Top-k-size keyword search on tree structured data

Aggeliki Dimitriou a, Dimitri Theodoratos b,*, Timos Sellis c

a School of Electrical and Computer Engineering National Technical University of Athens, Greece
b Department of Computer Science New Jersey Institute of Technology, United States
c School of Computer Science and Information Technology, RMIT University, Australia

Abstract

Keyword search is the most popular technique for querying large tree-structured datasets, often of unknown structure, in the web. Recent keyword search approaches return lowest common ancestors (LCAs) of the keyword matches ranked with respect to their relevance to the keyword query. A major challenge of a ranking approach is the efficiency of its algorithms as the number of keywords and the size and complexity of the data increase. To face this challenge most of the known approaches restrict their ranking to a subset of the LCAs (e.g., SLCAs, ELCAs), missing relevant results.

In this work, we design novel top-k-size stack-based algorithms on tree-structured data. Our algorithms implement ranking semantics for keyword queries which is based on the concept of LCA size. Similar to metric selection in information retrieval, LCA size reflects the proximity of keyword matches in the data tree. This semantics does not rank a predefined subset of LCAs and through a layered presentation of results, it demonstrates improved effectiveness compared to previous relevant approaches. To address performance challenges our algorithms exploit a lattice of the partitions of the keyword set, which empowers a linear time performance. This result is obtained without the support of auxiliary precomputed data structures. An extensive experimental study on various and large datasets confirms the theoretical analysis. The results show that, in contrast to other approaches, our algorithms scale smoothly when the size of the dataset and the number of keywords increase.

1. Introduction

Tree-based structures (e.g., XML, JSON, YAML) are a widely adopted format for exporting and exchanging data on the web. Keyword search is the most popular technique for retrieving information from the web because it frees the users from (a) mastering a complex query language (e.g., XQuery), and (b) having full knowledge of the schemas of the data sources they want to query. In contrast to keyword search on flat text documents, keyword search on XML (or other tree-structured) data returns not whole documents but appropriately selected fragments of XML trees that contain matches to all the keywords [28,14]. A large number of publications elaborate on the form [14,24,4,5,8,26,23] and the meaningful instances of these result fragments [14,10,22,31,15,29,18,32,25,26,30,17] in the input XML tree. Usually, the query results are the minimum connecting trees that contain one instance of every keyword in the query. These minimum connecting trees are represented by their root which is the lowest common ancestor (LCA) of the included keyword instances. Approaches that select and return as answer to keyword queries a subset of the LCAs in the XML tree are called filtering because they filter out LCAs that are considered irrelevant [10,22,31,18,32]. 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 [30].
A better approach would rank the LCAs placing on top those that are considered more relevant to the query. Ranking the LCAs greatly improves the usability of the system. Most ranking approaches are based on strategies employed for flat text documents (e.g., tfidf or PageRank) adapted to the hierarchical nature of the XML trees [14,10,30,2]. Recognizing the fact that users are usually interested in a small number of query results, some papers recently develop top-k algorithms for keyword search over XML data [19,8,21]. The goal of the top-k algorithms is to rank and select the top-k results without explicitly producing and ranking all of them.

Current approaches for keyword search on tree-structured data face a number of problems:

**Problem 1: Performance scalability.** The number of LCAs for a given keyword query can be very large. Even though multiple query matches can share the same LCA, this number can, in the worst case, be exponential on the size of the query (number of keywords). The complexity of previous algorithms that process and possibly rank the totality of the LCAs depends on the product of the size of the keyword inverted lists [15,30,23]. Consequently, such algorithms do not scale satisfactorily when the size of the dataset and the number of keywords increase.

**Problem 2: Dependence from additional auxiliary data structures.** In order to address the performance scalability problem a number of approaches rely on the construction of auxiliary data structures, on top of the keyword inverted lists (e.g., B-+tree [32], ranked Dewey inverted list and B+-tree [14], data summary index [23], hash count index [33]). Moreover, many approaches rely on additional auxiliary data structures and statistical information even for efficiently implementing query semantics (e.g., inter-connection index [10,9], normalized total correlation [30]). Building these auxiliary structures requires producing and storing the inverted lists and the auxiliary data structures of the dataset. This process is not only time consuming but also renders these approaches impractical to a number of applications including streaming applications.

**Problem 3: Quality of the answer.** In order to avoid producing a large number of LCAs the different ranking and top-k approaches proposed [14,10,8,21] produce and rank not all the LCAs but only a small subset defined by filtering semantics (e.g., SLCA [31,15,29,25], ELCA [14,32,33]). This strategy, even though computationally appealing, is semantically insufficient since, despite the potential quality of the ranking criteria, it penalizes the query answer with the deficiencies of the corresponding filtering semantics. For instance, if relevant results are missed by the filtering semantics they cannot be recovered and presented to the user, no matter how good or efficient the ranking technique is.

**Problem 4: User interface of top-k approaches.** In order to support the users in coping with a possibly large number of results but also for performance reasons, recent approaches return only top-k results to keyword queries [8,21]. However, the selection of k by the user at query time is a tricky issue: a selection of a small k may miss relevant results while a selection of a large k may overwhelm the user with a large number of irrelevant results and unnecessarily increase the response time. Using a good ranking function could address the problem of the multitude of returned results but it does not resolve the performance issue. Successfully selecting the appropriate k requires knowledge of the number of results, which depends on the size and structure of the dataset, the number of query keywords and the keyword frequencies in the dataset. Requiring the user to have detailed statistical information about the dataset and be able to apply complex techniques for estimating the number of relevant results defies the reason for using such a simple query language as keyword queries.

**Our approach.** In this paper, we present novel, efficient top-k algorithms which compute results of top-k LCA sizes. This approach contrasts to traditional top-k approaches which focus on computing top-k results. The concept of LCA size introduced in this paper reflects the proximity of keywords in tree-structured data and, similar to the concept of keyword proximity in the IR domain, is used here as a relevance criterion for the results of a keyword query. The efficiency of our algorithms is achieved by exploiting a lattice of keyword partitions of the query keywords. The paths of the lattice are used in gradually combining keyword instances into partial LCAs, allowing the exclusion of combinations of other instances of the same keywords with larger size before they contribute to the formation of full LCAs. This technique avoids the exhaustive computation of all keyword instance combinations in finding the LCAs of top-k sizes. As a consequence, our algorithms scale smoothly when the size of the dataset increases and tackle successfully the performance scalability issue (Problem 1). Interestingly, our algorithms achieve efficiency without recurring to the construction of auxiliary data structures, this way avoiding the preprocessing phase required by other approaches (Problem 2).

