Efficient Keyword Search on Large Tree Structured Datasets

Aggeliki Dimitriou
School of Electrical and Computer Engineering
National Technical University of Athens, Greece
angela@dblab.ntua.gr

Dimitri Theodoratos
Department of Computer Science
New Jersey Institute of Technology, USA
dth@cs.njit.edu

ABSTRACT
Keyword search is the most popular paradigm for querying XML data on the web. In this context, three challenging problems are (a) to avoid missing useful results in the answer set, (b) to rank the results with respect to some relevance criterion and (c) to design algorithms that can efficiently compute the results on large datasets.

In this paper, we present a novel multi-stack based algorithm that returns as an answer to a keyword query all the results ranked on their size. Our algorithm exploits a lattice of stacks each corresponding to a partition of the keyword set of the query. This feature empowers a linear time performance on the size of the input data for a given number of query keywords. As a result, our algorithm can run efficiently on large input data for several keywords. We also present a variation of our algorithm which accounts for infrequent keywords in the query and show that it can significantly improve the execution time. An extensive experimental evaluation of our approach confirms the theoretical analysis, and shows that it scales smoothly when the size of the input data and the number of input keywords increases.

Categories and Subject Descriptors
H.3.3 [Information Search and Retrieval]: Search process; H.3.4 [Systems and Software]: Performance evaluation (efficiency and effectiveness)

General Terms
Algorithms, Design, Performance

Keywords
keyword search, XML, tree-structured data, LCA, search algorithm, ranking

1. INTRODUCTION
In the era of social media and the web the amount of data that is published and exchanged between various data sources grows fast. In this context, tree-structured data in XML or JSON [7] have become the predominant formats for exporting and exchanging data. The reason for this success is twofold: (a) the user does not have to master a complex structured query language like XQuery in order to retrieve data, and (b) keyword queries can be issued against data sources without full or even partial knowledge of the structure of the source. They can even be issued against data sources with evolving schemas or against multiple data sources with different structures as is usually needed on the web.

The price to pay for this flexibility of keyword search in query formulation is imprecision in the query answers: usually a plethora of candidate solutions are identified but only few of them are useful to the user. The aim of the several approaches assigning semantics to keyword queries is to identify and return to the user all of the candidate solutions that are relevant (perfect recall) and no irrelevant ones (perfect precision).

In contrast to the Information Retrieval (IR) perspective to keyword search where the candidate solutions are whole documents containing instances of all the keywords, in the context of XML, the documents have a tree structure. The candidate solutions of a query are defined usually as the most specific subtrees (minimum connected trees) of the input XML tree document that contain an instance of all the keywords. These subtrees are appropriate as candidate solutions since they relate the corresponding keyword instances as closely as possible. The root of these subtrees is the LCA of the subtree keyword instances and is often used to identify the candidate solutions [18].

Different approaches that assign semantics to keyword queries on XML data define a subset of the LCAs as an answer to the query by exploiting the structural properties of the XML trees [8, 26, 17, 27]. We refer to these approaches as filtering approaches as they filter out part of the LCAs that they consider irrelevant. Some filtering approaches, take also into account semantic information (node labels and label paths in the XML tree) in order to filter out irrelevant LCAs [14, 6, 12, 10, 15, 24]. Although, filtering approaches are intuitively reasonable, they are sufficiently ad-hoc and they are frequently violated in practice resulting in low precision and/or recall [23].

Ranking the results instead of filtering improves the sys-
tem’s usability. Several semantics assigning approaches suggest ranking techniques for keyword queries on XML data that place on top those LCAs that are believed to be more relevant to the user’s intent [8, 6, 2, 13, 5, 23]. We refer to these approaches as ranking approaches. In order to rank the results they often exploit also value-based statistics techniques from IR (e.g., PageRank [3] or $tf \times idf$) adapted to the tree structure nature of XML. Ranking approaches, when combined with filtering approaches (that is, when they rank filtered LCAs) inherit the low recall of the filtering approaches.

Several algorithms have been proposed for computing keyword query answers according to filtering and ranking semantics [8, 9, 26, 27, 21, 16, 5]. A major challenge for all these algorithms is to achieve efficiency when the number of keywords and the number of instances in the XML dataset grows.

**Contribution.** In this paper we present an efficient algorithm for computing keyword queries with many keywords on large, deep and complex tree-structured datasets. Our algorithm does not exclude any LCA but returns all of them ranked on their size. The main contributions of our paper are the following:

- We introduce the notion of LCA size and we use it as a relevance criterion for the results of a keyword query
- We design an efficient multi-stack based algorithm which computes all LCAs of a keyword query on a data tree and ranks them based on their size. Our algorithm exploits the different partitions of the query keywords that together are organized into a lattice of multicolumn stacks.
- We analyze our algorithm and show that, for a given number of query keywords its performance is linear on the size of the input keyword inverted lists. This behavior contrasts with that of previous algorithms that compute all LCAs, whose complexity depends on the product of the size of the input inverted lists.
- We consider the case where some keywords have a small number of instances in the dataset (rare keywords) and we developed a variation of our algorithm which runs much faster than the basic algorithm (in some cases by orders of magnitude).
- We experimentally evaluate our algorithm on large real and benchmark datasets. Our experiments confirm the theoretical analysis of the algorithm and show that it outperforms previous algorithms and scales smoothly when the size of the input inverted lists and the number of keywords increases.

Since our algorithm does not restrict the number of LCAs it returns, it shows perfect recall. However, our intension is not to introduce new keyword query semantics: our algorithm can be the basis for implementing other approaches that combine LCA sizes with other metrics including ranking schemes that exploit value-based statistics.

**Outline.** In the next section we present related contributions. Section 3 introduces the theoretical framework of our work. In Section 4, we describe our main algorithm, LCAs$, and provide its complexity analysis. Section 5 elaborates on a variation of our algorithm for answering queries with rare keywords. In Section 6 we experimentally analyze our algorithm, and we conclude in Section 7 suggesting also future research directions.

