k_i$, since in $L_i$, we are only sure that leaf nodes are keyword nodes w.r.t. $k_i$. Our solution is maintaining a dummy entry $c^d_i$ in $L_{i+1}$ with $c^d_i.id = 0$ denoting that $v$ directly contains $k_i$ if $v$ has descendant nodes containing the same keyword as $v$. The details could be found from Appendix A.

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 $S_{ca}(v) = \emptyset$.

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

6.2 LCA Computation

If the underlying index is inverted lists of Dewey labels or one of its variants, we can check the satisfiability of each CA node $v$ w.r.t. LCA according to Lemma 4.

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

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

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 $S_{ca}(v) = \emptyset$ or $\exists i \in [1, m]$ such that $|S_i(v)| > 1$.

Proof: If $S_{ca}(v) = \emptyset$, it means that $v$ is an SLCA node, therefore an LCA node. If $\exists i \in [1, m]$, such that $|S_i(v)| > 1$, 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., $S_{ca}(v) = \emptyset$, 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 $|S_i(v)| > 1$. Therefore, $v$ is an LCA node of $Q$ iff $S_{ca}(v) = \emptyset$ or $\exists i \in [1, m]$, such that $|S_i(v)| > 1$. \hfill $\square$

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 Corollaries 1 and 2 still hold for all algorithms.

6.3 Other Semantics

Besides LCA, ELCA and SLCA, researchers have proposed other query semantics, including XSeach [1], MLCA [6] and VLCA [10], all are based on LCA. The common feature of XSeach, MLCA and VLCA is that their validation criteria needs 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, 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.

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 [4], [11], [17] on inverted lists, such that each probe operation takes $O(1)$ time without using binary search operation.

As shown in Fig. 4, the first hash table $H_F$ records the number of nodes in each $L_i$, which is used to choose the shortest LList. For each $L_i$, another hash table $H_i$ records, for each node of $L_i$, the number of its child nodes that contain $k_i$. Note that $H_i$ in our methods is
TDELCA-L, for a given query \( Q \), TDELCA-H only needs to process all CA nodes and their child nodes in \( L_1 \). For each processed node \( v \) in \( L_1 \), 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 again. According to Fig. 4 (a), \( H_F[|\text{Tom}|] = 11 < H_F[|\text{XML}|] = 13 \), our method processes nodes of \( L_1 \) to get all ELCA nodes. Take node 4 for example, since \( 4 \in H_2 \), we know that node 4 is a CA node. As the first child node of node 4 in \( L_1 \) is node 5, and \( 5 \not\in H_2 \), 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, for node 4, \( N_2 - |S_{\text{ca}}(3)| = 0 \). Finally, we get all ELCA results, i.e., nodes 1, 6, 10 and 16.

Note that since \( H_1 \) 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 IDDavey labels 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 \( O(m \cdot \sum_{v \in CA(Q)} |S_i(v)|) \).

Compared with the hash methods with time complexity \( O(m \cdot |L_1|) \) proposed by [17] for ELCA computation (Table 1), as \( CA(Q) \subseteq L_1 \) and \( \bigcup_{v \in CA(Q)} S_i(v) \subseteq L_1 \), we know that \( \sum_{v \in CA(Q)} |S_i(v)| \leq |L_1| \), thus we can expect that TDELCA-L works better than FwdELCA-HS, BwdELCA-HS and HybELCA-HS.

We can extend TDELCA-H to LCA and SLCA semantics based on Lemmas 3 and 5, which we call as TDELCA-L and TDSLCA-H, respectively.

