Fast ELCA Computation for Keyword Queries on XML Data

Rui Zhou
Faculty of Information and Communication Technologies
Swinburne University of Technology
Melbourne, VIC 3122, Australia
rzhou@swin.edu.au

Chengfei Liu
Faculty of Information and Communication Technologies
Swinburne University of Technology
Melbourne, VIC 3122, Australia
cliu@swin.edu.au

Jianxin Li
Faculty of Information and Communication Technologies
Swinburne University of Technology
Melbourne, VIC 3122, Australia
jianxinli@groupwise.swin.edu.au

ABSTRACT

Keyword search is integrated in many applications on account of the convenience to convey users' query intention. Recently, answering keyword queries on XML data has drawn the attention of web and database communities, because the success of this research will relieve users from learning complex XML query languages, such as XPath/XQuery, and/or knowing the underlying schema of the queried XML data. As a result, information in XML data can be discovered much easier.

To model the result of answering keyword queries on XML data, many LCA (lowest common ancestor) based notions have been proposed. In this paper, we focus on ELCA (Exclusive LCA) semantics, which is first proposed by Guo et al. and afterwards named by Xu and Papakonstantinou. We propose an algorithm named Hash Count to find ELCAs efficiently. Our analysis shows the complexity of Hash Count algorithm is $O(kd|S_1|)$, where k is the number of keywords, d is the depth of the queried XML document and $|S_1|$ is the frequency of the rarest keyword. This complexity is the best result known so far. We also evaluate the algorithm on a real DBLP dataset, and compare it with the state-of-the-art algorithms. The experimental results demonstrate the advantage of Hash Count algorithm in practice.

1. INTRODUCTION

Answering keyword queries on XML data has been extensively studied recently [8, 25, 15, 24, 22, 23, 17, 18, 13, 14, 5]. The driving force behind this investigation is the wide residence of XML data and the popularity of keyword search. Users are supported to query a large XML document using a set of keywords without knowing complex XML query languages (such as XPath, XQuery) and/or being aware of the underlying document structure. As a result, keyword queries provide significant convenience for general users to find information they are interested in.

To model the result of keyword queries on XML documents, a family of LCA (Lowest Common Ancestor) based notions have been proposed, such as, ELCA (Exclusive LCA) [8], SLCA (Smallest LCA) [24], MLCA (Meaningful LCA) [15] and Interconnection Relationship [5]. Other LCA-based query result semantics rely more or less on SLCA or ELCA by either imposing further conditions on the LCA nodes [14] or refining the subtrees rooted at the LCA nodes [17, 18, 13]. Among these LCA semantics, ELCA is believed to be one of the most influential semantics, because it can capture the highest number of possible results for a query. It is true that the more does not mean the better. However, due to the flexibility of keyword query, it is often the case that user's intention is difficult to infer, and besides data may be structured differently, it is unlikely to find out an ideal semantics as the most reasonable one, and hence a semantics capturing more results could be a better choice. Therefore, in this paper, we mainly focus on ELCA semantics.

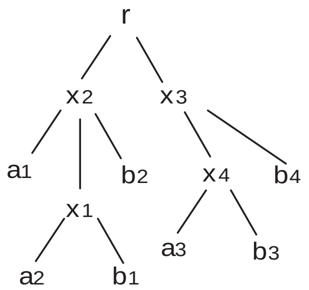


Figure 1: Examples of ELCA and SLCA

For ease of understanding, we illustrate the concept of ELCA and its counterpart, SLCA, using an example in Fig. 1. Formal definitions will be given in Section 2. Other LCA semantics will be introduced later in Section 5 in a nutshell for the sake of not distracting readers from the main issue of this paper.

EXAMPLE 1. *Consider the XML tree shown in Fig. 1, where keyword nodes are annotated with subscripts. Consider a keyword query using keywords $\{a, b\}$, based on the conventional LCA semantics, $\{x_1, x_2, x_3, x_4, r\}$ is produced as the result, for x_1 is the LCA of $\{a_2, b_1\}$, x_2 is the LCA of $\{a_1, b_2\}$[1], x_3 is the LCA of $\{a_3, b_4\}$, x_4 is the LCA of $\{a_3, b_3\}$ and r is the LCA of $\{a_1, b_4\}$. While using SLCA*

[1] Here x_2 is also the LCA of $\{a_1, b_1\}$ or $\{a_2, b_2\}$.

EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.
Copyright 2010 ACM 978-1-60558-945-9/10/0003 ...$10.00

semantics, only $\{x_1, x_4\}$ is given as the result, because x_2, x_3 and r are ancestors of x_1 or ancestors of x_4 or both. (SLCA semantics requires the result LCAs to be lowest so that the query result is a set of tightest XML fragments containing all the keywords.) Using ELCA semantics, the result set is $\{x_1, x_2, x_4\}$. Here, each ELCA node has its own "witness" nodes, e.g. x_1 is witnessed by $\{a_2, b_1\}$, x_2 is witnessed by $\{a_1, b_2\}$, and x_4 by $\{a_3, b_3\}$. Compared with SLCA semantics, x_2 is additionally identified as a result. This point of view is reasonable, because, other than "witness" nodes $\{a_2, b_1\}$ which have already been covered by x_2's child x_1, x_2 has its own contributors $\{a_1, b_2\}$.

ELCA semantics helps users find more reasonable results, but also brings the challenge of computing them. Unlike well-studied SLCA computation in [24, 22, 23], the performance of ELCA computation algorithms is less satisfactory. Dewey Inverted List (DIL) algorithm, proposed in [8], is the pioneer work of computing ELCAs. It is not complex, but suffers from scanning all the inverted lists of query keywords. Here, an inverted list of a keyword is a list of elements in the XML document that directly contain the keyword, eg. the inverted lists for keyword a and b on the tree in Fig. 1 is $\{a_1, a_2, a_3\}$[2] and $\{b_1, b_2, b_3, b_4\}$ respectively. The complexity of DIL algorithm is $O(kd|S|)$, where k is the number of keyword, d is the depth of document tree, $|S|$ is the total length of all keyword inverted lists. If the query keywords contain some "stop words" (whose inverted lists are very long), the performance of DIL algorithm will degrade dramatically. Another method, Indexed Stack (IS) algorithm [25], uses the shortest keyword inverted list as the working list, and first generates a set of ELCA candidates, and then verifies them. It avoids scanning all the inverted lists, but still needs to access non-working lists by logarithmic indexed lookups. The complexity is $O(kd|S_1|log|S_{max}|)$, where $|S_1|$ ($|S_{max}|$) is the length of the shortest (longest) keyword inverted list. (During implementation, if all the inverted lists are stored together as a single list, then $|S_{max}| = |S|$.) The $log|S_{max}|$ factor is for discovering connections between matched keywords. The merit of IS algorithm is that, in real applications, $|S_1|log|S_{max}|$ could be smaller than $|S|$, since keyword frequencies are often biased. While, in this paper, we will show the $log|S_{max}|$ factor can be further saved, resulting in an $O(kd|S_1|)$ algorithm.

The aim of this paper is, to introduce a (both theore -tically and experimentally) *efficient* algorithm that computes the ELCA nodes for a set of keywords over an XML document.

The key idea of our algorithm is to record the number of keyword instances appearing under a node, and use such information to judge whether an ELCA candidate is a real ELCA node. To be specific, we compare the number of keyword instances that a candidate node has covered with the total number of instances that the candidate node's children who contain all the keywords have covered. If, for each keyword, there exists an instance solely contributing to the candidate, then the candidate is a real ELCA node. For example, in Fig. 1, node x_2 contains two keyword a instances $\{a_1, a_2\}$ and two keyword b instances $\{b_1, b_2\}$. We denote the count of keyword a, b under x_2 as $C_{x_2}(a, b) = (2, 2)$. The only child of x_2 containing all the keyword is x_1, and $C_{x_1}(a, b) = (1, 1)$, for x_1 only covers a_2 and b_1, one instance for each keyword. $C_{x_2}(a, b)$ is larger than $C_{x_1}(a, b)$ on every dimension, and therefore x_2 is identified as an ELCA node, since it has a proprietary contributor for each keyword. Note that, if x_2 has more children containing all the keywords, the keyword counts of these children should be added together before compared with $C_{x_2}(a, b)$. Similarly, we have $C_{x_3}(a, b) = (1, 2)$, $C_{x_4}(a, b) = (1, 1)$ and x_4 is the only child of x_3 containing all the keywords. Comparing $C_{x_3}(a, b)$ with $C_{x_4}(a, b)$, x_3 does not have any proprietary keyword a instance, and hence is not an ELCA node. Keyword count information can be stored in a hash index, giving an $O(1)$ look-up time. The details of our algorithm, including index build-up, ELCA candidate generation, ELCA candidate verification, and optimized techniques, will be introduced in the main section of this paper.