Our algorithms implement ranking semantics and consider for ranking the full set of LCAs. Therefore, they do not suffer from deficiencies (low precision or recall) of previous ranking and top-k approaches which are restricted to a predefined, structurally determined subset of LCAs (Problem 3). The returned results are grouped in LCA size layers which are ranked on LCA size. The layers can have a varying number of results. This layered computation relieves the user from providing with the query an appropriate k for top-k result computation (Problem 4). The top-1-size layer contains results of highest proximity and shows high precision while recurrence to a subsequent layer is possibly needed only in case increased recall is desired.

**Contribution.** The main contributions of our paper are the following:

- We design novel, efficient multi-stack based algorithms, that exploit a lattice of stacks representing the different partitions of query keywords. Our algorithms compute: (a) keyword search results below a given LCA size threshold, (b) results of top-k LCA sizes and (c) top-k results.
- In contrast to previous approaches, ours does not involve auxiliary index structures and therefore it can be exploited on datasets which have not been preprocessed.
- We analyze our algorithms and show that for a fixed number of query keywords their performance is linear on the size of the input keyword inverted lists. This behavior contrasts with that of previous algorithms,
which also consider the full set of LCAs, whose complexity depends on the product of the size of the input inverted lists.

- We introduce the concept of LCA size and we use it to define layered ranking semantics called Tight LCA (TLCA) semantics. The layers contain results of the same LCA size. Selecting top-k-size LCAs defines filtering semantics for keyword queries where the results are grouped into ranked layers.

- We experimentally evaluate our algorithms with high frequency query keywords on large, real and benchmark datasets. Our experiments confirm the theoretical analysis and show that our algorithms outperform previous ones and scale smoothly when the size of the input inverted lists and the number of keywords increase.

- We also ran experiments to evaluate the effectiveness of TLCA. Our semantics and its layered filtering capability demonstrate improved performance compared to relevant approaches that rely on structural information to restrict the LCAs.

Outline. The next section reviews related contributions. Section 3 introduces preliminary concepts, the notion of LCA size and TLCA semantics. In Section 4, we describe our size threshold, top-k-size and top-k algorithms and analyze their complexity. Section 5 presents our experimental study on the efficiency of our algorithms and the effectiveness of our approach. Section 6 concludes and suggests future research directions.

2. Related work

A number of approaches for assigning semantics to keyword queries on XML trees identify candidate keyword query results among the lowest common ancestors (LCAs) of keyword instances [28] of the data tree. Some of them rely purely on hierarchical criteria, disregarding semantic information (node labels). According to the smallest LCA (SLCA) semantics [31,25] the valid LCAs do not contain other descendant LCAs of the same keyword set. Relaxing this semantics, exclusive LCA (ELCA) semantics was introduced in [14] and later formally in [32]. In addition to SLCA, ELCA qualifies also as relevant LCAs those that are ancestors of other LCAs as long as they refer to a different set of keyword instances.

The semantic approaches valuable LCAs (VLCAs) [10,18] and meaningful LCAs (MLCs) [22] attempt to capture the user intent by exploiting the labels that appear in the paths of the subtree rooted at an LCA. All these semantics are restrictive and depending on the case, they may demonstrate low recall rates as shown in [30].

None of the previous semantics inherently supports any ranking capability for presenting relevant qualified results. In order to fill this lack, (i) structural and semantic correlations [14,19,8,30,10,1], (ii) statistical measures [14,10,19,8,20,30] and (iii) probabilistic models [30,27,21] are exploited in the literature, sometimes in combination with the aforementioned semantics.

In this work, we consider the concept of LCA size as a metric for assessing the relevance of keyword search results to the keyword query. The concepts of size and distance have been used in the past with various meanings in the XML keyword search literature: pairwise keyword distances [10,19], distances among keyword matches and LCAs [14,10] and sizes of minimum connecting trees of keyword instances [15]. Size and distance metrics have been also used in the context of keyword search in relational and graph databases [16,13,12]. However, to the best of our knowledge, no other approach considers the LCA size as a relevance metric in XML keyword search.

The algorithms that compute LCAs for keyword queries depend on the query semantics and the ranking method used. They are designed to take advantage of the filtering semantics by pruning irrelevant LCAs early on in the computation. In [14] 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 [6] and of textual keyword proximity in the subtree of the ranked ELCA. In [31], two efficient algorithms for computing SLCA are introduced, exploiting special structural properties of SLCA. In addition, the authors suggest an extension of their basic algorithm, so that it returns all LCAs by augmenting the set of already computed SLCA. However, this approach does not compute LCA subtree sizes as we do in this paper. In [29] another algorithm for efficiently computing SLCA for both AND and OR keyword query semantics is developed. The Indexed Stack [32] and the Hash Count [33] algorithms improve the efficiency of [14] in computing ELCA. Finally, [4,5] elaborate on sophisticated ranking of candidate LCAs aiming primarily on effective keyword query answering.

The approach in [15] is the most relevant to our work. This approach returns all minimum connecting 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 threshold algorithm with algorithm SAOne which is a variation of SA also introduced in [15] and returns all LCAs. We also compare our top-k algorithms with adaptations of SAOne.

In [11] we present a multi-stack algorithm which also exploits multiple stacks to compute LCAs and their sizes.
This algorithm does not compute results restricted by a threshold or top-k results which is the focus of this paper.

3. Preliminaries and tight LCA semantics

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 [7], which encodes tree nodes according to a descendant relationship. Metonymy is encoded with its 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 [7], which encodes tree nodes according to a descendant relationship. Metonymy is encoded with its 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).

A keyword query Q is a set of keywords: \(Q = \{k_1, \ldots, k_n\}\). 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 connecting tree, \(M_k\), of a set \(S\) of nodes in a data tree \(D\) is the minimum subtree \(D_k\) of \(D\) that contains all nodes in \(S\). The root of \(D_k\) is the lowest common ancestor (LCA) of the nodes in \(S\), denoted \(\text{lca}(S)\). The size of \(D_k\) is the number of its edges. Let \(l\) be a set of instances of keywords in \(Q\). If \(l\) contains one instance for every keyword in \(Q\), we call \(l\) 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 now introduce the concept of LCA size. Let \(l\) and \(l'\) be two different but not necessarily disjoint instances of \(Q\) in an XML tree \(D\). Clearly, their minimum connecting trees \(M_l\) and \(M_{l'}\) 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 among the minimum connecting trees of the instances of \(Q\) that are rooted at \(l\).