### 7.2 The Optimized Hash Search Algorithm

As the performance of TDELCA-H is dominated by the number of hash probe operations, we may make improvements if we can save as many hash probe operations as possible. We propose the second hash indexes to accelerate ELCA computation. We show this method by the sample XML document \( D_1 \) in Fig. 5 (a). Given a query \( Q = \{k_1, k_2\} \), their LLists are shown in Fig. 5 (b) and (c), respectively. As \( |L_1| = 5 < |L_2| = 6 \), same as TDELCA-H, we use \( L_1 \) as the working list and check whether each node of \( L_1 \) is a CA node by hash probing \( H_2 \). The difference lies in that if a node \( v \) in \( L_2 \) does not have child nodes, its hash value equals \( -1 \) as shown by Fig. 5 (d), rather than 1 as that in Fig. 4 (b), where \(-1\) means that \( v \) directly contains \( k_2 \) and \( v \) does not have descendants directly containing \( k_2 \), based on which we have the following result.

Lemma 6: Let \( v \) be a CA node of \( Q = \{k_1, k_2, \ldots, k_m\} \), \( n_{\min} = \min \{H_2[v], H_3[v], \ldots, H_m[v]\} \). If \( n_{\min} = -1 \), then \( S_{\text{ca}}(v) = \emptyset \).
In $L_2$, then its hash value in $H_2$ is $\infty$. Based on the new hash indexes, when processing a node $v$ of $L_1$, we have the chance to avoid more unnecessary hash probe operations, as shown by Lemmas 7 and 8.

**Lemma 7:** Let $v$ be a CA node of $Q = \{k_1, k_2, ..., k_m\}$, $c_{\text{max}} = \max\{u_i | u_i \text{ is the first child node of } v \text{ in } L_i, i \in [2, m]\}$. For each node $u \in S_1(v)$, if $u < c_{\text{max}}$, then $u$ is not a CA node.

**Proof:** Suppose $u$ is a CA node, then $u$ must appear in each LList as one of $v$’s child node. Since $u_i$ is $v$’s first child node in $L_i(i \in [2, m])$, we know that $u \geq c_{\text{max}}$. Therefore, if $u < c_{\text{max}}$, $u$ is not a CA node.

**Lemma 8:** Let $v$ be a CA node of $Q = \{k_1, k_2, ..., k_m\}$, $c_{\text{max}} = \max\{u_i | u_i \text{ is the first child node of } v \text{ in } L_i, i \in [2, m]\}$. If $c_{\text{max}} = \infty$, then $v$ is an SLCA node.

**Proof:** $c_{\text{max}} = \infty$ means that $\exists i \in [2, m]$, such that $v$ does not have child nodes in $L_i$, therefore, anyone of $v$’s child nodes in $L_1(v)$ cannot be a CA node, thus $v$ is an SLCA node.

Based on Lemmas 7, 8, and the new hash indexes in Fig. 6, we have the algorithm for SLCA computation, which we call as TDSLCA-HO+. TDSLCA-HO+ has the same time complexity as TDSLCA-H and TDSLCA-HO, but usually issues less number of hash probe operations in practice. During the processing, when we get the max hash value, i.e., $c_{\text{max}}$, for a CA node $v$ in isCA() function, we can then safely skip those ones in $L_1(v)$ with ID values less than $c_{\text{max}}$ by Lemma 7, or immediately skip all nodes of $L_1(v)$ if $c_{\text{max}} = \infty$ by Lemma 8.

**Example 8:** Continue Example 7. After finding that node 1 is a CA node, we find that for node 1, $c_{\text{max}} = 3$. Therefore, node 2 will be processed without hash probe operations according to Lemma 7. When processing node 3, we know that $c_{\text{max}} = \infty$, thus directly output node 3 as an SLCA node, and skip processing nodes 4 and 5 according to Lemma 8. As a comparison, to process $Q$, the number of hash probe operations used by TDSLCA-H, TDSLCA-HO and TDSLCA-HO+ are 5, 3 and 2, respectively.

### 7.3 The Optimized Algorithm for SLCA Semantics

Even though TDSLCA-HO reduces the number of hash probe operations over TDSLCA-H, we can make further optimization for SLCA computation.