We summarize our contributions as follows:

- We propose an efficient algorithm, named Hash Count, to find ELCAs to answer keyword queries on XML data. We show the complexity of the algorithm is $O(kd|S_1|)$, which is the best complexity result among existing works.

- We conduct a series of experiments to compare the performance of Hash Count algorithm with existing algorithms (Dewey inverted list algorithm and Indexed Stack algorithm) on real XML dataset. The experimental study shows our Hash Count algorithm outperforms the existing ones in almost every case.

Here is a roadmap of this paper. In Section 2, we introduce a formal definition of ELCA, along with some notations used in this paper. We also introduce two existing algorithms for computing ELCA nodes. In Section 3, we give out the Hash Count algorithm. Two versions of the algorithm will be proposed, a naive one and an optimized one. Experimental results are shown in Section 4. Related works and conclusion are given in Section 5 and Section 6 respectively.

2. PRELIMINARIES

In this section, we first introduce ELCA query semantics, and then review the state-of-the-art works on how to locate ELCA nodes. We also point out the disadvantages of the current works.

2.1 ELCA semantics

We model XML documents as trees using the conventional labeled ordered tree model, and cross references are not considered in this paper. Each node of an XML tree corresponds to an XML element, an attribute or a text string. The leaf nodes are all text strings. A keyword may appear in element names, attribute names or text strings. If a keyword w appears in the subtree rooted at a node n, we say the node n contains keyword w. If w appears in the element name or attribute name of n, or w appears in the text value of n when n is a text string, we say node n directly contains keyword w.

A keyword query on an XML document often asks for an XML node that contains all the keywords, therefore, for large XML documents, indexes are often built to record

[2] The inverted list often stores a set of encoded numbers, because nodes in the XML are often encoded by some encoding scheme, such as Dewey Encoding.

which nodes directly contain which keywords in order to speed up keyword query processing. For example, for a keyword w_i, all nodes directly contain w_i are stored in a list S_i (called inverted list) and can be retrieved altogether at once. Each item in S_i is an encoded number (eg. using *preorder* encoding scheme) or an encoded object (eg. using Dewey encoding scheme). We always use S_1 to denote the shortest inverted list throughout this paper[3].

We follow the definitions in [25] on formalizing the semantics of LCA and ELCA. We introduce some notions first. Let $v \prec_a u$ denote v is an ancestor node of u, and $v \preceq_a u$ denote $v \prec_a u$ or $v = u$. The function $lca(v_1, \dots, v_k)$ computes the Lowest Common Ancestor (LCA) of nodes $v_1, \dots, v_k$. The LCA of sets $S_1, \dots, S_k$ is the set of LCAs for each combination of nodes in S_1 through S_k.

$$lca(w_1, \dots, w_k) = lca(S_1, \dots, S_k) = \\ \{lca(n_1, \dots, n_k) | n_1 \in S_1, \dots, n_k \in S_k\}$$

Given k keywords $\{w_1, \dots, w_k\}$ and their corresponding inverted lists $S_1, \dots, S_k$ of an XML tree T, the Exclusive LCA of these keywords on T is defined as:

$$elca(w_1, \dots, w_k) = elca(S_1, \dots, S_k) = \\ \{v | \exists n_1 \in S_1, \dots, n_k \in S_k (v = lca(n_1, \dots, n_k) \wedge \\ \forall i \in [1, k] \ \nexists x (x \in lca(S_1, \dots, S_k) \wedge child(v, n_i) \preceq_a x))\}$$

where $child(v, n_i)$ denotes the child node of v on the path from v to n_i. The meaning of a node v to be an ELCA is: v should contain all the keywords in the subtree rooted at v, and after excluding v's children which also contain all the keywords from the subtree, the subtree still contains all the keywords. In other words, for each keyword, node v should have its own contributor. With the above definition, readers are now clear on why the ELCAs in Fig. 1 are $\{x_1, x_2, x_4\}$. We now give another example to further illustrate the concept of ELCA, to avoid readers being misled.

EXAMPLE 2. *To answer keyword query $\{a, b\}$ on the XML tree in Fig. 2, obviously, $\{x_3, x_5\}$ are ELCA nodes. After excluding the subtrees rooted at x_3 and x_5, x_1 still has its own contributors a_1, b_3, but here x_1 is not an ELCA node, because a_1 and b_3 are screened by x_2 and x_4 respectively, for x_2 and x_4 both contain all the keywords. Therefore, the ELCAs for keyword $\{a, b\}$ on the tree are $\{x_3, x_5\}$.*

Considering the above example, some readers may prefer x_1 as a query result. This will bring in a new semantics similar to the current ELCA semantics. We stress that our focus of this paper is not to reason which semantics is more reasonable, but simply to concentrate on efficiently finding the ELCAs according to the original semantics. We point out that our algorithm and existing algorithms (DIL and IS) can be altered to cater to other similar semantics (like the above one).

Note that the definition of ELCA first appeared in [8]. We use the one in [25] for the convenience of presentation. Of course, these two definitions are equivalent. We also borrowed some other notions from [25], which will be introduced later in this paper when necessary.

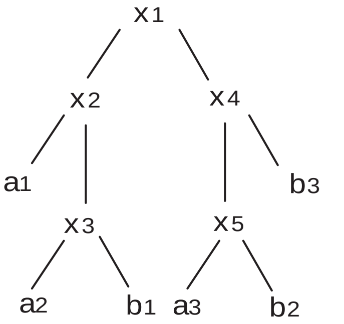


Figure 2: x_1 is not an ELCA

2.2 Current works on ELCA computation

XRank [8] is the first work presenting ELCA semantics. It is also one of the first several works to model keyword query result over XML documents. The Dewey Inverted List (DIL) algorithm proposed in [8] is the first algorithm to evaluate ELCA nodes. In DIL, k keyword inverted lists are accessed in sorted-merge order, and a stack is used to record the current LCA node. Once a new node is read from some list, the stack is updated with or without producing ELCA nodes progressively. To comment on this algorithm, DIL takes pride in its simplicity, but suffers from low efficiency, especially when some keyword lists are found to be very long (Such keywords are referred as "stop words".), because it has to scan to the end of all the inverted lists. The complexity of DIL algorithm is $O(kd|S|)$, where $|S|$ is total size of all the inverted lists.

Indexed Stack (IS) Algorithm [25] outperforms XRank by avoiding scanning all the lists. It takes the shortest list as the working list (corresponding to the rarest keyword), and computes the SLCA for each node in the working list with nodes in other lists to compromise an ELCA candidate set, denoted as $\bigcup_{v_1 \in S_1} slca(\{v_1\}, S_2, \dots, S_k)$. Here, $slca(\{v_1\}, S_2, \dots, S_k)$ is defined as follows:

$$slca(\{v_1\}, S_2, \dots, S_k) = \\ \{v | v \in lca(\{v_1\}, S_2, \dots, S_k) \wedge \\ \forall v' \in lca(\{v_1\}, S_2, \dots, S_k)(v \not\prec_a v')\}$$

For example, for the given keywords $\{a, b\}$, the ELCA candidate set $\bigcup_{v_1 \in S_a} slca(\{v_1\}, S_b)$ on the tree in Fig. 1 is $\{x_1, x_2, x_4\}$, for $\{x_2\} = slca(\{a_1\}, S_b)$, $\{x_1\} = slca(\{a_2\}, S_b)$ and $\{x_4\} = slca(\{a_3\}, S_b)$. Similarly, the candidate set on the tree in Fig. 2 is $\{x_2, x_3, x_5\}$. And then, those ELCA candidates are verified to pick out the real answers. In Fig. 1, $\{x_1, x_2, x_4\}$ are all real ELCAs, while in Fig. 2, only $\{x_3, x_5\}$ are real ELCAs. In the IS algorithm, the working list is accessed sequentially, and other lists are accessed randomly by index. The number of ELCA candidates is not larger than the number of nodes in the shortest keyword list. The complexity of IS algorithm is $O(kd|S_1|log|S_{max}|)$. The unperfect aspect of IS algorithm is that, given an ELCA candidate v, for each keyword w_i, it always attempts to find an *exact* node in S_i to "witness" v. In fact, we can judge whether v is a real ELCA without knowing the exact witness node, as long as we know there exists such a witness node. This point motivates our Hash Count algorithm.

[3] In this paper and also in the previous work [25], "frequency" is used to denote the length of a inverted list. The referred frequency means the number of distinct nodes that directly contain a certain keyword. It is not the real frequency of the keyword, eg. a keyword may appear several times in a leaf text string, but will be counted only once.