For example, in the XML tree of Fig. 1, the size of the LCA 1.1.1.3 of query \(Q = \{\text{XML, John, Smith}\}\) is 4 since there are exactly two minimum connecting 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.

Semantics. We now introduce the Tight LCA (TLCA) ranking semantics for keyword queries, which are based on the concept of LCA size. The term “tight” refers to the fact that the smaller the size of an LCA the more tightly the LCA and its instances are connected together. According to the TLCA semantics, the answer of a query \(Q\) over a tree \(D\) is the set of all the LCAs of \(Q\) on \(D\) ranked in ascending order of their size. More formally, a result of a query \(Q\) over an XML tree \(D\) as a pair \((l, s)\) of an LCA \(l\) of \(Q\) and its size \(s\). The answer \(A\) of \(Q\) over \(D\) is the list of all the results of \(Q\) over \(D\) ranked on their size: \(A = [(l_1, s_1), (l_2, s_2), \ldots], s_1 \leq s_j, 1 < j\). If two results have the same size, their relative order in the answer is indifferent. For example, the answer of \(Q = \{\text{XML, John, Smith}\}\) over the XML tree of Fig. 1 is \(A = \{(1.1.1), 2, (1.1.1.3), 4, (1.1.4)\}\).

TLCA semantics is inspired by the concept of keyword proximity in the IR domain [3] and is based on the intuition that the quality of an LCA depends on the proximity of a set of keyword instances. TLCA not only ranks the returned results but also groups them in layers defined by results of the same size. Therefore, results that have the same relevance to the query are grouped in the same layer. Retaining only the top-k size LCAs, we can also define layered filtering semantics for keyword queries. The top-1-size layer contains the most relevant results. In most cases, the top-2 layers are sufficient for collecting all relevant results.

We next show on an example XML tree how the filtering TLCA semantics compare with SLCA and ELCA semantics. Unlike VLCA [10,18] and MLCA [22] semantics, TLCA, SLCA and ELCA disregard the labels of the nodes in the tree and take into account only structural information. Consider the bibliographic database that records the papers of conferences, shown in Fig. 2, and the query \(\{\text{top-k, LCA, John, Smith}\}\) issued against this database. It is reasonable to assume that the user looks for papers authored by John Smith related to the subjects of top-k techniques and LCA. The most relevant result in the database of Fig. 2 is the paper node 1.1.1, which represents a paper on top-k and LCA authored by John Smith and to a lesser extent the paper node 1.1.2, which represents a paper on top-k search authored by John Smith referencing a paper on smallest LCA. ELCA misses the result paper 1.1.2 in favor of the
Let \( l \) be an instance of a query \( Q \), \( l = \text{lca}(l) \) and \( I_1, \ldots, I_n \) be subsets of \( I \) such that \( \bigcup I_i = I \). Let also \( l_i = \text{lca}(I_i \cap I) \), \( i = 1 \ldots n \), be partial LCAs of \( Q \). Then, \( l = \text{lca}(I_1, \ldots, I_n) \).

Two trees \( D \) and \( D' \) are called disjoint if they do not share any edges.

**Remark 2.** Two trees \( D \) and \( D' \), rooted at the same node \( r \), are disjoint if \( E_r \cap E_r' = \emptyset \), where \( E_r \) (resp. \( E_r' \)) is the set of outgoing edges of \( r \) in \( D \) (resp. \( D' \)).

**Remark 3.** Let \( D \) be a tree rooted at \( r \) and \( D_1, \ldots, D_n \) be disjoint subtrees of \( D \) also rooted at \( r \), which altogether contain all edges of \( D \). Then, \( \text{size}(D) = \sum \text{size}(D_i) \).

The minimum connecting tree of a set of keyword instances of a query instance \( l \) with root the XML \( p \), extended with the path connecting \( p \) with an ancestor \( n \), defines a partial LCA subtree of \( l \) rooted at node \( n \). As a consequence of Remark 3, the size of an LCA \( l \) can be computed based on the sizes of partial LCA subtrees rooted at \( l \).

**Example 1.** Let query \( Q = \{ \text{XML}, \text{Brown}, \text{RDF}, \text{Smith} \} \). Node 1.1.1.3 is the LCA of the instance \( l = \{1.1.1.3.1.1, 1.1.1.3.1.2, 1.1.1.3.2.1, 1.1.1.3.2.2 \} \) of \( Q \) in Fig. 1. Its size based on the unique minimum connecting tree of these keyword instances is equal to 6. All possible combinations of one to three of these keyword instances define a partial LCA subtree rooted at LCA 1.1.1.3. Node 1.1.1.3 is the LCA of any two or more of these subtrees (Remark 1). However, the correct size of LCA 1.1.1.3 can be computed only from the left subtree of 1.1.1.3 (with instances XML, Brown) and the right one (with instances RDF, Smith). They are the only disjoint ones (Remark 2), and therefore the size of LCA 1.1.1.3 is the sum of their sizes.

### 4. Computing \( k \)-size LCAs

In this section, we present algorithms which implement TLCA semantics. We first describe algorithm \( T-LCAs \), which for a given keyword query \( Q \) and a size threshold \( T \), returns the LCAs in an XML tree of size \( T \) or less. This algorithm constitutes the basis for the top-\( k \) algorithms presented in Section 4.3.

#### 4.1. Threshold algorithm \( T-LCAs \)

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

Based on a straightforward interpretation of Definition 1, the estimation of the size of an LCA implies the computation of all minimum connecting trees rooted at this LCA along with their sizes. Algorithm \( T-LCAs \), however, avoids exhaustively examining all minimum connecting trees of each LCA. In order to do so, it progressively combines keyword instances into partial and full LCAs bottom-up in the data tree.