We observe that according to Lemma 3, the hash value for each node in Figs. 4 (b), (c) and Fig. 5 (d) is useless for SLCA computation, since in this case, for each processed node $v$ in $L_1$, we only need to know whether $v$ appears in hash tables of other LLLists. Considering this problem, we propose the third hash indexes to accelerate SLCA computation. As shown in Fig. 6, for each node $v$ of $L_2$, we maintain in $H_2$ the ID value of its first child node. If $v$ does not have child nodes in $L_2$, then its hash value in $H_2$ is $\infty$. Based on the new hash indexes, when processing a node $v$ of $L_1$, we have the chance to avoid more unnecessary hash probe operations, as shown by Lemmas 7 and 8.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Dewey</th>
<th>IDList</th>
<th>IDList+ Hash Table</th>
<th>LLList</th>
<th>LLList+ Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMark</td>
<td>1.93GB</td>
<td>1.51GB</td>
<td>2.35GB</td>
<td>1.01GB</td>
<td>2.06GB</td>
</tr>
<tr>
<td>DBLP</td>
<td>1.08GB</td>
<td>887.1MB</td>
<td>1.53GB</td>
<td>591.2MB</td>
<td>1.23GB</td>
</tr>
<tr>
<td>Treebank</td>
<td>179.1MB</td>
<td>141MB</td>
<td>257.4MB</td>
<td>115.1MB</td>
<td>233.7MB</td>
</tr>
</tbody>
</table>

**TABLE 2:** Comparison of index sizes.
TABLE 3: Queries on XMark dataset. $\sum_{i=1}^{m} |L_i|$ denotes the sum of the length of all inverted lists of Dewey labels, $|L_m|$ denotes the length of the longest (shortest) inverted list of Dewey labels, $\sum_{i=1}^{m} |L_i|/|L_1|$ denotes the length of the longest (shortest) IDList/LList, $N_1$ denotes the sum of the lengths of all Dewey labels of $L_1$, $N_{total} = \sum_{i=CA(Q)} |S_i(v)|$ is the upper bound of the number of processed nodes by our methods, $|S|$ 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 | $\sum_{i=1}^{m} |L_i|$ | $\sum_{i=1}^{m} |L_i|/|L_1|$ | $\sum_{i=1}^{m} |L_i|$ | $\sum_{i=1}^{m} |L_i|/|L_1|$ | $|S|$ | $N_{total}$ | $|S|$ | $N_E$ | $R_E$ (%) | $Freq$ |
|-------|----------|-----------------|-----------------|-----------------|-----------------|-----|-------------|-----|-------------|---------|------|
| QX1   | villages,books | 1,290 | 829 | 4,300 | 461 | 2,149 | 2,634 | 471 | 231 | 9 | 1.95 |
| QX2   | baboon,patients,arizona | 1,975 | 742 | 3,689 | 382 | 1,335 | 1,294 | 1 | 4 | 1 | 0.26 |
| QX3   | cabbage,tissue,shocks,baboon | 2,088 | 742 | 3,689 | 366 | 1,809 | 2,734 | 376 | 178 | 9 | 2.46 |
| QX4   | shocks,necklace,cognition,cabbage,tissue | 2,041 | 596 | 2,996 | 200 | 1,091 | 1,739 | 210 | 149 | 9 | 4.5 |
| QX5   | female,order | 36,594 | 19,894 | 68,140 | 16,700 | 62,145 | 109,835 | 17,980 | 5,560 | 379 | 3.47 |
| QX6   | privacy,check,male | 85,960 | 36,300 | 101,095 | 18,428 | 59,883 | 98,832 | 2,353 | 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 | 699,688 | 1,538,902 | 249,844 | 34,322 | 34,189 | 11.22 |
| QX10  | date,lastname,emp | 1,112,760 | 457,231 | 1,229,698 | 304,969 | 574,800 | 2,333,169 | 126,507 | 54,294 | 43,792 | 14.36 |
| QX11  | incategory,keyword,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 | 2,248,752 | 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 |
| QX14  | cabbage,male,female | 38,688 | 19,894 | 68,140 | 366 | 1,809 | 2,734 | 376 | 118 | 9 | 2.49 |
| QX15  | male,female,keyword,incategory | 802,018 | 411,575 | 1,236,016 | 18,428 | 60,998 | 98,832 | 1,394 | 50,000 | 75 | 0.407 |
| QX16  | female,keyword,incategory,cabbage,food | 802,384 | 411,575 | 1,236,016 | 366 | 1,809 | 2,734 | 234 | 50,000 | 6 | 1.639 |