3. ELCA EVALUATION

In this section, we introduce the Hash Count algorithm to find the ELCA nodes. We will firstly give a brief idea of the algorithm in Section 3.1, followed by the hash indexes we need to build to serve our algorithm in Section 3.2. Then we give a naive version of the algorithm, which is a straightforward implementation of our idea in Section 3.3 and 3.4. Finally, we present an advanced version of Hash Count algorithm armed with some optimization techniques.

3.1 Outline of the Hash Count algorithm

Our Hash Count algorithm can be divided into two subtasks: (I) firstly, find out ELCA candidates (nodes that are possible to be ELCA nodes); (II) verify these candidates, discard the false positives and obtain the real results. Note that this framework is the same as the Indexed Stack algorithm [25], but techniques used are different. To be specific, our ELCA candidate definition is different, and even more simple, resulting in an easy step of candidate generation. On the other hand, our verification step takes advantage of an important observation and is more efficient.

As to subtask (I), the ELCA candidate set is defined as $\{v|v \in T \wedge \exists v_1(v_1 \in S_1 \wedge v \preceq_a v_1)\}$, where T denotes all the nodes in the XML tree. The above expression means a node is possible to be an ELCA only if it contains the least frequent keyword directly or indirectly. According to our definition, the number of ELCA candidates is bounded by $d|S_1|$, where d is the depth of the tree, $|S_1|$ is the frequency of the rarest keyword. Readers may notice that the requirement for a node to be an ELCA candidate is not strict, leading to a possible large candidate size. In fact, the upper bound of the candidate size, $d|S_1|$, is usually not very large, because there often exists a "selective" keyword, i.e. $|S_1|$ is relatively small, and XML documents are not extremely deep. In this candidate generation step, we aim to inspect each ELCA candidate (verified whether it is a real ELCA) at most once. This is done by maintaining a stack. Details will be explained in Section 3.3.

As to subtask (II), we verify an ELCA candidate v by comparing how many keyword instances v has covered with how many keyword instances have already been covered by some node u, where u is a child of v and u itself has contained all the keywords. The observation is that, if, for each keyword, the candidate v has some instance solely contributing to v itself, but not contributing to any child u (which contains all the keywords) under v, we can draw the conclusion that v is a real ELCA node. This part will be elaborated in Section 3.4.

3.2 Index Build-up

To better illustrate our algorithms, we introduce some indexes first. In these indexes, we identify each node of the XML tree with its preorder id (obtained by a preorder traversal of the tree). We stress that our algorithm also works with other numbering methods, such as postorder numbering or Dewey encoding, as long as each node in the tree can be uniquely identified. The indexes have three parts:

- A hash table called *Frequency Table* stores for each keyword the frequency of this keyword in the XML tree. It is used to find the least frequent keyword, i.e. to choose the working inverted list.

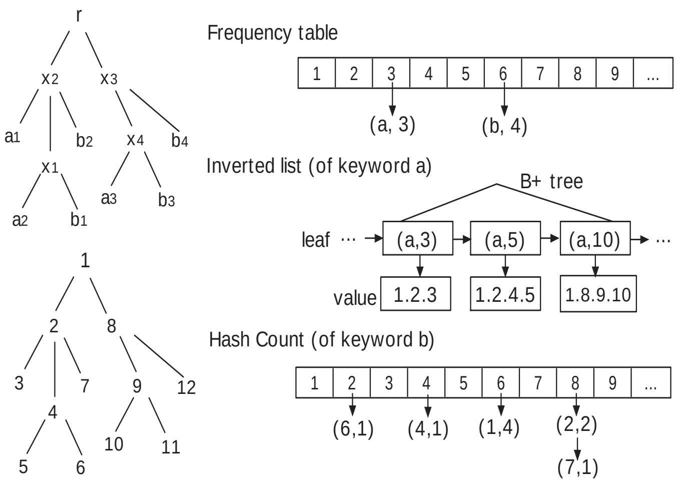


Figure 3: Examples of Indexes

- A disk-based B+ tree stores all the nodes in the XML tree that directly contain a keyword. These nodes are first grouped by the contained keyword, and then ordered by their preorder ids, i.e. the key of the B+ tree is a combined object (*keyword*, *preorder_id*), and the comparison operator is overloaded by comparing the *keyword* field and *preorder_id* field in turn. The B+ tree is often referred as *Inverted Lists* in the previous works. For each keyword, the size of its inverted list is the number of nodes directly containing that keyword. While in our B+ tree index, we also have a value field for each leaf node in the B+ tree. The value field also stores nodes that indirectly contain a keyword. Example 3 will make this point clear.

- For each keyword, we have a hash table (named *Hash Count Index*) to store the nodes in the XML tree that directly or indirectly contain that keyword, and how many keyword instances those nodes contain. For instance, given a keyword w_i, the Hash Count Index of w_i stores a set of pairs of the form (*preorder_id*, *count*), where *preorder_id* is the node which directly or indirectly contains keyword w_i, and *count* is the number of w_i's occurrences under that node. Here, *preorder_id* is the key of the Hash Count Index. We also give an example below.

EXAMPLE 3. *In Fig. 3, a preorder encoding of the XML tree in Fig. 1 is shown on the left side. Three types of index are shown on the right side. Frequency table stores frequencies* 3, 4 *for keyword a and b. Below it shows a portion of the inverted list with nodes directly containing keyword a. In the figure, such nodes are* $\{a_1, a_2, a_3\}$*, and are numbered with* $\{3, 5, 10\}$*. Each leaf node has a value field storing the node ids on the path from the leaf node to the document root, eg.* $(a, 3)$ *has the value* 1.2.3*, in which* 2 *is the parent of* 3*,* 1 *is the parent of* 2*. Nodes* 1*,* 2 *contain the keyword a indirectly. The usage of the value field will be explained in our algorithms in the next section. The Hash Count index of keyword b is then given, eg. node* 4*,* 6*,* 7 *each contains one keyword instance (* 4 *and* 6 *contain* b_1*,* 7 *contains* b_2*), denoted as* $(4, 1)$*,* $(6, 1)$*,* $(7, 1)$*; node* 2 *contains two keyword*

Algorithm 1 Naive Hash Count

```
Input: the shortest keyword inverted list, $S_1$
Output: all the ELCA nodes, $R$
 1: $R = \phi$;
 2: $stack$ = empty;
 3: while not end of $S_1$ do
 4:   Read a node value $v$ from $S_1$;
 5:   $p = lcp(stack, v)$; {find the longest common prefix $p$
      such that $stack[i].node = v[i]$, $1 \leq i \leq p$}
 6:   while $stack.size > p$ do
 7:     $popEntry = stack.pop()$;
 8:     if $isELCA(popEntry)$ then
 9:       $R := R \cup \{popEntry.node\}$;
10:       add $popEntry.node$ to $stack.top().childList$;
11:     else
12:       if $popEntry.childList \neq \phi$ then
13:         add $popEntry.node$ to $stack.top().childList$;
14:       end if
15:     end if
16:   end while
17:   for $p < j \leq v.length$ do
18:     $newEntry = [node := v[j]; childList := \phi]$;
19:     $stack.push(newEntry)$;
20:   end for
21: end while
22: while $stack$ is not empty do
23:   Repeat line 7 to line 15;
24: end while
```

instance $\{b_1, b_2\}$*, recorded as* $(2, 2)$*; node* 1 *contains four keyword instance* $\{b_1, b_2, b_3, b_4\}$*, represented as* $(1, 4)$*. These pairs are stored in the Hash Count index of keyword b. Note that* $(8, 2)$*,* $(9, 1)$*,* $(11, 1)$ *and* $(12, 1)$ *are also in the index, but not displayed in the figure.*

The indexes can be built in advance by one parse of an XML document. When a set of keywords is given, we use the frequency table to pick a least frequent keyword, and read node values from the inverted list of the least frequent keyword to generate a set of ELCA candidates. We also load into memory the Hash Count indexes of other keywords to verify whether an ELCA candidate is a real ELCA node. The case when the Hash Count indexes cannot fit into memory will be discussed in Section 3.5.5.

3.3 ELCA Candidate Generation

In this section, we introduce how to inspect each ELCA candidate only once. The key idea is, when we decide to verify a node v, we have done the verifications of all the ELCA candidates that are descendants of node v.

A naive Hash Count algorithm is given in Algorithm 1 (We will give a sophisticated optimized version in Section 3.5.). We will first go through the algorithm, and then give a running example to illustrate the algorithm.

