Cohesive Keyword Search on Tree Data

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

Ananya Dass
Department of Computer Science
New Jersey Institute of Technology, USA
ad292@njit.edu

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

Yannis Vassiliou
School of Electrical and Computer Engineering
National Technical University of Athens, Greece
yv@cs.ntua.gr

ABSTRACT

Keyword search is the most popular querying technique on semistructured data. Keyword queries are simple and convenient. However, as a consequence of their imprecision, there is usually a huge number of candidate results of which only very few match the user's intent. Unfortunately, the existing semantics for keyword queries are ad-hoc and they generally fail to "guess" the user intent. Therefore, the quality of their answers is poor and the existing algorithms do not scale satisfactorily.

In this paper, we introduce the novel concept of cohesive keyword queries for tree data. Intuitively, a cohesiveness relationship on keywords indicates that they should form a cohesive whole in a query result. Cohesive keyword queries allow term nesting and keyword repetition. Cohesive keyword queries bridge the gap between flat keyword queries and structured queries. Although more expressive, they are as simple as flat keyword queries and not require any schema knowledge. We provide formal semantics for cohesive keyword queries and rank query results on the proximity of the keyword instances. We design a stack based algorithm which efficiently evaluates cohesive keyword queries. Our experiments demonstrate that our approach outperforms in quality previous filtering semantics and our algorithm scales smoothly on queries of even 20 keywords on large datasets.

1. INTRODUCTION

Keyword search has been by far the most popular technique for retrieving data on the web. The success of keyword search relies on two facts: First, the user does not need to master a complex structured query language (e.g., SQL, XQuery, SPARQL). This is particularly important since the vast majority of people who retrieve information from the web do not have expertise in databases. Second, the user does not need to have knowledge of the schema of the data sources over which the query is evaluated. In practice, in the web, the user might not even be aware of the data sources which contribute results to her query. The same keyword query can be used to extract data from multiple data sources which have different schemas and they possibly adopt different data models (e.g., relational, tree, graph, flat documents).

There is a price to pay for the simplicity, convenience and flexibility of keyword search. Keyword queries are imprecise in specifying the query answer. They lack expressive power compared to structured query languages. As a consequence, there is usually a very large number of candidate results from which very few are relevant to the user intent. This weakness incurs at least two major problems: (a) because of the large number of candidate results, previous algorithms for keyword search are of high complexity and they cannot scale satisfactorily when the number of keywords and the size of the input dataset increase, and (b) correctly identifying the relevant results among the plethora of candidate results, becomes a very difficult task. Indeed, it is practically impossible for a search system to "guess" the user intent from a keyword query and the structure of the data source.

The focus of this work is on keyword search over web data which are represented as trees. Currently, huge amounts of data are exported and exchanged in tree-structure form [31, 33]. Trees can conveniently represent data that are semistructured, incomplete and irregular as is usually the case with data on the web. Multiple approaches assign semantics to keyword queries on data trees by exploiting structural and semantic features of the data in order to automatically filter out irrelevant results. Examples include Smallest LCA [18, 40, 37, 10], Exclusive LCA [16, 41, 42], Valuable LCA [11, 20], Meaningful LCA [24, 38], MaxMatch [28], Compact Valuable LCA [20] and Schema level SLCA [15]. A survey of some of these approaches can be found in [29]. 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. A better technique is adopted by other approaches which rank the candidate results in descending order of their estimated relevance. Given that users are typically interested in a small number of query results, some of these approaches combine ranking with top-k algorithms for keyword search. The ranking is performed using a scoring function and is usually based on IR-style metrics for flat documents (e.g., tf*idf or PageRank) adapted to the tree-structure form of the data [16, 11, 3, 5, 21, 10, 38, 23, 29, 32]. Nevertheless, scoring functions, keyword occurrence statistics and probabilistic models alone cannot capture effectively the intent of the user. As a consequence, the produced rankings are, in general, of low quality [38].

Our approach. In this paper, we introduce a novel approach which allows the user to specify cohesiveness relationships among keywords in a query, an option not offered by the current keyword query languages. Cohesiveness relationships group together keywords in a query to define indivisible (cohesive) collections of keywords. They partially relieve the system from guessing without affecting the user who can naturally and effortlessly specify such relationships.