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:

(Gp 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 [16], [17], TDELCA and TDELCA-L; (Group 1.2) algorithms that are based on hash search, including HybELCA-HS [17], TDELCA-H and TDELCA-HO.

(Gp 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+ [16], [17], TDSLCA and TDSLCA-L; (Group 2.2) algorithms that are based on hash search, including HybSLCA-HS [17], TDSLCA-H, TDSLCA-HO and TDSLCA-HO+.

(Gp 3) Algorithms of this group target at LCA computation, including ALCA [5], TDLCA, TDLCA-L, TDLCA-H and TDLCA-HO.

Note that as we have made detailed performance comparison between existing methods in [17], we refer readers to [17] 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 [16], [17], i.e., (1) use the same datasets, including XMarks (582MB) and DBLP (876MB). After parsing the two datasets, Oracle Berkeley DB4 is used to store the keyword inverted lists by a hash file, where each key is a keyword $k$ and the value associated with $k$ is the inverted list of $k$; (2) test the same set of queries on the two datasets; (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 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.

Table 2 shows the index sizes without using any compression scheme, i.e., each number is stored as a 4-byte integer, where “Dewey” denotes inverted lists of Dewey labels used by ALCA and TD$x$LCA, “IDList” denotes inverted lists of node IDs used by BwdELCA, BwdSLCA+, “IDList + Hash Table” denotes the index used by HybELCA-HS and HybSLCA-HS, “LList” denotes the index used by TD$x$LCA-L, and “LList+Hash Table” the index used by TD$x$LCA-H, TD$x$LCA-HO and TDSLCA-HO+.

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2016.2516536, IEEE Transactions on Knowledge and Data Engineering
be verified according to the number of hash probe operations. The reason that TDELCA-H and TDELCA-HO use less hash probe operations lies in that both TDELCA-H and TDELCA-HO can avoid the VUN problem, while HybELCA-HS may waste hash probe operations on useless nodes. Moreover, TDELCA-HO can reduce more unnecessary hash probe operations over TDELCA-H, thus works better in most cases.

By comparing all algorithms of Group 1, we know that TDELCA and TDELCA-L (TDELCA-H and TDELCA-HO) work better than BwdELCA (HybELCA-HS) by avoiding both the CAR and VUN problems. For our methods, TDELCA-L, TDELCA-H and TDELCA-HO beat TDELCA for all queries. An interesting phenomenon is that by introducing hash indexes, TDELCA-H and TDELCA-HO can work better than TDELCA-L for queries QX1 to QX4, QX6, QX14 to QX16, with the cost of double size (see the last two columns of Table 2). Even though, they 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 and TDELCA-HO can only utilize the positional relationship between nodes of $L_1$ to achieve pruning. Moreover, the hash indexes store the global ID for each node, which makes it difficult, if applying existing compression schemes, to reduce the index size as much as LList used by the TDELCA-L algorithm (which can be constructed based on Dewey labels according to Corollary 1).

8.2.2 Scalability

Besides the 16 queries in Table 3, we generated 174,406 queries by combining the set of keywords we selected from XMark dataset. Fig. 8 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.

![Fig. 8: Running time with different result selectivities.](image)

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

---

6. Normalized time of each algorithm is defined as $(t_1 \times 100)/t_2$, where $t_1$ is the running time of this algorithm, $t_2$ is the running time of the first algorithm in Figs. 7-?.