Line 1 initializes the final ELCA set R to be an empty set. Line 2 creates an empty stack. Each stack entry encapsulates an ELCA candidate, and has two fields (See Fig. 4). One field *entry.node* stores the node id of the ELCA candidate, the other field *entry.childList* stores the ids of the candidate's children who also contain all the keywords. The field *entry.childList* is used to verify whether *entry.node* is a real ELCA candidate (This step line 8 will be introduced in Algorithm 2 in the next section). For each item in S_1 (line 3-21), the algorithm will first pop up and verify the nodes that the algorithm will not see descendants of those nodes in future (line 6-16), and then push some new nodes into the stack (line 17-20). Line 5 is to find how many nodes need to remain in the stack, i.e. how many nodes need to be popped. A node is added into *stack.top*()'s *childList* either when the node is a real ELCA, or when the node's *childList* is not empty. Both cases imply that the node contains all the keywords. Line 22-24 is to verify the remaining nodes in the stack.

Fig 4 shows the snapshots of running the algorithm on the tree in Fig 1. We divide the process into four steps. In each step, a new value is read from the inverted list of keyword a. At the beginning, the stack is empty, and every ELCA candidate is encapsulated into a stack entry, and pushed into the stack. The *childList* field of the stack entry is initially set to empty (represented as {} in the figure), eg. (1,{}), (2,{}), (3,{}) are pushed into the stack in turn. When a_2 = 1.2.4.5 arrives, we first obtain the longest common prefix of a_2 and the stack to be 1.2, and then pop up and verify the nodes do not belong to the common prefix. Node 3 is popped and verified, followed by nodes 4 and 5 pushed into the stack in turn. Step 3 is similar to step 2. The difference is that in step 3, a real ELCA node 4 has been discovered, therefore when node 4 is popped from the stack, it is added into stack.top()'s *childList*. The current stack top entry becomes (2,{4}). Node 4 will be used to verify whether node 2 a real ELCA. Of the same spirit, when node 2 is popped from the stack, it is added into node 1's *childList*. Finally, the remaining nodes in the stack are all popped up and verified.

3.4 ELCA Candidate Verification

In this section, we present how to determine whether an ELCA candidate is a real ELCA node. We will first review the current solution in Indexed Stack algorithm to extract the hidden idea implied by that implementation, and afterwards we propose our solution, which is easier to implement, superior in theoretical complexity, and faster in practice (We only provide theoretical analysis in this section, and performance on real DBLP dataset will be shown later in Section 4).

Let us recall a concept first. Let $child\_elcaCan(v)$ denote the set of children of v that contain all keyword instances, i.e. for any $u \in child\_elcaCan(v)$, u is a child of v and either u or u's descendant is an ELCA candidate node.

$$child\_elcaCan(v) = \{u | u \in child(v) \wedge \\ \forall i \in [1, k](\exists x_i \in S_i(u \preceq_a x_i))\}$$

where $child(v)$ is the set of child nodes of v. Now assume the set $child\_elcaCan(v)$ is $\{u_1, \ldots, u_c\}$, where $c = |child\_elcaCan(v)|$. See Fig. 5, to verify whether v is an ELCA node, the previous solution is to probe every S_i to see if there exists a node $x_i \in S_i$ such that either x_i is in the forest under v to the left of the path vu_1, or in the forest under v to the right of the path vu_c, or in any forest under v between the paths vu_j and vu_{j+1} ($j \in [1, c-1]$) [25].

The idea behind this strategy is, for each keyword w_i, we need to find at least one witness x_i, which solely contributes to v (does not contribute to any $u \in child\_elcaCan(v)$). As aforementioned, there are $c + 1$ different portions under v

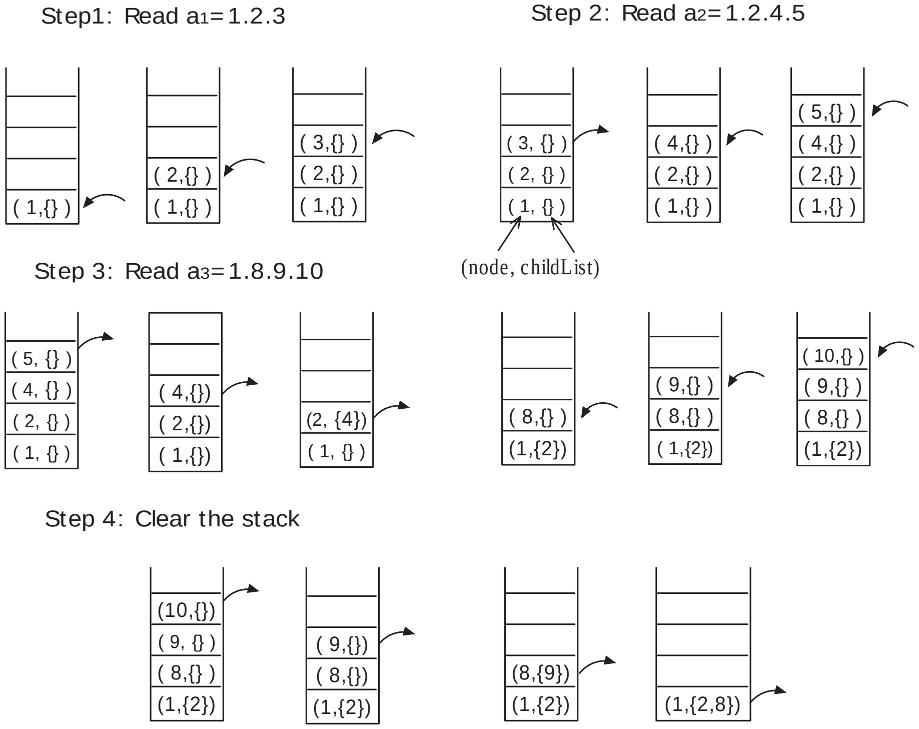


Figure 4: The snapshots of Naive Hash Count algorithm

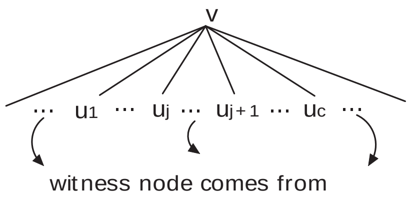


Figure 5: v **and** $child\_elcaCan(v)$

where x_i may come from, however, if v is a real ELCA node, it does not matter which portion x_i comes from, as long as it resides in one of them. Based on the above observation, we have the following lemma.

LEMMA 4. *Given an XML tree and a keyword query* $\{w_1, \ldots, w_k\}$*, a node* v *is a real ELCA node, if and only if, for any* $i \in [1, k]$,

$$C_i(v) > \sum_{u \in child\_elcaCan(v)} C_i(u) \qquad (1)$$

where, for a node n *in the XML tree,* $C_i(n)$ *denote, in the subtree rooted at node* n*, the number of nodes which directly contain keyword* w_i.

EXAMPLE 5. *See Fig. 3,* $child\_elcaCan(x_2) = \{x_1\}$*. For keyword* a*,* $C_a(x_2) = 2$*, for* x_2 *covers two keyword* a *instances* $\{a_1, a_2\}$*. And* $\sum_{u \in child\_elcaCan(x_2)} C_a(u) = C_a(x_1) = 1$*, we have* $C_a(x_2) > \sum_{u \in child\_elcaCan(x_2)} C_a(u)$*. Similarly, for keyword* b*, we also have* $C_b(x_2) > \sum_{u \in child\_elcaCan(x_2)} C_b(u)$*. Therefore,* x_2 *is a real ELCA. While* x_3 *is not a real ELCA, because* $C_a(x_3) = \sum_{u \in child\_elcaCan(x_3)} C_a(u) = 1$.

The proof of Lemma 4 is not difficult, and hence is omitted. Lemma 4 implies an efficient implementation to check whether an ELCA candidate is a real ELCA node. A straightforward implementation of the function isELCA() is shown in Algorithm 2. By utilizing hash index, fetching $C_i(u)$ (line 4) can be done in $O(1)$ time, and the complexity of one verification step (calling the function isELCA() once) is $O(k|entry.childList|)$. The total time complexity of Algorithm 1 is $O(k\Sigma_{entry \in stack}|entry.childList|)$. Here, $\Sigma_{entry \in stack}|entry.childList|$ is bounded by $d|S_1|$, because $\Sigma_{entry \in stack}|entry.childList|$ is bounded by the number of nodes ever pushed into the stack (refer to line 10 and line 12-14), and the total number of nodes pushed into the stack is further bounded by $d|S_1|$. To sum up, the time complexity of Naive Hash Count algorithm is $O(kd|S_1|)$. Comparing to Indexed Stack algorithm of complexity $O(kd|S_1|log|S_{max}|)$, our solution manages to save a $log|S_{max}|$ factor, which could be of substantial importance when one or some of the keywords have very long inverted lists.