For example, consider the keyword query `{XML John Smith George Brown}` to be issued against a large bibliographic database. The user is looking for publications on `XML` related to the authors `John Smith` and `George Brown`. If the user can express the fact that the instances of `John` and `Smith` should form a unit where the instances of the other keywords of the query `George`, `Brown` and `XML` cannot slip into to separate them (that is, the instances of `John` and `Smith` form a *cohesive whole*), the system would be able to return more accurate results. For example, it will be able to filter out publications on `XML` by `John Brown` and `George Smith`. It will also filter out a publication which cites a paper authored by `John` Davis, a report authored by `George Brown` and a book on `XML` authored by Tom `Smith`. This information is irrelevant and no one of the previous filtering approaches (e.g., ELCA, VLCA, CVLCA, SLCA, MLCA, MaxMach etc.) is able to automatically exclude it from the answer of the query. Furthermore, specifying cohesiveness relationships prevents wasting time searching for these irrelevant results. We specify cohesiveness relationships among keywords by enclosing them between parentheses. For example, the previous keyword query would be expressed as `(XML (John Smith) (George Brown))`.

Note that the importance of bridging the gap between flat keyword queries and structured queries in order to improve the accuracy (and possibly the performance) of keyword search has been recognized before for flat text documents. Google allows a user to specify, between quotes, a phrase which has to be matched intact against the text document. However, specifying a cohesiveness relationship on a set of keywords is different than phrase matching over flat text documents in IR. Indeed, a cohesiveness relationship on a number of keywords over a data tree does not impose any syntactic restriction (e.g., the order of the keywords) on the instances of these keywords on the data tree. It only requires that the instances of these keywords form a cohesive unit. We provide formal semantics for queries with cohesiveness relationships on tree data in Section 2.

Cohesiveness relationships can be nested. For instance the query `(XML (John Smith) (citation (George Brown)))` looks for a paper on `XML` by `John Smith` which cites a paper by `George Brown`. The cohesive keyword query language conveniently allows also for keyword repetition. For instance, the query `(XML (John Smith) (citation (John Brown)))` looks for a paper on `XML` by `John Smith` which cites a paper by `John Brown`.

Cohesive queries better express the user intent while being as simple as flat keyword queries. However, despite their increased expressive power, they enjoy both advantages of traditional keyword search: they do not require any previous knowledge of a query language or of the schema of the data sources. The users can naturally and effortlessly specify cohesiveness relationships when writing a query. The benefits, though, in query answer quality and performance compared to other flat keyword search approaches are impressive.

Contribution. The main contributions of our paper are as follows:

- We formally introduce a novel keyword query language which allows for cohesiveness relationships, cohesiveness relationship nesting and keyword repetition. Our semantics interpret the subtrees rooted at the LCA of the instances of cohesively related keywords in the data tree as impenetrable units where the instances of the other keywords cannot slip in.
- We rank the results of cohesive keyword queries on tree data based on the concept of LCA size. The LCA size reflects the proximity of keywords in the data tree and, similarly to keyword proximity in IR, it is used to determine the relevance of the query results.
- We design an efficient multi-stack-based algorithm which exploits a lattice of stacks—each stack corresponding to a different partition of the query keywords. Our algorithm does not rely on auxiliary index structures and, therefore, can be exploited on datasets which have not been preprocessed.
- We show how cohesive relationships can be leveraged to lower the dimensionality of the lattice and dramatically reduce its size and improve the performance of the algorithm.
- We analyze our algorithm and show that for a constant number of keywords it is linear on the size of the input keywords' inverted lists, i.e., to the dataset size. Our analysis further shows that the performance of our algorithm essentially depends on the maximum cardinality of the largest cohesive term in the keyword query.
- We run extensive experiments on different real and benchmark datasets to assess the effectiveness of our approach and the efficiency and scalability of our algorithm. Our results show that our approach largely outperforms previous filtering approaches, which do not benefit from cohesiveness relationships, achieving in most cases perfect precision and recall. They also show that our algorithm scales smoothly when the number of keywords and the size of the dataset increase achieving interactive response times even with queries of 20 keywords having in total several thousands of instances on large datasets.

2. DATA AND QUERY MODEL

We consider data modeled as an ordered labeled tree. Every node has an id, a label and possibly a value. For identifying tree nodes we adopt the Dewey encoding scheme [9], which encodes tree nodes according to a preorder traversal of the data tree. The Dewey encoding scheme naturally expresses ancestor-descendant and parent-child relationships among tree nodes and conveniently supports the processing of nodes in stacks [16].