\( T-LCAs \) combines 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 based on Remark 3. The sizes of partial LCAs are propagated upwards to their ancestors to contribute to the size of possible full LCAs. During the propagation, the size of a partial LCA \( l \) at \( n \) reflects the number of upward steps to an ancestor node \( n \). For simplicity, the size of partial LCA \( l \) at \( n \) refers to the size of \( l \) increased by the distance of \( n \) from \( l \). If a partial LCA size at a node exceeds the threshold, the partial LCA is excluded from further consideration. At every node in the upward propagation, 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 is recorded.
Algorithm 1. T-LCAs

1. **LCAs**\((k_1, \ldots, k_n; \text{ keyword query}, \text{ invL}: \text{ inverted lists}, T: \text{ size threshold})\)
2. \(\text{kwSubsets} = \{(k_1), (k_2), \ldots, (k_n)\}\)
3. \(\text{buildLattice}() /* \text{ constructs empty stacks of the lattice and updates kwSubsets */}\)
4. while current\(\text{Node} = \text{getNextNodeFromInvertedLists}()\) do
5.    coarseness\(\text{Level} = 1, \text{ size}=0, \text{ provenance}=\emptyset\)
6.    pLCA = \text{newPartialLCA(currentNode.ID, currentNode.kwSubset, size, provenance)}
7.    addPartialLCA(1, pLCA)
8.   while partialLCAs\(\text{lists contains partialLCAs for coarsenessLevel} do\)
9.     while partialLCA = partialLCAs\(\text{lists(coarsenessLevel).next()}\) do
10.        for every stack of coarseness\(\text{Level containing partialLCA.kwSubset} do\)
11.            if partialLCA.size \(\leq T\) /* only for top-k mode */
12.                then \(\text{push(stack, partialLCA.ID, partialLCA.kwSubset, partialLCA.size)}\)
13.                if coarseness\(\text{Level} < n\) then
14.                    if partialLCA.size \(\leq T\) /* only for top-k mode */
15.                        then \(\text{addPartialLCA(coarsenessLevel+1, partialLCA)}\)
16.        empty\(\text{Stacks}()\)
17.     addPartialLCA(coarseness\(\text{Level}, \text{partialLCA}\))
18.     if (partialLCA.ID, partialLCA.kwSubset) not in partialLCAs\(\text{lists}(cL) then\)
19.        insert partialLCA into partialLCAs\(\text{lists(coarsenessLevel)}\)
20.  else if current (partialLCA.ID, partialLCA.kwSubset) entry size < partialLCA.size then
21.        replace with partialLCA
22.        if stack.dewey not ancestor of node\(\text{ID} do\)
23.            pop(stack)
24.        while stack.dewey \(\neq \text{node\(\text{ID} do\)
25.            addEmptyRow(stack) /* updating stack.dewey until it is equal to node\(\text{ID */}
26.            replaceSizeIfSmallerWith(stack.topRow, kwSubsetColumn, size)
27.        empty\(\text{Stacks}()\)
28.    foreach coarseness\(\text{Level} do\)
29.        if partialLCAs\(\text{lists(coarseness\(\text{Level}) is not empty then\)
30.            while partialLCA = partialLCAs\(\text{lists(coarseness\(\text{Level).next()} do\)
31.                for every stack of coarseness\(\text{Level containing partialLCA.kwSubset do\)
32.                    if partialLCA.size \(\leq T\) then
33.                        push(stack, partialLCA.ID, partialLCA.kwSubset, partialLCA.size)
34.                    if coarseness\(\text{Level} < n\) and partialLCA.size \(\leq T\) then
35.                        addPartialLCA(coarseness\(\text{Level+1, partialLCA})
36.            empty\(\text{Stacks}()\)
37.        foreach stack of coarseness\(\text{Level do}\)
38.            repeat
39.                pop(stack,T)
40.            until top entry contains only propagated or empty elements and the other entries are empty;

In this framework, the problem of finding all LCAs under a size threshold translates into examining all existing arrangements of keywords under a candidate LCA and retaining those, whose minimum size does not exceed the threshold. 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 computation of a number of minimum connecting trees of keyword instances is possibly avoided.

A lattice of keyword partitions. The possible arrangements of a keyword set reflect the partitions of the set. We can define a refinement relation \(\leq\) on the partitions of a set: \(P_1 \leq P_2\) 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 \(P_1 \leq P_2, P_1\) 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. Fig. 3 shows the Hasse diagram of the lattice of the keyword set \{XML, Author, John, Smith\}. For \(k\) keywords the lattice consists of \(k\) coarseness levels. At every coarseness level all partitions have the same number of elements \((k, k/2^0, \ldots, 1)\).

The unique partition of the first coarseness level is the source partition that contains singletons for all keywords. The unique partition of the last coarseness level is the sink partition that contains a set with all the keywords. Every partition in one level can be produced from a partition of
the previous level by unioning two of its elements. A partition in one level may be produced by different partitions of the previous level. In our context, every path from the source partition to the sink partition represents a unique way of combining keyword instances to partial LCAs to full LCAs. Algorithm T-LCAsz maintains a separate stack for each partition. Keyword instances of a query are pushed in the source stack and query results (full LCAs) are popped from the sink stack.

The stacks. Every node in the lattice is a multicolumn stack (example stacks are shown in Figs. 4(b), 5 and 6). The columns of a stack are named by the keyword subsets of the lattice node. Each stack entry corresponds to a tree node, and its previous entry in the stack corresponds to its parent node. All entries are identified by their Dewey codes (shown on the left of the stacks in Figs. 4(b), 5 and 6). For each keyword subset S, a stack entry n contains: (a) the size s of the partial LCA of S at tree node n and (b) its provenance numbers (namely Dewey steps) identifying one or more of the outgoing edges of tree node n. These edges indicate which partial LCA subtree of S rooted at n contributes the size s.

For instance, Fig. 4(b) shows different states of the source stack of the lattice for the query Q={XML, John, Brown} on the data tree of Fig. 4(a). Focusing on the second state of this stack we can see that it currently holds the node with Dewey code 1.1.1.2. The size of the top entry element at column John is 0, indicating that node 1.1.1.2 is an instance of John. The other element stored in the stack is located at the column for XML and corresponds to node 1.1.1. The LCA size 1 indicates that node 1.1.1 is a partial LCA for XML with size 1. The provenance 1 (shown in the shaded box below the size) further indicates that this partial LCA size is contributed by the subtree of its first child (i.e., node 1.1.1.1).

Algorithm outline. T-LCAsz iterates over the inverted lists of the keywords in document order (lines 4–18) after initializing (line 3) the lattice that corresponds to the query keywords. Each instance is pushed (procedure push(), line 25) into the lattice, as a partial LCA of a single keyword instance, starting from the source stack whose elements contain sizes for singleton keyword sets only. Fig. 4(b) shows the source stack of the lattice for the keyword query Q={XML, John, Brown} on the example tree of Fig. 1 (repeated also in Fig. 4(a) for convenience). The keyword instances are shown in bold in Fig. 4(a). Fig. 4(b) illustrates the states of the source stack after all but the last keyword instance are pushed into the stack.