## 2. RELATED WORK

A number of approaches for assigning semantics to keyword queries on XML trees identify candidate keyword query answers among lowest common ancestors (LCAs) of nodes containing query keywords [20]. Some of them rely purely on hierarchical criteria, disregarding semantic information in the XML tree (node labels or label paths). According to the smallest LCA semantics (SLCAs) [26, 17, 17] the valid LCAs do not contain other descendant LCAs of the same keyword set. Relaxing this semantics, exclusive LCAs (ELCAs) were introduced in [8] and later formally in [27]. ELCAs qualify as relevant LCAs also those that are ancestors of other LCAs, as long as they refer to a different set of keyword instances. The semantic approaches valuable LCAs (VLCS) [6, 12] and meaningful LCAs (MILCs) [14] attempt to capture the user intent by exploiting the labels that appear in the paths of the subtree rooted at an LCA. However, all these semantics are restrictive and demonstrate low recall rates as shown in [23].

![Figure 1: Example data tree](image-url)
query answer depend on the LCA semantics and ranking method used. They are designed to take advantage of the adopted LCA semantics by pruning irrelevant LCAs early on in the computation. In [8] a stack based algorithm that processes inverted lists of query keywords and returns ranked ELCAs is presented. The ranking is performed based on precomputed tree node scores according to an adaptation of PageRank [3] and of textual keyword proximity in the subtree of the ranked ELCA. In [26], two efficient algorithms for computing SLCAs exploiting special structural properties of SLCAs are introduced. In addition, the authors suggest an extension of their basic algorithm, so that it returns all LCAs by augmenting the set of already computed SLCAs. However, this approach does not compute LCA subtree sizes as we do in this paper. They experimentally study their algorithms for SLCAs on 2-5 keyword queries. In [21] another algorithm for efficiently computing SLCAs for both AND and OR keyword query semantics is developed. ELCA computation which is more efficient than that in [8] is achieved by the indexed stack algorithm designed in [27]. Finally, [1, 2, 23] elaborate on sophisticated ranking of candidate LCAs aiming primarily on effective keyword query answering.

The approach in [9] is the most relevant to our work. This approach returns all minimum connected trees (MCTs) of a keyword query whose size is smaller than a given threshold. The main algorithm of this work, SA, aims at grouping together isomorphic MCTs. We experimentally compare our algorithm with algorithm SAOne which is a variation of SA also introduced in [9] and returns all LCAs.

3. FRAMEWORK AND DEFINITIONS

We model XML data, as usual, as ordered labeled trees. Tree nodes represent XML elements or attributes. Every node has an id, a label (corresponding to an element tag or attribute name) and possibly a value (corresponding to the text content of an element or to an attribute’s value). For identifying tree nodes we adopt the Dewey encoding scheme [4], which encodes tree nodes according to a pre-order traversal of the XML tree. The Dewey encoding expresses naturally ancestor-descendant relationships among tree nodes and conveniently supports the processing of nodes in stacks [8].

A keyword query \( Q \) is a set of keywords: \( Q = \{k_1, ..., k_n\} \). A keyword \( k \) may appear in the label or in the value of a node \( n \) in the XML tree, in which case we say that node \( n \) constitutes a keyword instance of \( k \), or more simply an instance of \( k \). Since a node may contain multiple distinct keywords in its value and label, it may be an instance of multiple keywords.

The minimum connected tree, \( T_S \), of a set \( S \) of nodes in an XML tree \( T \) is the minimum subtree \( T_S \) of \( T \) that contains all nodes in \( S \). The root of \( T_S \) is the lowest common ancestor (LCA) of the nodes in \( S \), denoted \( lca(S) \). The size of \( T_S \) is the number of its edges. Let \( I \) be a set of instances of keywords in \( Q \). If \( I \) contains one instance for every keyword in \( Q \), we call \( I \) a query instance for \( Q \). The LCA of an instance of \( Q \) is also called LCA of \( Q \). An LCA of a proper subset of \( Q \) is a partial LCA of \( Q \).

We rank LCAs based on their size which is defined next. Let \( I \) and \( I' \) be two different but not necessarily disjoint instances of \( Q \) in an XML tree \( T \). Clearly, their minimum connected trees \( T_I \) and \( T_{I'} \) may be rooted at the same LCA \( l \).

DEFINITION 1. Given an XML tree, the size of an LCA \( l \) of a query \( Q \) is the size of the smallest minimum connected tree of the instances of \( Q \) rooted at \( l \).

For example, in the XML tree of Figure 1, the size of LCA 1.1.1.3 of query \( Q = \{\text{XML, John, Smith}\} \) is 4 since there are exactly two minimum connected trees of instances of \( Q \) rooted at 1.1.1.3 (one containing the instance 1.1.1.3.1.2 of John and the other containing the instance 1.1.1.3.2.2 of John) and their sizes are 5 and 4.

A solution of a query \( Q \) over an XML tree \( T \) is a pair \((l, s)\) of an LCA \( l \) of \( Q \) and its size \( s \). For example, the pair (1.1.1.3, 4) is a solution of \( Q = \{\text{XML, John, Smith}\} \) over the XML tree of Figure 1. The answer \( A \) of \( Q \) over \( T \) is the list of all the solutions of \( Q \) over \( T \) ranked on their size: \( A = \{(l_1, s_1), (l_2, s_2), \ldots\} \), \( s_1 \leq s_2 \), \( i < j \). If two solutions have the same size, their relative order in the answer is indifferent. The ranking of the LCAs on their sizes is based on the intuition that the quality of an LCA depends on how close it brings a set of keyword instances.

Next, we make some remarks on LCAs and minimum connected trees that are used in our algorithms.

![Figure 2: Partial LCA subtrees under LCA 1.1.1.3 of query \{XML, Brown, RDF, Smith\}]
4. COMPUTING LCAS AND THEIR SIZES

In this section, we present and analyze algorithm LCAsz, which computes for a given keyword query the LCAs and their sizes in an XML tree.

4.1 Algorithm LCAsz

LCAsz is a stack based algorithm that returns all LCAs in size order. The input of the algorithm is the set of the inverted lists of all keywords of an XML tree T and a keyword query Q. The output is the answer $A = [(l_1, s_1), (l_2, s_2), \ldots]$ of Q on T.