3.5 Optimized Hash Count Algorithm

The Naive Hash Count algorithm presents the key idea of finding ELCA nodes using the number of keyword occurrences rather than physically searching for a keyword witness. There are also several important optimizations that can dramatically affect the performance of the algorithm. We now introduce the optimizations, and give out an optimized Hash Count algorithm in Algorithm 3. The modified parts in Algorithm 3 (compared to Algorithm 1) will be explained when we introduce the corresponding optimization

Algorithm 2 isELCA(*entry*) in Naive Hash Count

```
Description: To verify whether an ELCA candidate is a
real ELCA node, if yes, return true; otherwise return false.
Input: entry, a stack entry encapsulating an ELCA candidate node;
Output: a boolean value;
 1: for $i = 1$ to $k$ do
 2:    $C_{sum} := 0$;
 3:    for each $u$ in $entry.childList$ do
 4:       Look up $C_i(u)$ in Hash Count Index of keyword $w_i$;
 5:       $C_{sum} + = C_i(u)$;
 6:    end for
 7:    Look up $C_i(entry.node)$ in Hash Count Index of keyword $w_i$;
 8:    if $C_i(entry.node) = C_{sum}$ then
 9:       return false;
10:    end if
11: end for
12: return true;
```

techniques.

3.5.1 Reducing isELCA() Calls

In the Naive Hash Count algorithm, for every node in S_1, all its ancestor nodes will be checked by the function isELCA(). In fact, many of these nodes do not need to go into isELCA(). The reason is every keyword witness node could contribute to only one ELCA node. For example, in Fig. 3, a_1 contributes to x_2, a_2 contributes to x_1 and a_3 contributes to x_4. After x_1 and x_4 are found to be ELCAs, x_1 and x_4 will prevent keyword instances below them from contributing to their parents according to the ELCA definition. As a result, we use the notion *potential ELCA* to capture those nodes in the stack that need to be checked in isELCA(). Potential ELCA is defined as follows:

Definition 1. If node v is an ELCA candidate and contains at least one witness node x_1 from S_1 (the working list), satisfying x_1 does not contribute to any ELCA nodes under v, we call node v a potential ELCA.

Example 6. *In Fig 3, when x_2 is pushed into the stack, it is not a potential ELCA. After a_1 is verified not to be an ELCA, x_2 is set to be a potential ELCA, for there exists a keyword instance a_1 not covered below x_2. A potential ELCA may or may not be a real ELCA. In the example, the potential ELCA x_2 is a real ELCA. But if we remove b_2 node from the tree, x_2 becomes a false one, despite that it is still set to be a potential ELCA after a_1 is popped from the stack.*

For each stack entry, we add a boolean value to identify whether the entry is a potential ELCA. For a popped entry from the stack, if $entry.isPotenialELCA = false$, we discard the entry immediately without calling $isELCA(entry)$ (line 8 in Algorithm 3). Line 13 sets the stack top to be a potential ELCA, if the current node is a potential ELCA, but not a real ELCA, because it means the current node has an uncovered keyword instance that may contribute to its parent. Line 25 initializes the $isPotentialELCA$ field for every new stack entry to be false, except for the last pushed one (line 28). The last pushed entry encapsulates a node directly containing the keyword, and if the node is found not containing other keywords, the $isPotentialELCA$ property will be propagated upward (recall line 13). Comparing with Algorithm 1, we avoid calling isELCA() function for nodes $\{x_3, r\}$.

Algorithm 3 ELCA Evaluation with Optimization

```
Input: the shortest keyword inverted list, $S_1$
Output: all the ELCA nodes, $R$
 1: $stack$ = empty;
 2: $R = \phi$;
 3: while not end of $S_1$ do
 4:    Read a node value $v$ from $S_1$;
 5:    $p = lcp(stack, v)$; {find the longest common prefix $p$
       such that $stack[i].node = v[i]$, $1 \leq i \leq p$}
 6:    while $stack.size > p$ do
 7:       $popEntry = stack.pop()$;
 8:       if $popEntry.isPotentialELCA = true$ then
 9:          if $isELCA(popEntry)$ then
10:             $R := R \cup \{popEntry.node\}$;
11:             add $popEntry.selfCount[]$ to
                $stack.top().descCount[]$;
12:          else
13:             $stack.top().isPotentialELCA = true$;
14:             if $popEntry.descCount[] \neq \vec{0}$ then
15:                add $popEntry.selfCount[]$ to
                   $stack.top().descCount[]$;
16:             end if
17:          end if
18:       else
19:          if $popEntry.descCount[] \neq \vec{0}$ then
20:             fetch $popEntry.selfCount[]$ and
                add $popEntry.selfCount[]$ to
                $stack.top().descCount[]$;
21:          end if
22:       end if
23:    end while
24:    for $p < j \leq v.length$ do
25:       $newEntry = [node := v[j]; isPotenialELCA :=$
          $false; selfCount[] := \vec{0}; descCount[] := \vec{0}]$;
26:       $stack.push(newEntry)$;
27:    end for
28:    $stack.top().isPotentialELCA = true$;
29: end while
30: while $stack$ is not empty do
31:    Repeat line 7 to line 22;
32: end while
```

3.5.2 Reducing Index Look-ups

Algorithm 2 in the Naive Hash Count algorithm is a straightforward implementation of Lemma 4. For some node u and some keyword w_i, $C_i(u)$ may be looked up twice in line 4 (when checking whether u is a real ELCA) and line 7 (when checking whether u's parent is a real ELCA). Therefore, to avoid repeated look-ups, for each stack entry $entry$, we use a k-dimension vector to record the occurrences of each keyword under $entry.node$ (There are in total k keywords.). In this way, each node is looked up only once. We use $selfCount$ to denote the k-dimension vector. This programming trick is indeed simple, but will facilitate space cost reduction for Algorithm 1, which will be illustrated in the next section.

Algorithm 4 isELCA($entry$) in Optimized Hash Count

Description: To verify whether an ELCA candidate is a real ELCA node, if yes, return true; otherwise return false.
Input: $entry$, a stack entry encapsulating an ELCA candidate node;
Output: a boolean value;

```
1: for  $i = 2$ to $k$  do
2:     Look up $C_i(entry.node)$ in Hash Count Index of keyword $w_i$;
3:     $entry.selfCount[i] := C_i(entry.node)$;
4:     if  $entry.descCount[i] = C_i(entry.node)$  then
5:         return false;
6:     end if
7: end for
8: return true;
```

3.5.3 Reducing Space Cost

We now introduce how to reduce space cost for Algorithm 1. The idea is to replace the child lists with another k-dimension vector, $descCount$.

We first analyze the space complexity of the Naive Hash Count algorithm. The space cost comes from two parts, the stack size and the size of child lists. The stack size is bounded by d, where d is the depth of the XML tree. The size of child lists, $\Sigma_{entry \in stack}|entry.childList|$, is bounded by $|S_1|$. The total complexity is $O(d + |S_1|)$. With index look-ups reduction, the space complexity of the algorithm increases to $O(kd+k|S_1|)$, because, for each $entry.node$ and each node in $entry.childList$, we need to record a $selfCount$ vector of size k.

However, we are able to reduce space complexity to $O(kd)$ without sacrificing efficiency. The idea is, for each node v in the stack, using another k-dimention vector, denoted as $descCount$, to accumulate the keyword occurrences of nodes in $child\_elcaCan(v)$. Let node $u \in child\_elcaCan(v)$, u's $selfCount$ can be added into v's $descCount$ after checking isELCA(u) without storing u into v's child list.

Algorithm 4 gives the isELCA() function in the optimized Hash Count algorithm. It has the same spirit as Algorithm 2, but is more efficient. There are $k-1$ loops except for the least frequent keyword (line 1). In each loop, the number of keyword instances $entry.node$ has covered is looked up from the Hash Count index of w_i, and recorded into $selfCount$ vector in line 3. Line 4 equals to line 8 of Algorithm 2, but here $descCount[]$ is accumulated on the fly. The accumulation may take place at three places in Algorithm 3, line 11, 15 or 20. In any case, it means the $entry.node$ has contained all the keywords, and all the instances covered by $entry.node$ cannot contribute to $entry.node$'s parent. Specifically speaking, (I) line 11 deals with the case when the node is a real ELCA, like node x_1 in Fig. 3; (II) line 15 deals with the case when the node is a potential ELCA, but not a real one. We do not have a corresponding example in Fig. 3. If we remove b_2 from the XML tree, then x_2 is a such node. (III) line 20 deals with the case when the node is not a potential ELCA but still contains all the keywords, i.e. it has a descendant node to be an ELCA. $\{x_3, r\}$ are such examples.