The algorithm proceeds level by level, each level corresponding to a coarseness level of the lattice. At each stage, partial LCAs of coarser keyword partitions are formed by combining partial LCAs from the previous coarseness level, as long as their sizes do not exceed the given size threshold T (procedure pop(), line 9).

Fig. 5(a) continues the running example of Fig. 4 and depicts the different snapshots of the source stack while processing the instance 1.1.2.1 of the keyword XML. The first snapshot of the source stack in Fig. 5(a) is the last source stack state of Fig. 4(b). The top entry in this snapshot corresponds to the node author 1.1.3.2.2 of the data tree and the rest of the stack entries correspond to its ancestor nodes also shaded in the tree of Fig. 4(a). Pushing the new XML instance into the source stack triggers the popping of all non-ancestor nodes of the new XML instance from the stack. During the process, new partial LCAs are formed (shown in shaded boxes in Fig. 5(a)). These partial LCAs are then pushed into the stacks of the next coarseness level (Fig. 5(b)).

Procedure 1. pop

```plaintext
begin
    cols = stack.columns
    popped = stack.pop()
    if cols = 0 then
        // addResult() updates T only in top-k mode
        T = addResult(stack.dewey, popped[0].size)
        // Produce new LCAs from two partial LCAs
        if cols > 1 then
            for i=0 to cols do
                for j=i to cols do
                    if popped[i] and popped[j] contain sizes and popped[i].provenance \ popped[j].provenance = ∅ and
                    popped[i].size + popped[j].size ≤ T then
                        newLCA = newPartialLCA(stack.dewey, newKWSubsets, newSize, newProvenance)
                        if cardinalityOf(newKWSubsets) = stack.coarsenessLevel+1 then
                            addPartialLCA(stack.coarsenessLevel+1, newLCA)
                        else
                            createNewStack(stack, stack.coarsenessLevel+1, i, j, newKWSubsets) // if it does not exist
                            pLCA = newPartialLCA(stack.dewey, newKWSubsets, newSize, newProvenance)
                            if cardinalityOf(newKWSubsets) = stack.coarsenessLevel+1 then
                                addPartialLCA(stack.coarsenessLevel+1, pLCA)
                    else
                        removeLastDeweyStep(stack.dewey)
```
Newly constructed partial LCAs are added to a partial LCA list of the next coarseness level before being actually processed at that level (line 6 of algorithm T-LCAsz and line 16 of procedure pop). The usage of partial LCA lists between consecutive coarseness levels prevents multiple updates of a single stack by LCAs coming from different stacks of the previous coarseness level (see, for instance, partitions with multiple incoming edges in the lattice of Fig. 3).

The computation of new LCAs at a certain node is performed when this node is popped from a stack (procedure pop(), lines 9–16), which indicates that all its subtrees are already processed. In the running example of Fig. 5(a), the partial LCA 1.1.1.3 for keywords {XML, John} is produced from the partial LCAs of {XML} and {John} at the third pop. As usual with stack-based algorithms, pop actions are needed when a new node n is to be pushed into a stack and the top node is not an ancestor of n. Thus, T-LCAsz pops top stack nodes until the parent of the new partial LCA appears in the top stack position (lines 26–27). If the stack is the sink stack of the lattice, the popped nodes are full LCAs and are added to the result set (procedure pop(), lines 4–5). Otherwise, the partial LCAs of each popped node are paired to build new (partial) LCAs (procedure pop(), lines 6–16). If a partial LCA size is below the threshold T, it is also propagated to the parent node incremented by one. In Fig. 5(a), the partial LCA 1.1.1.2.2 of John with size 0 is propagated to node 1.1.1.3.2 with size 1. Its provenance is also set to 1, which identifies the child 1.1.1.3.2.1 of 1.1.1.3.2 contributing the partial LCA size.

Before two partial LCAs are combined to produce a new LCA, two conditions are examined: (a) whether their total size exceeds T, and (b) whether their provenance indicator sets are disjoint (line 9). The latter reflects the requirement of Remark 2 (Section 3), and allows the construction of new partial LCAs only by combining disjoint partial LCA subtrees. The provenance indicator, which determines the disjointness of partial LCA subtrees, may be set in two ways. First, when a partial LCA size for some keyword 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 (procedure pop(), line 21). This Dewey step identifies the root of the partial LCA subtree that contributes this size. As an example, look at the first pop action of Fig. 5(a) and its effect on the provenance of the partial LCA of John. 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 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, John} produced at the third pop action in Fig. 5(a). This partial LCA is subsequently pushed into the stack {X, J, B} of Fig. 5(b).

4.2. T-LCAsz analysis

Algorithm T-LCAsz produces result LCAs to keyword queries by progressively combining keyword instances to partial to full LCAs. The processing of instances evolves along the directed paths of the lattice of stacks. Consider a keyword query of k keywords with threshold T on an XML tree of depth d. A keyword instance enters the lattice from the source stack (first coarseness level) and may reach the sink stack (last coarseness level), depending on size comparisons with comparable partial LCAs along the way. In the worst case, the instance will trigger a partial LCA push action to all the stacks of the lattice. The number of stacks is given by the Bell number of k, which is equal to the number of all possible partitions of a set with k elements and is defined by the following recursive formula:

\[ B_{n+1} = \sum_{i=0}^{n} \binom{n}{i} B_i, \quad B_0 = B_1 = 1 \]

Each partial LCA may at most need d pops of stack entries and d pushes of empty entries in the stack where it is to be pushed. The number of pushes and pops is also restricted by the threshold T. Each pop entails at most k(k − 1)/2 element combinations resulting in new partial LCAs and an equal number of comparisons with threshold T (O(k^2) time). For each pop, k updates of sizes in the parent stack entry are performed after k comparisons with threshold T (O(k) time). Thus, each pop requires O(k^2) time.

If \( |S_i| \) denotes the length of the ith keyword inverted list, the complexity of T-LCAsz is

\[
O\left( tk^2 B_k \sum_{i=1}^{k} |S_i| \right), \quad t = \min(d, T)
\]
Therefore, T-LCAsz is linear on the size of the inverted lists for a constant $k$ and $d$. In practice, T-LCAsz performs even better than this worst case scenario. Threshold $T$ limits the width and depth of keyword instance minimum connecting trees. This entails a reduction of the number of the (partial) LCAs produced and the corresponding stack updates. Lower thresholds affect T-LCAsz by filtering out partial LCAs early on, at finer coarseness levels of the lattice, further reducing the execution time.