Based on a straightforward interpretation of Definition 1, the calculation of the size of an LCA implies the computation of all minimum connected trees rooted at this LCA along with their sizes. Algorithm LCAsz, however, avoids exhaustively examining all minimum connected trees of each LCA. In order to do so, it combines keyword instances into partial and full LCAs, which are progressively compared resulting into one winning partial or full LCA with minimum size for every node and every keyword subset of the keyword query. LCAsz performs this operation bottom-up combining step by step partial LCAs of instances of a subset S of the query keywords, located lower in the XML tree, into LCAs of instances of a superset of S located higher in the tree. Partial LCAs are propagated upwards to their ancestors. For simplicity, in the rest of the paper, the size of a partial LCA $l$ at a node n under consideration refers to the size of the partial LCA subtree of $l$ rooted at n. In this sense, during the propagation, the size of a partial LCA is incremented to reflect the number of upward steps from the relevant partial LCA. At every node in this path, the partial LCA size is compared against the size of any comparable partial LCAs (i.e., LCAs that refer to the same set of keywords) and only the minimum size for every partial LCA is recorded. This way, only one size is kept for every collection of comparable partial LCAs computed that far in the subtree of this node. For example, in Figure 1, there are 6 different partial LCA subtrees rooted at node 1.1.1 for the keyword set \{John, Smith\} having 5 different sizes. Only the minimum size 1 corresponding to partial LCA 1.1.1.2 wins.

\[ \text{LCAsz is a stack based algorithm that returns all LCAs in size order.} \]

\[ \text{The input of the algorithm is the set of inverted lists of all keywords of an XML tree T and a keyword query Q.} \]

\[ \text{The output is the answer } A = [(l_1, s_1), (l_2, s_2), \ldots] \text{ of Q on T.} \]

\[ \text{Based on a straightforward interpretation of Definition 1, the calculation of the size of an LCA implies the computation of all minimum connected trees rooted at this LCA along with their sizes.} \]

\[ \text{Algorithm LCAsz, however, avoids exhaustively examining all minimum connected trees of each LCA. In order to do so, it combines keyword instances into partial and full LCAs, which are progressively compared resulting into one winning partial or full LCA with minimum size for every node and every keyword subset of the keyword query.} \]

\[ \text{LCAsz performs this operation bottom-up combining step by step partial LCAs of instances of a subset S of the query keywords, located lower in the XML tree, into LCAs of instances of a superset of S located higher in the tree.} \]

\[ \text{Partial LCAs are propagated upwards to their ancestors. For simplicity, in the rest of the paper, the size of a partial LCA } l \text{ at a node } n \text{ under consideration refers to the size of the partial LCA subtree of } l \text{ rooted at } n. \]

\[ \text{In this sense, during the propagation, the size of a partial LCA is incremented to reflect the number of upward steps from the relevant partial LCA.} \]

\[ \text{At every node in this path, the partial LCA size is compared against the size of any comparable partial LCAs (i.e., LCAs that refer to the same set of keywords) and only the minimum size for every partial LCA is recorded.} \]

\[ \text{This way, only one size is kept for every collection of comparable partial LCAs computed that far in the subtree of this node. For example, in Figure 1, there are 6 different partial LCA subtrees rooted at node 1.1.1 for the keyword set \{John, Smith\} having 5 different sizes. Only the minimum size 1 corresponding to partial LCA 1.1.1.2 wins.} \]
Algorithm 1: LCAs

1. \textbf{LCAs}($k_1, \ldots, k_n$: keyword query, \textit{invL}: inverted lists)

2. \texttt{kwSubsets} = \{\{k_1\}, \{k_2\}, \ldots, \{k_n\}\}

3. \textbf{while} currentNode = getNextNodeFromInvertedLists() \textbf{do}

4. \hspace{1em} coarsenessLevel = 1, size=0, provenance=\emptyset

5. \hspace{1em} pLCA = newPartialLCA(currentNode.ID, currentNode.kwSubset, size, provenance)

6. \hspace{1em} addPartialLCA(1, pLCA)

7. \hspace{1em} \textbf{while} partialLCALists contains partialLCAs for coarsenessLevel \textbf{do}

8. \hspace{2em} \textbf{if} number of stacks \textless max number of stacks at this coarsenessLevel \textbf{then}

9. \hspace{3em} break

10. \hspace{2em} \textbf{while} partialLCA = partialLCALists(coarsenessLevel).next() \textbf{do}

11. \hspace{3em} \textbf{for} every stack of coarsenessLevel containing partialLCA.kwSubset \textbf{do}

12. \hspace{4em} push(stack, partialLCA.ID, partialLCA.kwSubset, partialLCA.size)

13. \hspace{3em} \textbf{if} coarsenessLevel \textless n \textbf{then}

14. \hspace{4em} addPartialLCA(coarsenessLevel+1, partialLCA)

15. \hspace{3em} coarsenessLevel++

16. \hspace{1em} emptyStacks()

17. \hspace{1em} addPartialLCA(coarsenessLevel, partialLCA)

18. \hspace{1em} \textbf{if} (partialLCA.ID, partialLCA.kwSubset) \textbf{not in} partialLCALists(coarsenessLevel) \textbf{then}

19. \hspace{2em} insert partialLCA into partialLCALists(cL)

20. \hspace{1em} else \textbf{if} current (partialLCA.ID, partialLCA.kwSubset) entry size \textless partialLCA.size \textbf{then}

21. \hspace{2em} replace with partialLCA

22. \hspace{1em} push(stack, nodeID, kwSubset, size)

23. \hspace{2em} \textbf{while} stack.dewey \textless nodeID \textbf{do}

24. \hspace{3em} pop(stack)

25. \hspace{2em} \textbf{while} stack.dewey \neq nodeID \textbf{do}

26. \hspace{3em} addEmptyRow(stack) /* updating stack.dewey until it is equal to nodeID */

27. \hspace{3em} replaceSizeIfSmallerWith(stack.topRow, kwSubsetColumn, size)

28. \hspace{2em} emptyStacks()

29. \hspace{2em} foreach coarsenessLevel \textbf{do}

30. \hspace{3em} \textbf{if} partialLCALists(coarsenessLevel) is not empty \textbf{then}

31. \hspace{4em} \textbf{while} partialLCA = partialLCALists(coarsenessLevel).next() \textbf{do}

32. \hspace{5em} \textbf{for} every stack of coarsenessLevel containing partialLCA.kwSubset \textbf{do}

33. \hspace{6em} push(stack, partialLCA.ID, partialLCA.kwSubset, partialLCA.size)

34. \hspace{4em} \textbf{foreach} stack of coarsenessLevel \textbf{do}

35. \hspace{5em} repeat

36. \hspace{6em} pop(stack)

37. \hspace{5em} until top entry contains only propagated or empty elements and the other entries are empty;

and is propagated upwards to the ancestors of 1.1.1.