A keyword k may appear in the label or in the value of a node n in the data tree one or multiple times, in which case we say that node n constitutes an *instance* of k. A node may contain multiple distinct keywords in its value and label, in which case it is an instance of multiple keywords.

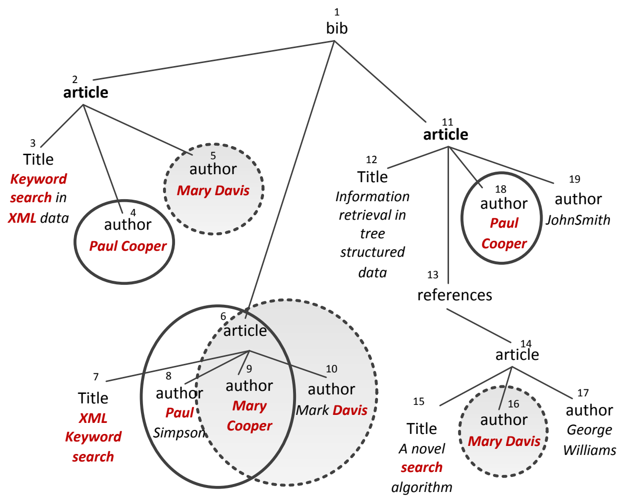


Figure 1: Example data tree D_1

2.1 Cohesive semantics

A *cohesive keyword query* is a keyword query, which besides keywords may also contain groups of keywords called terms. Intuitively, a term expresses a cohesiveness relationship on keywords and/or terms. More formally, a cohesive keyword query is recursively defined as follows:

Definition 1 (Cohesive keyword query). *A* term *is a multiset of at least two keywords and/or terms. A* cohesive keyword query *is: (a) a set of a single keyword, or (b) a term. Sets and multisets are delimited within a query using parentheses.*

For instance, the expression `((title XML) ((John Smith) author))` is a keyword query. Some of its terms are T_1 = `(title XML)`, T_2 = `((John Smith) author)`, T_3 = `(John Smith)`, and T_3 is *nested* into T_2.

A keyword may occur multiple times in a query. For example, in the keyword query `((journal (Information Systems) ((Information Retrieval) Smith))` the keyword `Information` occurs twice, once in the term `(Information Systems)` and once in the term `(Information Retrieval)`.

In the rest of the paper, we may refer to a cohesive keyword query simply as query. The syntax of a query Q is defined by the following grammar where the non-terminal symbol T denotes a term, and the terminal symbol k denotes a keyword:

$$\begin{array}{lll} Q & \rightarrow & (k) \mid T \\ T & \rightarrow & (S\ S) \\ S & \rightarrow & S\ S \mid T \mid k \end{array}$$

We now move to define the semantics of cohesive keyword queries. Keyword queries are embedded into data trees. In order to define query answers, we need to introduce the concept of query embedding. In cohesive keyword queries, m occurrences of the same keyword in a query are embedded to one or multiple instances of this keyword as long as these instances collectively contain at least m times this keyword. Cohesive keyword queries may also contain terms, which, as mentioned before, express a cohesiveness relationship on their keyword occurrences. In tree structured data, the keyword instances in the data tree (which are nodes) are represented by their LCA [36, 16, 29]. The instances of the keywords of a term in the data tree should form a cohesive unit. That is, the subtree rooted at the LCA of the instances of the keywords which are internal to the term should be impenetrable by the instances of the keywords which are external to the term. Therefore, if l is the LCA of a set of instances of the keywords in a term T, i is one of these instances and i' is an instance of a keyword not in T, then $lca(i', i) = lca(i', l) \neq l$.

As an example, consider query Q_1 =`(XML keyword search (Paul Cooper) (Mary Davis))` issued against the data tree D_1 of Figure 1. In Figure 1, keyword instances are shown in bold and the instances of the keywords of a term below the same article are encircled. The mapping that assigns `Paul` to node 8, `Mary` and `Cooper` to node 9 and `Davis` to node 10 is not an embedding of Q_1 since `Mary` slips into the encircled subtree of `Paul` and `Cooper` rooted at article node 6: the two circles of article node 6 overlap. These ideas are formalized below.