### 4.3. Top-$k$-size and top-$k$ results

When a threshold size for LCAs is not known in advance, T-LCAsz cannot be exploited. In this section we introduce algorithms topKsz-LCAsz and topK-LCAsz which rank keyword search results in ascending size order and restrict the answer to the set of LCAs with top-$k$ sizes and to the set of top-$k$ LCAs, respectively.

According to the TLCA semantics, keyword query results of the same size have the same rank and therefore are equivalent with respect to their relevance to a keyword query. Algorithms topKsz-LCAsz and topK-LCAsz modify T-LCAsz so that it can function with an adjustable size threshold. Both algorithms initially set the size threshold to infinite and they pass through a learning phase during which they gradually reduce it, until it converges to the $K^{th}$ smallest size or to the size of the $K^{th}$ LCA, respectively. The earlier in the computation the final value of threshold $T$ is determined, the higher the pruning capacity of the algorithm and the less LCAs are computed.

Function addResult() of T-LCAsz adds a new LCA to the result set (function addResult(), lines 1–6). In the case of the top-$k$ algorithms, addResult() adjusts the size threshold $T$ of T-LCAsz (function adjustResultSet(), line 7) during their learning phase. The result set consists of buckets of results with the same size. Every new LCA is added to the bucket corresponding to its size (function addNode(), line 6). If this bucket does not exist, it is first constructed (function createNewSize(), line 5).

After the addition of a new result in the execution of topKsz-LCAsz, if the result set comprises exactly $K$ different size buckets, the size threshold $T$ is updated to the maximum size of the buckets (procedure getLastSize(), line 12). Otherwise, if the new LCA triggers the

---

**Fig. 5.** States of the stacks for $Q=\{XML, John, Brown\}$ with threshold $T=5$ when T-LCAsz processes XML instance.
construction of a new size bucket increasing the number of buckets to \(K+1\), the bucket of maximum size is discarded, and \(T\) is set to the size of the new maximum size.

**Function 1. addResult**

```
1 addResult(lcaDewey, lcaSize)
2   if results contain lcaSize then
3       sizeBucket = getsize(results, lcaSize)
4     else
5       sizeBucket = createNewSize(results, lcaSize)
6     addNode(sizeBucket, lcaDewey)
7     return adjustResultSet()\
8 adjupsetSet() /* topKsz-LCAsz */
9   if countSizes(results) > K then
10      removeMaxSize(results) /* remove max size bucket */
11    /* Check if K threshold is reached */
12   if countSizes(results) = K then
13      return getLastSize(results)
14     else
15     return \(\infty\)
16 adjupsetSet() /* topK-LCAsz */
17   if count(results) > K then
18      removeLast(results) /* remove an arbitrary dewey from the bucket with max size */
19    /* Check if K threshold is reached */
20   if count(results) = K then
21     return getLastSize(results)
22 else
23   return \(\infty\)
```

In the case of topK-LCAsz, the adjustment of threshold \(T\) starts when \(K\) results have been computed and the threshold \(T\) is set to the size of the bucket of maximum size (procedure getLastSize(), line 19). If a new LCA increases the total number of results to \(K+1\), an arbitrary LCA is removed from the bucket of the maximum size. If the bucket is emptied by this removal, it is discarded.

The body of T-LCAsz is also adapted to accommodate the functionality of topKsz-LCAsz and topK-LCAsz. Two additional checks (lines 11 and 15 of the algorithm T-LCAsz) guarantee that the threshold is not further reduced between the construction of an LCA and its processing at the subsequent coarseness levels.

**5. Experimental evaluation**

Our experiments demonstrate (i) the efficiency of T-LCAsz, topKsz-LCAsz and topK-LCAsz algorithms and (ii) the effectiveness of top-k-size TLCA ranking and filtering semantics.

The experiments were conducted on a computer equipped with a 1.8 GHz dual core Intel Core i5 with Mac OS Lion installed. The code was implemented in Java.

We used the real datasets DBLP1 and NASA2 and the benchmark auction dataset XMark.3 Table 1 provides statistics for them. After being parsed, the inverted lists of their keywords were stored in a MySQL database. XML elements and attributes were equally regarded as distinct tree nodes.

DBLP is much larger with greater variety of keywords but very shallow. The 97% of the keywords are located at nodes of depth 2 and these nodes amount for the 90% of the total number of nodes of the data tree. XMark and NASA are 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 relative to its tree size.

**5.1. Efficiency of the algorithms**

Tables 2–4 list the set of queries used in our efficiency experiments. For each query the tables show its keywords, the total number of keyword instances and the total number of their results (i.e., without imposing any size threshold). For every query, the distribution of results in different sizes is also presented. Table 5 shows the number of instances of the

---

1 http://www.informatik.uni-trier.de/ley/db/
2 http://www.cs.washington.edu/research/xmldatasets/www/repository.html
3 http://www.xml-benchmark.org
keywords and their type (that is, whether the keyword matches the label of the node or the value of a node) in the relevant data tree. Note, however, that our algorithms do not distinguish these two different types of instances. Different threshold and top-k-size queries with the same number of keywords can return answers of different cardinalities even if they have a similar number of keyword instances. For example, queries $Q_5^X$ and $Q_5^N$ both have 5 keywords and a similar number of keyword instances but the distribution of results into sizes is totally different.

5.1.1. Performance comparison

The performance of our proposed algorithms is compared against algorithm SA [15] and its variations. Algorithm SA is the only known approach in the literature, which is inherently capable to compute the sizes of keyword query results in XML data. It aims at identifying minimum connecting trees of a set of keywords with distances (DMCTs) and further merges homomorphic DMCTs into grouped distance MCTs (GDMCTs). It returns only GDMCTs not exceeding a given threshold size. We compare our algorithms with a variation of SA, algorithm SAOne [15], which computes only LCAs and does not need to merge homomorphic DMCTs. In our implementation of SAOne, we let the smallest among the DMCTs of an LCA to contribute its size to that LCA. SAOne is naturally comparable with T-LCAzs and has been proven to outperform the naïve approach that exhaustively computes all MCTs for a given query on a data tree [15]. For topKsz-
LCAsz and topK-LCAsz performance comparisons, we further extended SAOne to accept $K$ as input and (i) compute results for the top $K$ LCA sizes ($\text{topKsz-SAOne}$) and (ii) compute the top $K$ results based on their LCA sizes ($\text{topK-SAOne}$). Both implementations prune the result set early in the computation by preventing the generation of large size DMCTs.