In this framework, the problem of finding all LCAs with their sizes translates into examining all possible arrangements of keywords under a candidate LCA and discarding among them those with larger LCA sizes. The goal is to avoid a partial LCA size computation as low in the XML tree as possible based on the sizes of its descendant partial LCAs. This way the number of minimum connected trees of keyword instances, that need to be computed, decreases.

A lattice of keyword partitions. The possible arrangements of a keyword set reflects the partitions of the set. We can define a refinement relation \(\preceq\) on the partitions of a set: \(P_1 \preceq P_2\) iff for every set \(s_1 \in P_1\) there is a set \(s_2 \in P_2\) such that \(s_1 \subseteq s_2\). If \(P_1 \preceq P_2\), \(P_1\) is said to be finer than \(P_2\) and \(P_2\) coarser than \(P_1\). It is well known that this relation is a partial order and the set of partitions equipped with this partial order forms a lattice. Figure 3 shows the Hasse diagram of the lattice of the keyword set \{XML, Author, John, Smith\}. At every coarseness level all partitions have the same number of elements. Every partition in one level can be produced by taking the union of two elements of a partition of the previous level. A partition in one level may be produced by different partitions of the previous level. In our context, a partition represents a full LCA and its elements correspond to partial LCAs. Algorithm LCAs maintains a separate stack for each partition. The unique stack at coarseness level 1, is the initial stack that contains singletons for all keywords. The last coarseness level always contains one stack, too. This final stack holds full LCAs, i.e., query solutions. Every path from the
Procedure pop

1 pop(stack)
2 cols = stack.columns
3 /* number of kwSubsets in the partition of the stack */
4 popped = stack.pop()
5 if cols == 1 then
6   addResult(stack.dewey, popped[0].size)
7 /* Produce new LCAs from two partial LCAs */
8 if cols > 1 then
9   for i=0 to cols do
10      for j=i to cols do
11         if popped[i] and popped[j] contain sizes AND popped[i].provenance \ popped[j].provenance = {} then
12            newKwSubset = popped[i].kwSubset \ popped[j].kwSubset
13            newSize = popped[i].size + popped[j].size
14            pLCA = newPartialLCA(stack.dewey, newKwSubset, newSize, newProvenance)
15            if cardinalityOf(newKwSubset) == stack.coarsenessLevel+1 then
16               addPartialLCA(stack.coarsenessLevel+1, pLCA)
17 /* Update ancestor (i.e., new top entry) with sizes from popped entry */
18   if stack is not empty and cols > 1 then
19      for i=0 to cols do
20         if popped[i].size + 1 < stack.topRow[i].size then
21            stack.topRow[i].size = popped[i].size + 1
22            stack.topRow[i].provenance = {lastStep(stack.dewey)}
23   removeLastDeweyStep(stack.dewey)

initial to the final stack suggests a unique way of combining partial LCAs to produce full LCAs.

Stack structure. The key data structures used by LCAs are the stacks of the keyword partitions. A stack stores information about the nodes in the path defined by a Dewey code. Each stack entry corresponds to a Dewey step and the node identified by the Dewey code subsequence to this step. E.g., if the top entry of a stack refers to node 1.2.3.4.5, it contains information about nodes in the path from the root to the node 1.2.3.4.5 and the third stack entry from the bottom refers to node 1.2.3. Pushing and popping actions on the stack add and remove steps from the end of the Dewey code. All the entries of the stack are arrays indexed by the keyword subsets of the partition represented by the stack. The array element at the column named by the keyword subset S may contain the size s of a partial LCA l of S at the node corresponding to the array entry. If a size s is present, the element contains also one or more provenance numbers (namely Dewey steps) identifying one or more of the outgoing edges of the entry node. These edges indicate which one of the partial LCA subtrees of S rooted at l contributes s.

Algorithm description. The algorithm, outlined in Listing 1, implements the ideas described above. The main block of LCAs is a loop (lines 3-15) that iterates over the keyword inverted lists, processing each time the first (in document order) keyword instance from the nodes in the lists. The processing consists of pushing this instance to all the stacks that have a column named by the set containing only the corresponding keyword. This processing applies not only to single keyword instances but also to partial LCAs produced by the stacks. In this sense, single keyword instances are treated uniformly as partial LCAs of a single keyword with size 0 (line 5).

Observing the lattice illustrated in Figure 3, one may notice that most partitions have more than one incoming edge. These edges entail multiple updates of a single stack for the same node. However, LCAs avoid this redundancy by using lists of partial LCAs, one for each coarseness level. Partial LCAs are added to these lists before being propagated to the next level (line 6 of algorithm LCAs and line 16 of procedure pop). This usage of partial LCA lists filters partial LCAs and synchronizes their propagation. As a consequence, stack pushes are minimized. When a partial LCA appears at a certain coarseness level, it is added to the list of the subsequent level (lines 17-21). If a list already contains a comparable LCA with the same node id, only the smallest between the two is kept in the list (lines 20-21). For instance, in the example tree of Figure 1, there are multiple partial LCA subtrees rooted at 1.1.1 for the keyword set {XML, John, Smith}. However, in a partial LCA list only the partial LCA entry with id 1.1.1.1 and size 2 appears for {XML, John, Smith}. This size corresponds to the minimum connected tree of the instances 1.1.1.1 for XML and 1.1.1.2 for John and Smith.