Back to the space cost analysis, in the optimized Hash Count algorithm, each stack entry has four fields, an encapsulated node id, a boolean value representing whether the node is a potential ELCA, a $selfCount$ vector storing the number of keyword instances covered by the node, and a $descCount$ vector recording the number of keyword instances have already been covered. Without storing a child list for a stack entry, the space cost is reduced to $O(kd)$. Here, k is the size of vectors $selfCount$ and $descCount$, the tree depth d bounds the size of the stack.

3.5.4 A running example

We give a set of snapshots for running Algorithm 3 on the XML tree in Fig. 3. At the beginning (step 1), each node is encapsulated into a stack entry, and pushed into the stack, with selfCount vector and descCount vector set to $\vec{0}$ and the $isPotentialELCA$ field of the last pushed stack entry set to $true$. In step 2, when $(3, true, (0,0), (0,0))$ is popped from the stack, its $isPotentialELCA$ field is examined. After node 3 is found not to be a real ELCA, the $isPotentialELCA$ field of the current top stack entry is set to $true$, because keyword instances under node 3 may contribute to node 2 since node 3 is not an ELCA. And then node 4 and 5 are pushed into the stack. In step 3, node 5 is first popped up from the stack and then node 4 is discovered as an ELCA, for the $selfCount$ vector of node 4 is found to be (1,1) larger than its $descCount$ vector (0,0) on every dimension. After node 4 is added into the result set, the $selfCount$ vector of node 4 is accumulated into the current stack top node 2's $descCount$. In a similar manner, when node 2 is popped from the stack, its $selfCount$ vector $(2,2)$ is compared with its $descCount$ vector (1,1). Obviously, node 2 is also an ELCA. Node 2's $selfCount$ is also added into node 1's $descCount$. Then nodes 8, 9, 10 are pushed into the stack in turn. Step 4 operates similarly to step 3. One different point we want to mention is that when node 8 and 1 are popped from the stack. They will not go into isELCA() function to be verified, because after node 9 is found to be an ELCA. The $isPotentialELCA$ fields of node 8 and 1 will not be set to true, and thus node 8 and 1 can be discarded directly.

3.5.5 A potential drawback and the solution

We have shown the advantage of Hash Count algorithm in terms of both time and space complexity. The benefit is brought by introducing the Hash Count indexes. Comparing with the existing algorithms, the potential drawback of Hash Count algorithm is when some Hash Count index cannot fit into memory. In this case, to fetch keyword count information requires disk access to Hash Count index. The performance of Hash Count algorithm will degrade significantly, since disk access is much slower than in-memory operations.

To find a solution to the above problem is not difficult. For a keyword whose hash count index cannot fit into memory, we try to find a witness node for a candidate in the same way as the IS algorithm using the keyword's inverted list. The disk complexity is $O(B)$, where B is the number of disk blocks storing the inverted list. This complexity is much less than $O(d|S_1|)$, the disk complexity if we still use Hash Count index for the stop word.

4. EXPERIMENTS

In this section, we report the performance of Hash Count algorithm and compare it with the state-of-the-art algorithms, DIL algorithm [8] and IS algorithm [25]. The version

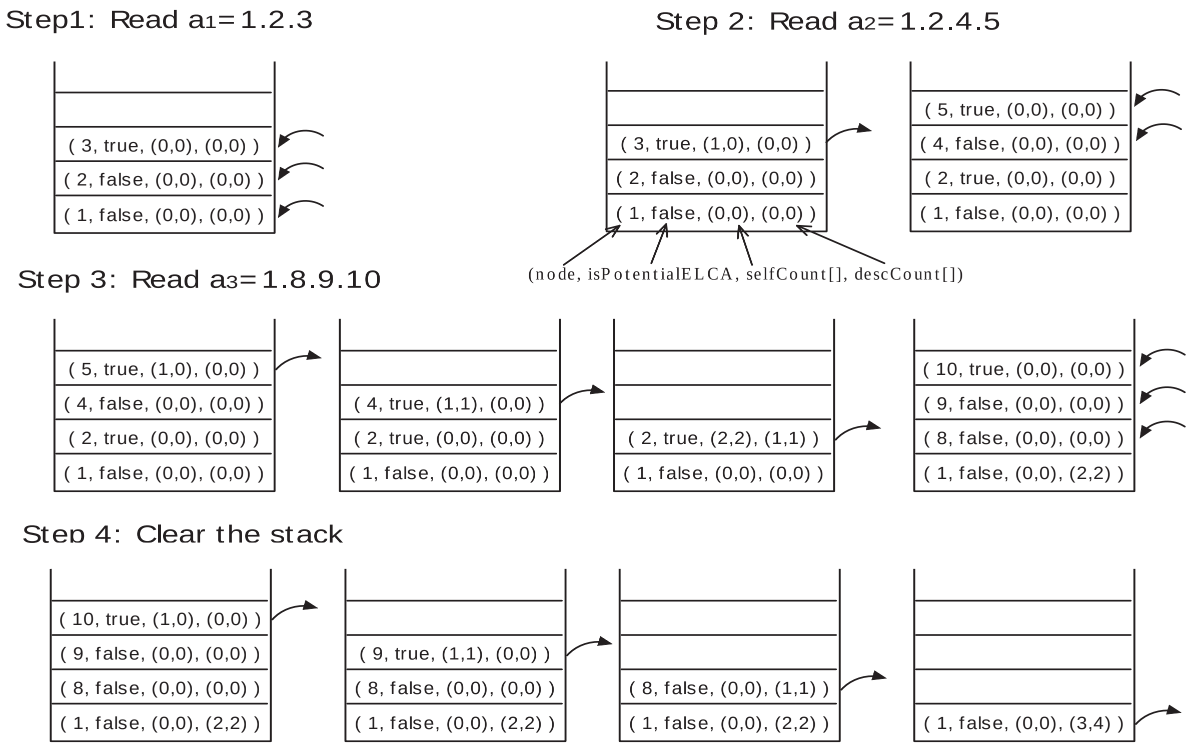

Step1: Read a1=1.2.3
(3, true, (0,0), (0,0))
(2, false, (0,0), (0,0))
(1, false, (0,0), (0,0))
Step 2: Read a2=1.2.4.5
(3, true, (1,0), (0,0))
(2, false, (0,0), (0,0))
(1, false, (0,0), (0,0))
(5, true, (0,0), (0,0))
(4, false, (0,0), (0,0))
(2, true, (0,0), (0,0))
(1, false, (0,0), (0,0))
(node, isPotentialELCA, selfCount[], descCount[])
Step 3: Read a3=1.8.9.10
(5, true, (1,0), (0,0))
(4, false, (0,0), (0,0))
(2, true, (0,0), (0,0))
(1, false, (0,0), (0,0))
(4, true, (1,1), (0,0))
(2, true, (0,0), (0,0))
(1, false, (0,0), (0,0))
(2, true, (2,2), (1,1))
(1, false, (0,0), (0,0))
(10, true, (0,0), (0,0))
(9, false, (0,0), (0,0))
(8, false, (0,0), (0,0))
(1, false, (0,0), (2,2))
Step 4: Clear the stack
(10, true, (1,0), (0,0))
(9, false, (0,0), (0,0))
(8, false, (0,0), (0,0))
(1, false, (0,0), (2,2))
(9, true, (1,1), (0,0))
(8, false, (0,0), (0,0))
(1, false, (0,0), (2,2))
(8, false, (0,0), (1,1))
(1, false, (0,0), (2,2))
(1, false, (0,0), (3,4))

Figure 6: The snapshots of Optimized Hash Count algorithm

of our algorithm we use is the optimized Hash Count algorithm with the modification discussed in Section 3.5.5 (to deal with the case when hash count index is large). All the experiments are done on a laptop with 1.6GHz Turion CPU and 512M memory.

Dataset We test the algorithms on the DBLP dataset [1]. We first parse the data once to build keyword frequency table, keyword inverted lists and hash count indexes. These indexes are stored with Berkeley DB [2], precisely speaking, the inverted list is stored in Berkeley DB's B-tree, and the frequency table and hash count indexes for the keywords are stored in hash tables. The size of the XML document is 180M.