Figs. 7–9 show performance results for three pairs of algorithms (i.e., T-LCAsz vs SAOne, topKsz-LCAsz vs topKsz-SAOne, and topK-LCAsz vs topK-SAOne). The $y$-axis displays the net processing time of the inverted lists of the query keywords after they are loaded in memory. The loading time is not taken into account as it affects equally all implemented algorithms. Note that scale of the $y$-axis is logarithmic. All algorithms are set to return the results of the top size for each query (Tables 2–4). For the threshold algorithms, threshold $T$ is set to the top size (e.g., $T=0$ for $Q_1^T$, $T=2$ for $Q_2^T$), for the top-$k$-size algorithms $K=1$ and for the top-$k$ algorithms $K$ is set to the number of top size results (e.g., $K=442$ for $Q_1^T$, $K=92$ for $Q_2^T$). As we can see, T-
LCAsz, topKsz-LCAsz and topK-LCAsz clearly outperform SAOne and its variations in all cases. The difference is not very significant on DBLP for queries with few keywords and a small number of instances. However, as the number of keywords and their instances increases the difference of performance among the two families of algorithms grows substantially. T-LCAsz, topKsz-LCAsz and topK-LCAsz are not affected by the input size (i.e., the length of the inverted lists), as shown by the computation time that proves to be below 10 s in most cases.

Top-k results algorithms (topK-LCAsz and topK-SAOne) generally need more time to answer keyword queries as their size threshold might not converge early in the computation to its final value, producing in the meantime many results that exceed this threshold. On the XMark and NASA datasets, topK-SAOne fails to compute in practice top-k results for queries with more than 3 and 4 keywords respectively. In contrast, topK-LCAsz efficiently copes with the threshold convergence and demonstrates a good performance.

---

**Fig. 10.** Performance of T-LCAsz varying the number of instances (retrieving results of size at most $T$).

**Fig. 11.** Performance of topKsz-LCAsz varying the number of instances (retrieving results of top-k sizes).

**Fig. 12.** Performance of topK-LCAsz varying the number of instances (retrieving top-k results).
5.1.2. Scaling

In order to run scaling experiments with input inverted lists of different sizes, we considered the inverted lists of the keywords in the queries of Tables 2–4 and truncated them. All the keywords of these queries are selected to have large inverted lists (large number of instances), which can be truncated at different lengths. Choosing keywords of high frequency stresses the algorithms, since many partial LCAs are produced during the processing. Every displayed measurement in the plots is averaged over 10 executions of the relevant query. The loading time of the inverted lists again was not measured.

Figs. 10–12 illustrate how T-LCAsz, topKsz-LCAsz and topK-LCAsz scale varying the length of the input inverted lists. The keyword inverted lists were truncated to contain from 1000 to 10,000 instances per keyword. For clarity, the plots of only two queries for each dataset are displayed but the results for the other queries are similar.

For each query there are two or three curves that correspond to different T or K values. For T-LCAsz, the threshold T is set to the top two sizes observed for that query. Analogously, for topKsz-LCAsz, K is set to 1 and 2 for top-1 and top-2 size results, respectively. For topK-LCAsz, the number of results is related to the number of keyword instances. For this reason, the three values of K were set to 1 (i.e., the top result) and to the two different values that roughly correspond to the number of results with top-1 and top-2 sizes of the query under consideration.

The queries in the scaling diagrams cover a broad spectrum of features: $Q_1^D$ and $Q_1^X$ result in just a few LCAs compared to the number of their keyword instances and almost all LCAs are of the top size. On the contrary, $Q_3^D$ and $Q_3^X$ have many results in comparison to the number of their instances. Most of the results spread across the top one size (for $Q_3^D$) or the top two sizes (for $Q_3^X$). Finally, the keywords in the queries on the NASA dataset are not strongly correlated as the number of results is small in comparison to their input size.

For a given number of keywords, the scaling diagrams show the linear behavior of the execution time of T-LCAsz, topKsz-LCAsz and topK-LCAsz with respect to the input size. This behavior confirms the complexity analysis presented in Section 4.2. The algorithms scale smoothly independently of the underlying dataset or the number of the query keywords.

5.1.3. Comparison of the three algorithms

Fig. 13 illustrates the relative performance of T-LCAsz, topKsz-LCAsz and topK-LCAsz, when computing the results of the minimum size. For instance, in the case of $Q_1^D$, T=2 for T-LCAsz, K=1 for topKsz-LCAsz and K=12 for topK-LCAsz. Queries $Q_1^D$, $Q_3^D$ and $Q_5^D$ were chosen for the comparison.

As a general observation, T-LCAsz performs faster than the other two. This is expected since in the case of T-LCAsz, the size threshold T is already known in advance and no learning phase is needed. Algorithms topKsz-LCAsz and topK-LCAsz may or may not differ significantly in their performance, depending on the particular characteristics of the dataset and the keyword query. In most cases, topKsz-LCAsz reaches the final threshold T much faster than topK-LCAsz. The reason is that topKsz-LCAsz needs only one result of the minimum size to determine T, while topK-LCAsz needs K results of the minimum size for converging to the same threshold. For example, when processing query $Q_1^D$, topKsz-LCAsz (with K=1) finds the minimum size 2 after processing only the 49th keyword instance out of 90,627, whereas topK-LCAsz (with K=8010) completes the computation of the 8010 minimum size results after having processed all instances (i.e., at the 90,627th iteration). As shown in the bar chart of Fig. 13, in this case, topKsz-LCAsz performs almost as fast as T-LCAsz and topK-LCAsz lags behind.

5.2. Effectiveness of TLCA semantics

We experimentally studied the effectiveness of both top-k-size TLCA filtering and ranking semantics. In these experiments, we used DBLP and NASA which are real datasets but demonstrate different characteristics (Table 1). Table 6 shows the queries issued against each dataset. Algorithm topKsz-LCAsz was set to compute top-2-size LCAs. The LCAs are returned grouped in two layers: the layer of the minimum size (top-1-size) LCAs and the layer of the subsequent size LCAs. The binary relevance (correctness) and graded relevance assessments of all the LCAs were provided by expert users. For the graded relevance, a 4-value scale was used with 0 denoting irrelevance. In order to avoid the manual assessment of each LCA in the XML tree, which is unfeasible because the LCAs are usually numerous, we used the tree patterns that the query instances of these LCAs define in the XML tree. These patterns show how the query keyword instances are combined under an LCA to form an MCT, and how the LCA is connected to the root of the data tree. Since they are bound by the schema of a dataset they are in practice much less numerous than the LCAs. The relevance of an LCA is the maximum relevance of the patterns with which the query instances of the LCA comply.