All partial LCAs are pulled sequentially from partial LCA lists (line 7). They are pushed into stacks only at the coarseness level of the list, which contain a column named by the
partial LCA’s keyword subset (lines 11-12). A partial LCA is not processed before all the stacks at this level have been created (lines 8-9). This way, LCAsz guarantees that no partial LCA will fail to be pushed into a stack just because this one has not yet been created. When all keyword instances are processed, all stacks are emptied in coarseness level order (line 16): first, for every coarseness level, the remaining partial LCAs in the corresponding partial LCA list are processed (lines 29-33), and then, procedure pop is called on all the entries of the stacks of that level (lines 34-37).

Pushing a partial LCA into a stack (lines 22-27) can be done only if this partial LCA is a child or self of the stack’s top node. To this end, entries from the stack are popped until an ancestor of the partial LCA is reached. Figure 4 illustrates a sequence of pop and push actions on the stack of coarseness level 1 for query \{XML, John, Smith\} when instance 1.1.2.1 for XML is processed. Under certain conditions, this popping may produce new partial LCAs. This issue is discussed in the next paragraph. After popping, LCAsz prepares the stack for the new node, by adding empty entries, until the stack top entry corresponds to the new partial LCA (lines 25-26). These are the entries at 1.1.2 and 1.1.2.1 in the last stack state of Figure 4. Finally, LCAsz replaces the size in the top stack entry element corresponding to the keyword subset of the partial LCA with the new one (line 27). The replacement is performed only if no size for this keyword subset is recorded or the new size is smaller than the recorded one. Size replacement allows LCAsz to process without extra popping and pushing instances of different keywords at the same node (e.g., John and Smith at 1.1.3.1.2.2 of the first stack state of Figure 4), as well as keywords at internal nodes of the data tree.

Procedure pop (invoked at line 24 of LCAsz) is the most important of the algorithm, as it is the one that forms partial LCAs and returns solutions (full LCAs). The first step of this procedure (line 3) actually pops the stack’s top entry, leaving the stack with the parent of the previous top entry. If the stack contains only one column, it is the (final) stack of the last coarseness level that produces solutions (lines 25-26). These are the entries at 1.1.2 and 1.1.2.1 in the last stack state of Figure 4. Finally, LCAsz replaces the size in the top stack entry element corresponding to the keyword subset of the partial LCA with the new one (line 27). The replacement is performed only if no size for this keyword subset is recorded or the new size is smaller than the recorded one. Size replacement allows LCAsz to process without extra popping and pushing instances of different keywords at the same node (e.g., John and Smith at 1.1.3.1.2.2 of the first stack state of Figure 4), as well as keywords at internal nodes of the data tree.

For all pairs of keyword subsets in the popped entry, which have a partial LCA size and do not share any provenance Dewey steps (line 9), the algorithm creates a new partial LCA. Then, it proceeds by (a) creating a stack at the next coarseness level (if it does not already exist) from the current one by merging the two keyword subsets (lines 13), and (b) adding the new partial LCA to the partial LCA lists of the subsequent coarseness level (lines 14-16). Finally, if any of the sizes in the popped entry increased by one is smaller than the corresponding size of the parent entry, LCAsz replaces the size and the provenance indicators in the parent entry with the new ones (lines 19-21). For instance, in the example of Figure 4, the first pop action replaces the sizes of 1.1.1.3.2 for John and Smith based on the sizes of their popped child 1.1.1.3.2.2 and sets the provenance indicators of these elements to 2, as the last dewey step of popped 1.1.1.3.2 entry is 2. This way, the parent of a popped node is updated, eventually, with the smallest sizes propagated from its children for all its keyword subsets.

The requirement for disjoint provenance indicator sets (line 9) reflects the requirement of Remark 2, and allows only disjoint partial LCA subtrees to produce new (partial) LCAs. The provenance indicator of an entry element may be set in two ways. First, when a partial LCA size for some key-
word subset is propagated from a node to its parent, the parent’s provenance for the same keyword subset is set to the last Dewey step of the child node (line 21). This Dewey step identifies the root of the partial LCA subtree that contributes this size. For an example, see the first pop action of Figure 4. Second, if the size corresponds to a partial LCA that was formed at the current entry node by two entry elements, the provenance indicator is set to be equal to the union of the provenance indicators of these elements (line 11). An example is the provenance indicator of the partial LCA 1.1.1.3 of {XML, Smith} produced at the third pop action in Figure 4 and inserted in the stack {XS, J} of Figure 5.

4.2 LSAsz analysis

Algorithm LCAsz sequentially processes the instances of the inverted lists of the query keywords. Every instance is pushed in the initial stack and then, partial and full LCAs are progressively formed along the paths of the stack lattice (see Figure 3). At every stack in a path this instance is possibly integrated in a new partial LCA subtree. In the worst case, every instance will provoke an update to all the stacks of the lattice. The number of all possible partitions (stacks) of a set of keywords \{w₁, ..., wₖ\} is given by the Bell number \(B_k\)

\[ B_k = \sum_{i=1}^{k} S(k, i) \]

where \(S(k, i)\) is the Stirling number of the second kind of \(k\) elements partitioned in \(i\) subsets. In other words, for each \(i \in [1..k]\), \(S(k, i)\) is the number of partitions of the coarseness level \(k-i+1\), where the partitions consist of \(i\) keyword subsets. Pushing a partial LCA in a stack may require emptying completely the stack in the worst case. Consequently, for a data tree with depth \(d\), \(d\) pop actions are performed at most for each push action. Each pop of an entry entails at most \(k(k-1)/2\) element combinations to form partial LCAs and at most \(k\) size updates to its parent entry. Therefore, the cost of procedure pop is \(O(k^2)\). When a node at depth \(d\) in the tree is pushed, it requires also at most \(d - 1\) stack push actions of empty entries and one size replacement, taking in total \(O(d)\) time. Thus, if \(|S_i|\) is the number of instances of the inverted list of keyword \(w_i\), the worst case time complexity of LCAsz for processing all the keyword instances is:

\[ O(dk^2B_k\sum_{i=1}^{k}|S_i|) \]

The theoretical worst case scenario is not likely to happen in practice. For example, the worst case assumption that the push of a node in a stack requires complete emptying of the stack, implies that this instance does not share a path with any partial LCA in the stack. However, if all instances are not correlated, popping of nodes does not result in a partial LCA at any level of the tree except from the root. This observation shows that the emptying depth of a stack (factor \(d\)) and the number of partial LCAs induced by a pop action (factor \(k^2\)) are inversely related. From another perspective, updating all the stacks as a result of processing a single keyword instance, implies the presence of all possible keyword arrangements in a subtree of the data tree. Certainly, such a setting imposes high requirements on the number of instances in the subtree and on its height, that cannot be satisfied by all keyword instances.