Definition 2 (Query embedding). *Let Q be a keyword query on a data tree D. An embedding of Q to D is a function e from every keyword occurrence in Q to an instance of this keyword in D such that:*

a. *if $k_1, \ldots, k_m$ are distinct occurrences of the same keyword k in Q and $e(k_1) = \ldots = e(k_m) = n$, then node n contains keyword k at least m times.*

b. *if $k_1, \ldots, k_n$ are the keyword occurrences of a term T, k is a keyword occurrence not in T, and $l = lca(e(k_1), \ldots, e(k_n))$ then: (i) $e(k_1) = \ldots = e(k_n)$, or (ii) $lca(e(k), l) \neq l$.*

Given an embedding e of a query Q involving the keyword occurrences $k_1, \ldots, k_m$ on a data tree D, the *minimum connecting tree* (MCT) M of e on D is the minimum subtree of D that contains the nodes $e(k_1), \ldots, e(k_m)$. Tree M is also called an MCT of query Q on D. The root of M is the *lowest common ancestor* (LCA) of $e(k_1), \ldots, e(k_m)$ and defines one *result* of Q on D. For instance, one can see that the article nodes 2 and 11 are results of the example query Q_1 on the example tree D_1. In contrast, the article node 6 is not a result of Q_1.

We use the concept of LCA size to rank the results in a query answer. Similarly to metrics for flat documents in information retrieval, LCA size reflects the proximity of keyword instances in the data tree. The *size* of an MCT is the number of its edges. Multiple MCTs of Q on D with different sizes might be rooted at the same LCA node l. The size of l (denoted $size(l)$) is the minimum size of the MCTs rooted at l.

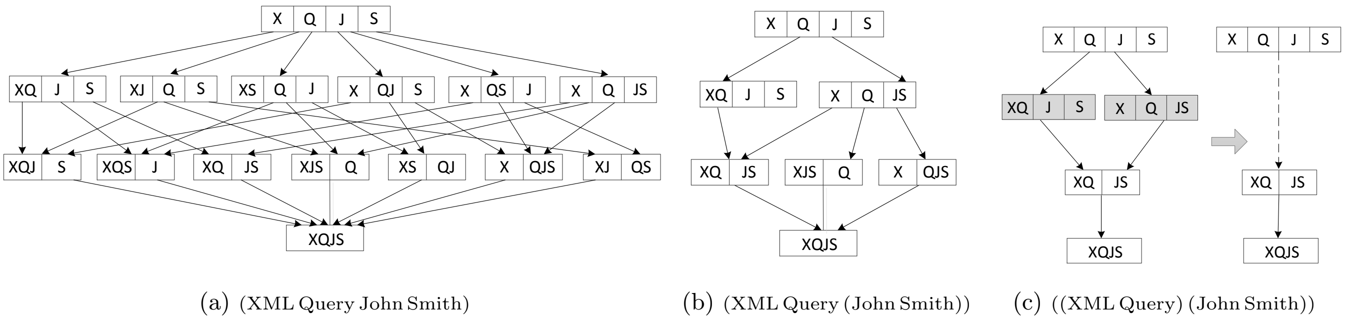
For instance, the size of the result article node 2 of query Q_1 on the data tree D_1 is 3 while that of the result article node 11 is 6 (note that there are multiple MCTs of different sizes rooted at node 11 in D_1).

Definition 3. *The answer to a cohesive keyword query Q on a data tree D is a list $[l_1, \ldots, l_n]$ of the LCAs of Q on D such that $size(l_i) \leq size(l_j), i < j$.*

For example, article node 2 is ranked above article node 11 in the answer of Q_1 on D_1.

2.2 Result ranking using cohesive terms

The LCA size naturally reflects the overall proximity of the query keyword instances in the subtree of a result LCA. Every result LCA contains partial LCAs corresponding to



X Q J S
XQ J S
XJ Q S
XS Q J
X QJ S
X QS J
X Q JS
XQJ S
XQS J
XQ JS
XJS Q
XS QJ
X QJS
XJ QS
XQJS
(a) (XML Query John Smith)
(b) (XML Query (John Smith))
(c) ((XML Query) (John Smith))

Figure 2: Lattices of keyword partitions for the query (`XML Query John Smith`) with different cohesiveness relationships

the nested cohesive terms in the query. These partial LCAs contribute with their size to the overall size of the LCA. However, we would like to take also into account how compactly the keyword instances are combined to form partial LCAs for each one of the nested cohesive terms. Consider, for instance, Figure 1 and the query Q_1 =`(XML keyword search (Paul Cooper) (Mary Davis))`. Article node 2 is an LCA for Q_1 and author node 4 is a partial LCA for the cohesive term (`Paul Cooper`) contributing with LCA size 0 to the total size of the LCA node 2. The fact that the keyword instances of `Paul` and `Cooper` are compactly connected to form a partial LCA of size 0 is also important, as is that the total size of the result is small.