5.2.1. Effectiveness of top-k-size TLCA layered filtering semantics

To evaluate the filtering aspect of the TLCA semantics, we used the top-1-size layer to achieve improved precision and top-2-size layers to achieve improved recall. The TLCA semantics is compared with the SLCA and ELCA semantics. Similar to TLCA, SLCA and ELCA use only structural information for selecting the relevant results. Other approaches are based on determining complex semantic relationships among the nodes in the XML tree, which requires the construction of auxiliary data.
structures and expensive preprocessing of the data \cite{10,18,22}. The comparison was based on the broadly used precision ($P$), recall ($R$) and $\mathcal{F}$—measure $= 2P \times R / (P + R)$ metrics \cite{3}. The experimental results are shown in Fig. 14.

The top-1-size TLCA result set demonstrates perfect precision for all queries on both datasets. The precision of the other approaches varies in most cases below 50%. All approaches show high recall on DBLP, since its shallowness does not leave room for generation of false negatives. On the NASA dataset, though, top-2-size shows better recall than all the other approaches for all queries. The table in Fig. 14(c) shows the average $\mathcal{F}$—measure values over all issued queries for each dataset. $\mathcal{F}$—measure combines precision and recall figures into a single number and, as one can see, top-1-size and top-2-size TLCA show much better results on $\mathcal{F}$—measure than SLCA and ELCA.

5.2.2. Effectiveness of top-k-size TLCA ranking semantics

We evaluated the ranking of TLCA semantics by computing the Mean Average Precision (MAP) \cite{3} and the Normalized Discounted Cumulative Gain (NDCG) \cite{3} on the queries of Table 6. MAP is the average of the individual precision values that are obtained every time a correct result is observed in the ranking of the query. If a correct result is never retrieved, its contributing precision value is 0. MAP penalizes an algorithm when correct results are missed or incorrect ones are ranked highly. Given a specific position in the ranking of a query result set, the Discounted Cumulative Gain (DCG) is defined as the sum of the grades of the query results until this ranking position, divided (discounted) by the logarithm of that position. The DCG vector of a query is the vector of the DCG values at the different ranking positions of the query result set. Then, the NDCG vector is the result of normalizing this vector with the vector of the perfect ranking (i.e., the one obtained by the grading of the result set by the experts). NDCG penalizes an algorithm when the latter favors poorly graded results over good ones in the ranking.

Table 7 shows the MAP and NDCG values of top-1-size and top-2-size approaches on the queries of Table 6. Because TLCA does not specify an order for results of the same size, we computed MAP and NDCG values for two rankings of the results which respectively rank LCAs of the same size in descending and ascending order of their grades. Each value displayed in Table 7 is the average of these two bounds which correspond to the best and worst value for the corresponding metric.

The perfect NDCG values in both datasets confirm our intuition about TLCA semantics, as they indicate that top-1-size results, also get the highest grades. The perfect values of NDCG for top-2-size TLCA show that the ranking in ascending order of LCA size complies with the perfect ranking of the results. Most MAP values are slightly inferior to 100. This is due to the fact that a small number of correct LCAs is rejected in some cases. The high NDCG values, though, guarantee that the excluded LCAs are not highly graded.

Table 6

<table>
<thead>
<tr>
<th>DBLP</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q_D</td>
<td>neural networks algorithms</td>
<td></td>
</tr>
<tr>
<td>Q_D</td>
<td>author Yi Chen</td>
<td></td>
</tr>
<tr>
<td>Q_D</td>
<td>Yi Chen XML</td>
<td></td>
</tr>
<tr>
<td>Q_D</td>
<td>editor Vijay year 2012</td>
<td></td>
</tr>
<tr>
<td>Q_D</td>
<td>best paper award year 2012</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NASA</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q_N</td>
<td>observed stars classification</td>
<td></td>
</tr>
<tr>
<td>Q_N</td>
<td>sun astrolabe santiago</td>
<td></td>
</tr>
<tr>
<td>Q_N</td>
<td>photometric measurements</td>
<td></td>
</tr>
<tr>
<td>Q_N</td>
<td>ccd photometric system magnitudes</td>
<td></td>
</tr>
<tr>
<td>Q_N</td>
<td>galaxies clusters optical observations</td>
<td></td>
</tr>
</tbody>
</table>

Table 7

<table>
<thead>
<tr>
<th>DBLP</th>
<th>MAP (%)</th>
<th>NDCG (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top1sz</td>
<td>91</td>
<td>100</td>
</tr>
<tr>
<td>top2sz</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NASA</th>
<th>MAP (%)</th>
<th>NDCG (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top1sz</td>
<td>89.8</td>
<td>100</td>
</tr>
<tr>
<td>top2sz</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>
6. Conclusion

In this paper, we have addressed the problem of computing top-k-size LCAs as answers of keyword queries on tree-structured data. The concept of LCA size is used to introduce top-k-size ranking and filtering semantics for keyword queries, which is called Tight LCA semantics. TLCA semantics does not compromise quality by ranking a predefined subset of the LCAs as previous ranking and top-k approaches do. Despite its conceptual simplicity, it demonstrates improved quality characteristics. Through a layered presentation of results it addresses user interface problems related with top-k approaches.

We have devised novel multi-stack top-k algorithms, which organize the stacks in a lattice. In contrast to previous approaches, they do not require a preprocessing of the datasets for the creation of auxiliary index structures and therefore they are also appropriate for streaming applications. Our experimental results confirm the theoretical analysis and show that our algorithms exhibit linear performance with respect to the input size for a given query length. Therefore, they can scale smoothly when the size of the dataset increases.

Our future work aims at designing new algorithms built on the lattice of keyword partitions which further improve performance by achieving dimensionality reduction of the lattice. Towards this direction we plan to adopt more expressive keyword query language that will also improve the effectiveness of our answering mechanisms. We also plan to examine how our approach can be combined with keyword statistics based approaches to further refine our result ranking.

References