The complexity shown above is a parameterized time complexity of LCAsz that depends on the total size of the inverted lists \(\sum|S_i|\), the number of query keywords \(k\) and the depth of the data tree \(d\). An important observation is that LCAsz is linear on the size of the inverted lists for a constant \(k\) and \(d\). In contrast, the complexity of previous algorithms is at least \(O(|S|^k)\), preventing them from scaling as the size of the input inverted lists increases. LCAsz still has an exponential dependency on the number of keywords \(k\) because of \(B^k\), but without the involvement of the input size.

5. LCASZ FOR RARE KEYWORDS

In this section, we present a variation of LCAsz, namely LCAszI, that exploits appropriately the dependence of LCAsz performance on the number of keywords to reduce the execution time. The basic idea behind this modification, is the restriction of the number of query keywords to the frequent ones. LCAsz is primarily executed on the frequent keywords only. The rare keywords are taken into account throughout the process, but considered as a new parameter of the algorithm. The following paragraphs outline the rationale of LCAszI for the special case of unique instance keywords. For keywords with few instances (i.e., rare keywords) this procedure is applied repeatedly.

Consider the scenario where a number of rare keywords appears only once in the data tree. In our running example, such keywords are Brown and RDF with LCA 1.1.1.3. A keyword query asking for Brown and RDF, whether also containing other keywords or not, is constraint (Remark 1) to return in the answer node 1.1.1.3 or its ancestors (dashed nodes in Figure 6). This observation places every potential solution to a keyword query containing Brown and RDF on the path between nodes 1 and 1.1.1.3. The partial LCA subtree of Brown and RDF rooted at 1.1.1.3 contributes equally to the size of any full LCA of a query containing these keywords. Thus, algorithm LCAszI proceeds by ignoring any contribution to the size of the LCA by the edges of the partial LCA subtree of the rare keywords. These edges are shown in bold (solid or dotted) in Figure 6. To compensate, before a so-
solution is returned by LCAszI, the size of the subtree of the rare keywords is added to the size of the full LCA. Taking these remarks into consideration, LCAszI is constructed by modifying LCAsz as follows:

- LCAszI executes LCAs on the frequent keywords only.
- During the computation while the root of the partial LCA subtree of the frequent keywords is not on the path between the root and the partial LCA of the rare keywords, it is not returned as a result. Instead, its size is propagated to its parent.
- Before a full LCA \( l \) of the frequent keywords is returned as a solution, its size is incremented by the size of the partial LCA subtree of the rare keywords under \( l \). In Figure 6, the sizes of these subtrees for the rare keywords Brown and RDF under different nodes are shown next to the node ids.
- In computing partial LCA sizes, only the edges outside the subtree of the rare keywords are counted (thin edges in Figure 6).

Ignoring the edges of the rare keywords’ partial LCA subtree (bold solid and dotted edges in Figure 6) does not compromise neither the correctness of the calculation of partial LCA sizes, nor the comparison of partial LCA sizes during the algorithm’s execution. The pruning that LCAsz performs among comparable partial LCAs based on their sizes is done with respect to a specific node. This node corresponds to the current stack entry. Under this node, the partial LCA size of the rare keywords is fixed. For instance, when node 1.1.1 is examined by LCAszI, the size of the partial LCA subtree of Brown and RDF is 5. For the keyword query \{John, Brown, RDF\}, the comparison between the LCA sizes that correspond to the subtrees containing [John,1.1.1.2] or [John,1.1.1.3.1.2] favors the second, either regarded as a comparison of 1 against 0 (i.e., ignoring edges outside the subtree of Brown, RDF) or of 6 against 5.

### 6. EXPERIMENTAL RESULTS

We run experiments to study the efficiency of LCAsz and LCAszI. We also implemented the SAOne algorithm [9], which is the only known algorithm that computes all LCAs of a keyword query with their sizes, in order to experimentally compare it with LCAsz. Algorithms implementing filtering semantics like SLCAs, ELCA, VLCAs and MLCAs are not directly comparable with LCAsz regarding efficiency, as they compute a significantly smaller number of LCAs as an answer to keyword queries.

The experiments were conducted on a virtual machine installed on a Windows 7 system. The virtual machine’s RAM was set to 8GB. The code was implemented in Java and the JVM heap space was left unchanged to the default value of 1.5GB.

We used the real datasets DBLP [11] and NASA [19] and the benchmark auction dataset XMark [25]. Table 1 provides statistics for them.

<table>
<thead>
<tr>
<th></th>
<th>DBLP</th>
<th>XMark</th>
<th>NASA</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>850 MB</td>
<td>150 MB</td>
<td>23 MB</td>
</tr>
<tr>
<td>maximum depth</td>
<td>5</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>average depth</td>
<td>1.97</td>
<td>6.10</td>
<td>5.06</td>
</tr>
<tr>
<td>avg depth per keyword</td>
<td>2.00</td>
<td>4.53</td>
<td>4.99</td>
</tr>
<tr>
<td># nodes</td>
<td>20,966,212</td>
<td>1,666,315</td>
<td>476,646</td>
</tr>
<tr>
<td># keywords</td>
<td>2,509,843</td>
<td>57,775</td>
<td>65,862</td>
</tr>
<tr>
<td># distinct labels</td>
<td>35</td>
<td>74</td>
<td>61</td>
</tr>
<tr>
<td># distinct label paths</td>
<td>152</td>
<td>514</td>
<td>95</td>
</tr>
</tbody>
</table>

Table 1: DBLP, XMark and NASA dataset statistics

smaller but deeper and more complex since their labels may appear in many different label paths. Keywords and nodes distribute almost equally in levels 2-10 in XMark and 2-7 in NASA. The NASA dataset, though, in comparison to XMark contains more keywords in relation to its size.