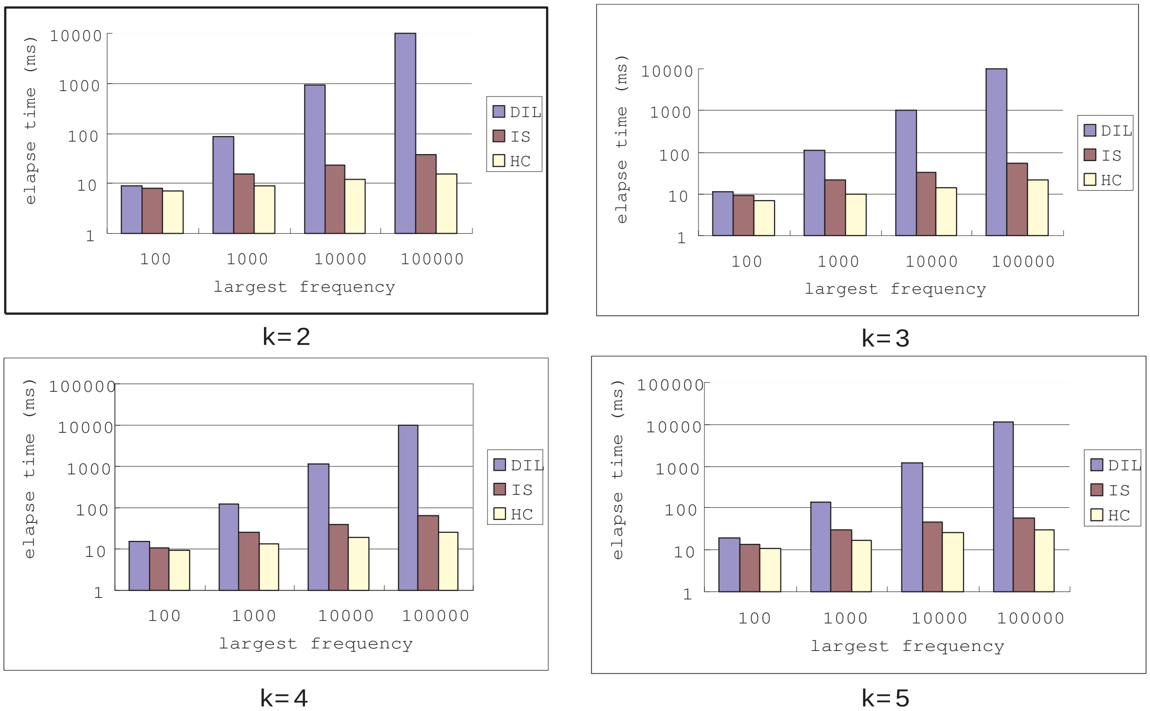
Keyword Query Selection We consider two issues, when we select keyword queries to test the algorithms. (I) Selecting a keyword with a specified frequency (the length of inverted list): we use keyword frequency table to choose keywords having frequencies close to specified values. For instance, we put keywords with frequency close to 100 into a set K_{100}. When we want to choose a keyword with frequency 100, we randomly select a keyword from K_{100}. In our experiments, the most used keyword frequencies are 10, 100, 1,000, 10,000, 100,000. (II) We manually refine the keyword queries. For instance, to form a keyword query, we randomly select two keywords, one from K_{10} and one from K_{1000}. However, it is possible that the combination of these two keywords may not make sense, just because they simply happen to have the specified frequencies. Therefore, for each type of query, we manually choose three reasonable ones from randomly generated ones. The elapse time is the average time of answering three queries. Each query is run five times after memory warm-up.

4.1 Varing the highest frequency

In Fig. 7, we fix the occurrence of the least frequent keyword at 10, and vary the highest frequency from 100 to 100,000 at 10 times each step. We use L/H to denote the lowest/highest frequency. When L=10, H=100, three algorithms have almost the same performance, and the advantage of Hash Count algorithm is not obvious, because if H is not very large, DIL algorithm is acceptable, for $|S|$ is not very large. As to the IS algorithm, the number of logarithmic look-ups is also small. As H increases, DIL algorithm degrades dramatically, and IS algorithm degrades mildly. The advantage of Hash Count algorithm become more and more obvious. Hash Count is better that DIL in orders of magnitude, and better than IS by up to 30%. We also vary the number of keywords from 2 to 5. As the number of keywords goes up, the elapse time of all three algorithms goes up as well, but the impact of the number of keywords is not as significant as the length of the longest keyword inverted list. Hash Count algorithm is always the best choice among the three.

4.2 Varing the lowest frequency

In Fig. 8, we fix the occurrence of the most frequent keyword at 100,000, and vary the smallest keyword frequency from 10 to 10,000. As L increases from 10 to 10,000, the elapse time of DIL algorithm stays almost the same with a mild increase. The reason is that $|S|$ is almost around 100,000, although has a mild increase. The gap between the DIL algorithm with the other two algorithms become closer and closer, for the working list of IS and Hash Count algorithms becomes larger and larger, but DIL is still the last choice. Hash Count algorithm is better than IS algo-



k=2

k=3

k=4

k=5

Figure 7: Varing the largest frequency

rithm by an obvious margin, because $log|S_{max}|$ now is not a neglectable factor.

4.3 The same frequencies

In Fig. 9, we test the cases for $L = H$ from 10 to 10,000. We did not choose queries with $L = H = 100,000$, because such queries are too general and do not make sense. When $L = H = 10$, three algorithms are almost the same, DIL wins slightly owing to its not sophisticated implementation. With the length of inverted lists goes up, the advantage of Hash Count becomes obvious. DIL again suffers from the total length of all the input lists, and IS has an increasing $log|S_{max}|$ look-ups, and is even worse than DIL. As the number of keywords increase, the benefit the Hash Count algorithm has brought becomes strengthened.

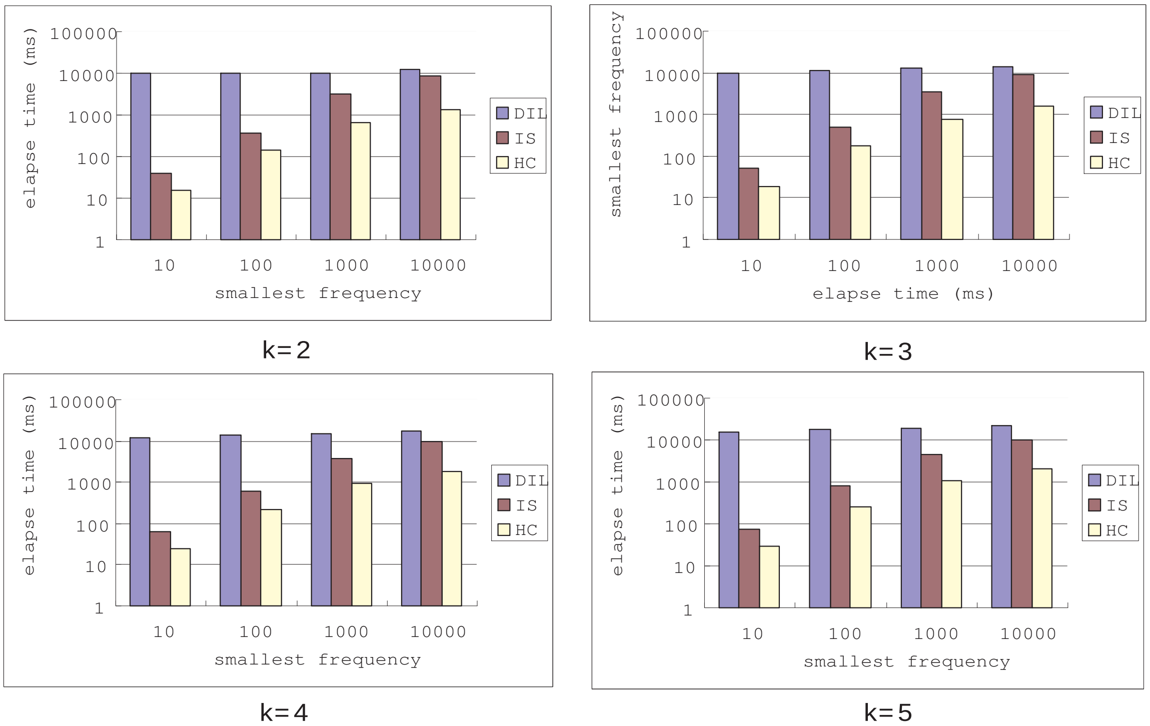
4.4 Discussion

In the above experiments, we have tested several keyword sets. An interesting observation is that although DIL algorithm and IS algorithm may outperform one another in different situations, Hash count algorithm is often the best in almost every case. We analyze the reason as follows: When there exists a rare keyword, Hash Count can take advantage of this information by using the shortest inverted list as the input. It manages to share the same merit as IS algorithm to achieve very good performance; when keyword frequencies are not biased, it also has satisfactory performance because it does not need to read all the inverted lists, consequently, the more the number of keywords is, the faster Hash Count is, comparing to the DIL algorithm. This conclusion only holds when the hash indexes can fit into memory. If all other inverted lists except for the shortest one are not able to fit into memory, Hash Count algorithm will perform the same as the IS algorithm.