7. The number of comparison operations can help us understand the performance variance in an in-depth way, which is denoted by $N_C$ and 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.
selectivity; (A2) fix the result selectivity and vary the number of keywords.

For A1, we have the following observations according to Fig. 8: (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-HO works best. Generally speaking, the lower (higher) the selectivity, the more significant the performance gain of TDELCA (HybELCA-HS, TDELCA-H and TDELCA-HO) on BwdELCA; and both TDELCA-L and TDELCA-HO work best for all cases on average.

For A2, we have the following observation: the performance gains of TDELCA-L, HybELCA-HS, TDELCA-H and TDELCA-HO over BwdELCA increase with the increase of keywords, while that of TDELCA only increases when the result selectivity is low.

### 8.2.3 Experimental Results on DBLP Dataset

We selected 10 queries for DBLP dataset (Table 4 in Appendix B). The experimental results are shown in Fig. 9, 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-H and TDELCA-HO beat 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. 9 we have similar result as that of Section 8.2.1, i.e., TDELCA-L works better than TDELCA for all queries, and for TDELCA-L, TDELCA-H and TDELCA-HO, no one can beat the other for all queries.

### 8.2.4 Experimental Results on Treebank Dataset

We selected 10 queries for Treebank dataset (Table 5 in Appendix B). The experimental results are shown in Fig. 10, from which we have the same observations as that of Section 8.2.3. However, by comparing Figs. 9 and 10, 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, TDELCA-H and TDELCA-HO can be done more efficiently compared with the binary search of BwdELCA on longer flattened IDList index.

For A1, we have the following observations according to Fig. 9: (1) TDELCA works much better than BwdELCA; for algorithms of Group 1.1, our methods are much more efficient than BwdELCA; for algorithms of Group 1.2, TDELCA-H and TDELCA-HO beat HybELCA-HS for all queries. The reason also lies in that our methods avoid both the CAR and VUN problems. 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 three 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.

### 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 three 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. 61472339, 61073040, 61572421, 61272124, 61170027). Wei Wang was partially supported by ARC DP130103401 and DP130103405. Jeffrey Xu Yu was partially supported by Council of the Hong Kong SAR, China No. 14209314 and 418512.

### References


Junfeng Zhou received the BS and MS degrees from Yanshan University, and the PhD degree from Renmin University of China. He is currently a professor in the School of Information Science and Engineering, Yanshan University. His research interests include information retrieval techniques, query processing, and optimization.

Wei Wang received the PhD degree in computer science from Hong Kong University of Science and Technology in 2004. He is currently an associate professor in the School of Computer Science and Engineering at the University of New South Wales, Australia. His research interests include integration of database and information retrieval techniques, similarity search, and query processing, and optimization.

Ziyang Chen received the BS, MS and PhD degrees from Yanshan University. He is currently a professor in the School of Information Science and Engineering, Yanshan University. His research interests include information retrieval techniques, query processing, and optimization.

Jeffrey Xu Yu received the BE, ME, and PhD degrees in computer science, from the University of Tsukuba, Tsukuba, Japan, in 1985, 1987, and 1990, respectively. He is currently a professor in the Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, Hong Kong. His current research interests include graph mining, graph database, social networks, keyword search, and query processing and optimization. He is a senior member of the IEEE, a member of the IEEE Computer Society, and a member of the ACM.

Xian Tang received the BS and MS degrees from Yanshan University, and the PhD degree from Renmin University of China. She is currently a lecture in the School of Economics and Management, Yanshan University. Her research interests include information retrieval techniques, query processing, and flash databases.

Yifei Lu was a PhD student at University of New South Wales, Australia. He received his Bachelor degree in Computer Science from Shanghai Jiao Tong University in China. His research interests include indexing and query processing techniques for database and information retrieval.

Yukun Li received the BS degree from Shandong University, MS degree from North China Electric Power University and the PhD degree from Renmin University of China. He is currently a senior engineer in the School of Computer and Communication Engineering, Tianjin University of Technology. His research interests include information retrieval techniques and web data management.