The keyword inverted lists produced by parsing the XML datasets where stored in a relational database. In order to run experiments with input inverted lists of different sizes we considered inverted lists of keywords of high frequency and truncated them at different lengths. Choosing keywords of high frequency guarantees that many partial LCAs are produced during the processing. Every displayed measurement in the plots is averaged over 10 executions of different queries formed with the frequent keywords. The loading time of the inverted lists was not taken into account, since it is the same for all algorithms and we want to focus on comparing their processing time.

Performance of LCAsz. Table 2 shows the results of a sample of queries execution over the three datasets. These queries contain keywords with thousands of instances in the corresponding datasets and LCAsz answers them in practical time. Notice, for example, \( Q_7^D \) that processes 256,086 instances in about 4 sec returning 1027 results. Query solutions (LCAs) have different sizes (last column) and are returned ranked. The range of the solution sizes reflects the correlation among the keywords searched, with 0 denoting co-existence of keywords in the same node.

Scaling of LCAsz. Figure 7 shows how LCAsz scales when the number of keywords and the number of keyword instances increases in each one of the three datasets. Each curve in the plots corresponds to measurements for queries with the same number of keywords (varying from 2 to 7). Queries with 8 keywords follow the same pattern and are shown in Figure 8. They are not included in the plots of Figure 7 for clarity, since their execution time exceeds the top y axis value of 3.5 sec and their inclusion would sandwich the other curves. The number of instances per keyword ranges from 10 to 1000. This means that for a 7 keyword query the total number of instances ranges from 70 to 7000.

The displayed results confirm the linear behavior of LCAsz execution time with respect to the input size for a given number of keywords, which was discussed in Section 4.2. The slope difference between two consecutive lines shows how the execution time of LCAsz scales on the total number of keyword instances for two consecutive query lengths. For


<table>
<thead>
<tr>
<th>query</th>
<th>execution time</th>
<th># keywords</th>
<th>total # kw instances</th>
<th># results</th>
<th>LCA sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q^1$ - (information system security)</td>
<td>0.97 sec</td>
<td>3</td>
<td>250,886</td>
<td>1,027</td>
<td>0 - 4</td>
</tr>
<tr>
<td>$Q^2$ - (online analytical processing)</td>
<td>0.86 sec</td>
<td>4</td>
<td>48,669</td>
<td>22</td>
<td>0 - 4</td>
</tr>
<tr>
<td>$Q^3$ - (database query language)</td>
<td>0.96 sec</td>
<td>3</td>
<td>55,250</td>
<td>120</td>
<td>0 - 4</td>
</tr>
<tr>
<td>$Q^4$ - (semantic web services automatic composition)</td>
<td>12.91 sec</td>
<td>5</td>
<td>101,420</td>
<td>10</td>
<td>0 - 4</td>
</tr>
<tr>
<td>$Q^5$ - (spatial GIS applications)</td>
<td>1.11 sec</td>
<td>3</td>
<td>66,201</td>
<td>13</td>
<td>0 - 4</td>
</tr>
<tr>
<td>$Q^6$ - (sensor networks power consumption)</td>
<td>3.63 sec</td>
<td>4</td>
<td>129,608</td>
<td>15</td>
<td>0 - 4</td>
</tr>
<tr>
<td>$Q^7$ - (network computing algorithms)</td>
<td>2.25 sec</td>
<td>3</td>
<td>129,841</td>
<td>26</td>
<td>0 - 4</td>
</tr>
<tr>
<td>$Q^8$ - (database query language)</td>
<td>0.96 sec</td>
<td>3</td>
<td>55,250</td>
<td>120</td>
<td>0 - 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XMark</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q^1$ - (province school female student)</td>
<td>0.80 sec</td>
<td>4</td>
<td>15,742</td>
<td>11</td>
<td>7 - 12</td>
</tr>
<tr>
<td>$Q^2$ - (province school female student)</td>
<td>2.20 sec</td>
<td>5</td>
<td>19,087</td>
<td>11</td>
<td>7 - 16</td>
</tr>
<tr>
<td>$Q^3$ - (cash shipping Europe)</td>
<td>1.05 sec</td>
<td>3</td>
<td>32,929</td>
<td>43</td>
<td>3 - 9</td>
</tr>
<tr>
<td>$Q^4$ - (charges shipping location United States)</td>
<td>35.55 sec</td>
<td>5</td>
<td>114,319</td>
<td>12,308</td>
<td>2 - 9</td>
</tr>
<tr>
<td>$Q^5$ - (certainly apply leading expense offers approved)</td>
<td>3.19 sec</td>
<td>6</td>
<td>5,923</td>
<td>11</td>
<td>9 - 14</td>
</tr>
<tr>
<td>$Q^6$ - (approved school expense offers student apply)</td>
<td>3.89 sec</td>
<td>6</td>
<td>8,975</td>
<td>11</td>
<td>9 - 14</td>
</tr>
<tr>
<td>$Q^7$ - (cash payment delay order)</td>
<td>3.40 sec</td>
<td>4</td>
<td>47,928</td>
<td>250</td>
<td>2 - 10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NASA</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q^1$ - (bright stars photometric measurements)</td>
<td>0.62 sec</td>
<td>4</td>
<td>10,782</td>
<td>27</td>
<td>0 - 18</td>
</tr>
<tr>
<td>$Q^2$ - (estimated diameter planetary objects)</td>
<td>0.19 sec</td>
<td>4</td>
<td>2,814</td>
<td>9</td>
<td>0 - 10</td>
</tr>
<tr>
<td>$Q^3$ - (stars position meridian circle)</td>
<td>0.54 sec</td>
<td>4</td>
<td>11,052</td>
<td>27</td>
<td>0 - 9</td>
</tr>
<tr>
<td>$Q^4$ - (spectral photovoltaic photoconductive stars measurements)</td>
<td>1.11 sec</td>
<td>5</td>
<td>11,237</td>
<td>3</td>
<td>0 - 7</td>
</tr>
<tr>
<td>$Q^5$ - (spectrometric equipment calibration spectral band)</td>
<td>0.63 sec</td>
<td>5</td>
<td>5,453</td>
<td>3</td>
<td>0 - 7</td>
</tr>
<tr>
<td>$Q^6$ - (stellar spectral classification stars)</td>
<td>0.84 sec</td>
<td>4</td>
<td>12,329</td>
<td>91</td>
<td>0 - 17</td>
</tr>
<tr>
<td>$Q^7$ - (stellar spectral classification stars emission)</td>
<td>1.99 sec</td>
<td>5</td>
<td>13,718</td>
<td>35</td>
<td>0 - 21</td>
</tr>
</tbody>
</table>