5. RELATED WORKS

Keyword search on XML data has drawn the attention of database community recently. Unlike traditional keyword search to return a web page, the result of keyword search on an XML document is modeled as an XML fragment. Two issues to answer keyword queries have been intensively investigated: effectiveness and efficiency.

As to the first category, effectiveness, many notions have been proposed to model keyword query semantics. For pure keyword query, ELCA proposed in [8] is the first semantic approach to model keyword query result, and [8] also proposed a ranking strategy to rank the resulting ELCA nodes. Another widely accepted notion is SLCA, first proposed in [24]. In fact, the importance of keyword queries on XML data was realized several years ago, when some keyword-like languages were addressed and studied. In [21], *meet* operator was used for querying an XML document by finding LCA nodes of keywords given in the query. Schema-free XQuery [15] adopted the concept of MLCA to incorporate keyword search into XQuery. MLCA is similar to SLCA, but with more imposed conditions. These LCA-based notions have the following relationship: $mlca(S_1, \ldots, S_k) \subseteq slca(S_1, \ldots, S_k) \subseteq elca(S_1, \ldots, S_k) \subseteq lca(S_1, \ldots, S_k)$. Some other works utilized semantics to refine the produced LCA nodes, eg, XSEarch [5] and CVLCA [14] have improved the quality of LCA and ELCA respectively. Another stream of works (also focusing on the effectiveness issue) aim to identify what information should be returned to users. The observation is that an LCA node may not be a meaningful entity [17], or the query result should follow some universally accepted properties [18]. The works [17, 18] use subtrees



k=2

k=3

k=4

k=5

Figure 8: Varing the lowest frequency

rooted at SLCA nodes as base to infer meaningful returned fragments. Kong et al. [13] developed the idea in [18] using subtrees rooted at ELCA nodes.

On the other hand, the efficiency of LCA evaluation have been studied mainly on SLCA [24, 22, 23] and ELCA [8, 25]. The performance matters, because computing LCAs is an important step of many XML keyword search systems, such as [17, 18, 13]. Our work falls in this category.

Keyword query are also studied on graphs and in relational databases, such as [6, 3, 4, 10, 11, 12, 9, 19, 16, 20, 7]. (Just list a number of them.) The semantics of keyword query result is no longer LCA-based entities, since the data is not of a tree structure. The techniques of these works can be definitely borrowed to process XML data, but it is obvious that these techniques are not well-fitted, since they are not specially designed for XML.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed an efficient algorithm named Hash Count to find ELCAs for keyword queries on XML data. Our idea is to use hash index to find ELCA candidates and use keyword occurrences under a node to facilitate ELCA verification. We have shown that Hash Count achieves time complexity $O(kd|S_1|)$, where k is the number of keywords, d is the depth of the XML document, $|S_1|$ is the occurrence of the least frequent keyword. Comparing to DIL in [8] of complexity $O(kd|S|)$ and Indexed Stack algorithm in [25] of $O(kd|S_1|log|S|)$, where $|S|$ is the occurrence of all keyword instances, Hash Count has given out the best complexity result so far. Furthermore, we have shown that Hash Count indeed performs better than the state-of-the-art works on real datasets.

One interesting direction for future work is to compress the hash indexes without sacrificing the algorithm performance. Currently, hash index records nodes both directly and indirectly containing keywords, leading to a potentially large number of nodes to store. A desirable improvement is to reduce the number of nodes stored while keeping index look-ups still in constant time.

An unavoidable drawback of Hash Count algorithm is that it cannot determine "witness" nodes. Though, for the notion of ELCA, it does not matter which nodes are the contributing nodes to an answer, the information about witness nodes may be helpful to the users.

Acknowledgment

We would like to thank Baichen Chen and anonymous reviewers for their helpful comments both on the current version of this paper and future directions they pointed out. We also appreciate the help from Chong Sun for his shared code. This work was supported by the Australian Research Council Discovery Project under the grant number DP0878405.

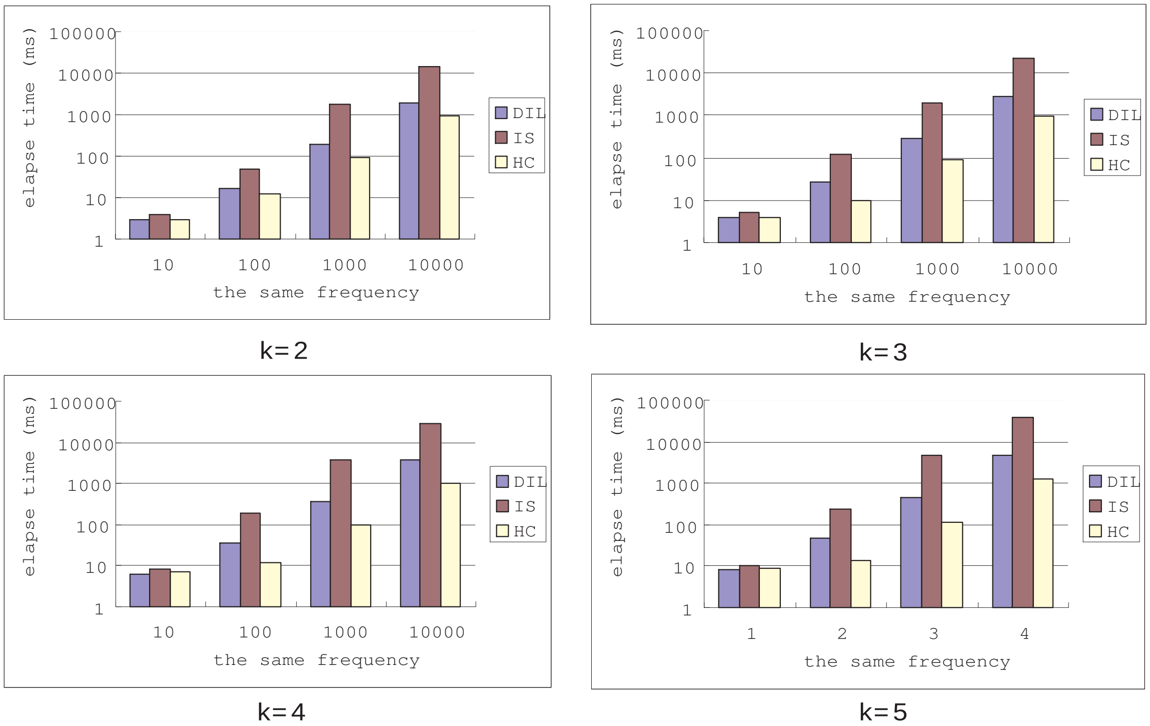
7. REFERENCES

[1] DBLP XML Records. In *http://dblp.uni-trier.de/xml/*.

[2] Oracle Berkeley DB. In *http://www.oracle.com/technology/products/berkeley-db/index.html*.

[3] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.



k=2

k=3

k=4

k=5

Figure 9: Keyword frequencies are the same

[5] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. Xsearch: A semantic search engine for xml. In *VLDB*, pages 45–56, 2003.

[6] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *VLDB*, pages 26–37. Morgan Kaufmann, 1998.

[7] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD Conference*, pages 927–940, 2008.

[8] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *SIGMOD Conference*, pages 16–27, 2003.

[9] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD Conference*, pages 305–316, 2007.

[10] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[11] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on xml graphs. In *ICDE*, pages 367–378, 2003.

[12] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[13] L. Kong, R. Gilleron, and A. Lemay. Retrieving meaningful relaxed tightest fragments for xml keyword search. In *EDBT*, pages 815–826, 2009.

[14] G. Li, J. Feng, J. Wang, and L. Zhou. Effective keyword search for valuable lcas over xml documents. In *CIKM*, pages 31–40, 2007.

[15] Y. Li, C. Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, pages 72–83, 2004.

[16] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD Conference*, pages 563–574, 2006.

[17] Z. Liu and Y. Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD Conference*, pages 329–340, 2007.

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

[19] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD Conference*, pages 115–126, 2007.

[20] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: the power of rdbms. In *SIGMOD Conference*, pages 681–694, 2009.

[21] A. Schmidt, M. L. Kersten, and M. Windhouwer. Querying xml documents made easy: Nearest concept queries. In *ICDE*, pages 321–329, 2001.

[22] C. Sun, C. Y. Chan, and A. K. Goenka. Multiway slca-based keyword search in xml data. In *WWW*, pages 1043–1052, 2007.

[23] W. Wang, X. Wang, and A. Zhou. Hash-search: An efficient slca-based keyword search algorithm on xml documents. In *DASFAA*, pages 496–510, 2009.

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

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