Table 2: DBLP, XMark and NASA queries

DBLP this transition is smoother than for the XMark and NASA datasets. This happens due to the shallowness of DBLP that bounds the number of possible pushes and pops in the stacks of LCA as to the maximum of 5 (1.07 on average as shown in Table 1).

Comparison with a previous algorithm. Figure 8 presents a comparison of the execution times of LCAz and algorithm SAOne proposed in [9]. The main algorithm SA of [9] aims at identifying minimum connecting trees of a set of keywords with distances (DMCTs), which are homomorphic with respect to the keyword set, and grouping them together into grouped distance MCTs (GDMCTs). It returns only GDMCTs not exceeding a given threshold size. To this end the sizes of all GDMCTs are also computed. Algorithm SAOne is a variation of SA that computes only LCAs, thus avoiding the merge of homomorphic DMCTs. In this setting, one DMCT suffices for returning an LCA as a result. In our implementation of SAOne, we let the smallest LCA sizes of an LCA to contribute its size to that LCA. Without a threshold SAOne returns all DMCTs and their sizes. For the sake of comparison we adjusted SAOne appropriately, so that it supports keyword co-occurrence in the internal nodes of the data tree. We also adapted it to work with Dewey codes instead of regional encoding. Finally, there should be noted, that SAOne benefits from an additional inverted list of pairwise common ancestors computed in advance. This precomputation allows bypassing internal nodes from processing (i.e., stack pushing and popping) when they are not partial LCAs.

Figure 8 shows that LCAz performs in most cases better than SAOne for all three datasets. Notice that the execution time in the y axis is presented in logarithmic scale. SAOne experiments are illustrated with dotted curves. SAOne performs slightly better on the DBLP dataset when the number of keywords and instances is small. Increasing the number of instances the execution time of SAOne grows very fast, as its complexity depends on the input length to the power of twice the number of keywords. For example, considering the queries of Table 2, SAOne needs 43 sec against 0.96 sec for LCAz for query $Q^1$ over 55,250 instances of DBLP.

As the depth of the datasets grow, the number of possible
The benefit of LCAszI over LCAsz. Figure 9 illustrates the experiments on LCAszI performance. In particular, the plots show LCAszI time benefit over LCAsz in the case of 8 keyword queries with rare keywords that vary from 1 to 6. The experiments in this case were run in the setting that our scaling experiments did. They were repeated for different sets of 8 frequent keywords. For every keyword 100 instances were chosen. Among the frequent keywords, the rare ones were randomly selected and for those an equal number of instances was chosen from their inverted lists varying from 1 to 10. In order to let the rare instances distribute homogeneously, we chose 1-10 instances from each list spread along the whole length of the keyword lists. For each selection of rare keywords and instances, LCAsz and LCAszI were repeated 10 times.

The benefit gained from the execution of LCAszI in most cases is obvious. The numbers on the bars indicate the total number of instances that shows a benefit. For example, a selection of 3 rare keywords of 4 instances each results in a total of 12 rare instances but in another instance arrangement, e.g., 3 rare keywords with 1, 1 and 64 instances, the rare keywords’ subtrees are again 64 (1∗1∗64) but the total number of their instances this time is 66. In this sense, Table 9(c) shows the maximum number of rare keyword instances making LCAszI beneficial.
rare instances that can favor the use of LCAszI over LCAsz for a specific number of rare keywords.

The benefit of using LCAszI increases as the number of rare keywords compared to the total keyword number increases, too. This behavior is justified by the reduction of the coarseness levels and stacks in the execution of LCAsz since only stacks for the processing of frequent keywords are created. For instance, for an 8 keyword query with 6 rare keywords the computation constructs a lattice of stacks for only 2 keywords. XMark can benefit from LCAszI more than DBLP, as the Table 9(c) shows. From another perspective, this is indicated also by the smoother slope increase in the scaling plots of LCAsz (Figure 7) for DBLP in comparison to that of XMark. Experiments of LCAsz on NASA dataset for rare keywords were not included in the paper because of lack of space, but the results are similar to those of XMark.

7. CONCLUSION AND FUTURE WORK

We introduced LCA size as a measure of relevance of the results of a keyword query on tree-structured data, and presented a novel multi-stack based algorithm that returns all results of a keyword query on tree-structured data, and presented a variation of our algorithm by filtering LCA size with value-based semantics and with the creation of LCAs since only stacks for the processing of frequent keywords are created. For instance, for an 8 keyword query with 6 rare keywords the computation constructs a lattice of stacks for only 2 keywords. XMark can benefit from LCAszI more than DBLP, as the Table 9(c) shows. From another perspective, this is indicated also by the smoother slope increase in the scaling plots of LCAsz (Figure 7) for DBLP in comparison to that of XMark. Experiments of LCAsz on NASA dataset for rare keywords were not included in the paper because of lack of space, but the results are similar to those of XMark.

Our current work focuses on extending the algorithms presented here to support top-K keyword query answering. We are also planning to study how our algorithm can be combined with LCA filtering semantics and with value-based statistics approaches for ranking the results.

8. REFERENCES