To reflect the importance of intra-cohesive-term proximity, we propose a new ranking scheme which takes also into account the cohesive terms and their sizes in a query result. Our ranking method does not require the preprocessing of the dataset and does not constrain the result representation to specific index terms [4].

We represent each query result as a vector in the cohesive term space for a given query Q. Let Q be a query with m cohesive terms, including the outermost term, i.e., the query itself. Each LCA l_j of a data tree D with respect to Q is represented by the following vector:

$$\overrightarrow{l_j} = (C_1 s_{1,j}, C_2 s_{2,j}, \ldots, C_m s_{m,j})$$

where C_i is the weight of the term T_i in the query Q with respect to the dataset D and $s_{i,j}$ is the size of the the partial LCA for the term T_i within the LCA l_j. Intuitively, the parameter C_i reflects the compactness of the term T_i in the dataset D. That is, how closely the instances of the keywords in T_i appear in the partial LCAs for T_i in the dataset D. Let P_i be the set of the LCAs of a term T_i in D. Then, C_i is defined as follows:

$$C_i = \frac{|P_i|}{1 + \sum_{p \in P_i} size(p)}$$

The smaller the average size of the LCAs of a term in a dataset D the more compact the term is in D. The vector $\overrightarrow{l_j}$ of an LCA l_j is used to define the score of l_j:

$$score(l_j) = |\overrightarrow{l_j}|$$

The query results are ranked in ascending order of their *score*. The weight C_i rewards results which demonstrate small sizes for non-compact terms and penalizes results that demonstrate large sizes for terms, which are expected to be compact.

3. THE ALGORITHM

We designed algorithm CohesiveLCA for keyword queries with cohesiveness relationships. Algorithm CohesiveLCA computes the results of a cohesive keyword query and ranks them in descending order of their LCA size. The idea behind CohesiveLCA is that LCAs of keyword instances in a data tree can result from combining LCAs of subsets of these instances (i.e., partial LCAs of the query) in a bottom-up way in the data tree. CohesiveLCA progressively combines partial LCAs to eventually return full LCAs of instances of all query keywords higher in the data tree. During this process, LCAs are grouped based on the keywords contained in their subtrees. The members of these groups are compared among each other in terms of their size. CohesiveLCA exploits a lattice of partitions of the query keywords.

The lattice of keyword partitions. During the execution of CohesiveLCA, multiple stacks are used. Every stack corresponds to a partition of the keyword set of the query. Each stack entry contains one element (partial LCA) for every keyword subset belonging to the corresponding partition. Stack based algorithms for processing tree structured data push and pop stack entries during the computation according to a preorder traversal of the data tree. Dewey codes are exploited to index stack entries which at any point during the execution of the algorithm correspond to a node in the data tree. Consecutive stack entries correspond to nodes related with parent-child relationships in the data tree.

The stacks used by algorithm CohesiveLCA are naturally organized into a lattice, since the partitions of the keyword set (which correspond to stacks) form a lattice. Coarser partitions can be produced from finer ones by combining two of their members. Partitions with the same number of members belong to the same coarseness level of the lattice. Figure 2a shows the lattice for the keyword set of the query `(XML Query John Smith)`. CohesiveLCA combines partial LCAs following the source to sink paths in the lattice.

Reducing the dimensionality of the lattice. The lattice of keyword partitions for a given query consists of all possible partitions of query keywords. The partitions reflect all possible ways in which query keywords can be combined to form partial and full LCAs. Cohesiveness relationships restrict the ways keyword instances can be combined in a query embedding to form a query result. Keyword instances may be combined individually with other keyword instances to form partial or full LCAs only if they belong to the same term: if a keyword a is "hidden" from a keyword b inside a term T_a, then an instance of b can only be combined with an LCA of all the keyword instances of T_a and not individually with an instance of a. These restrictions result in significantly reducing the size of the lattice of the keyword partitions as exemplified next.

Figures 2b and 2c show the lattices of the keyword partitions of two queries. The queries comprise the same set of keywords `XML`, `Query`, `John` and `Smith` but involve different cohesive relationships. The lattice of Figure 2a is the full lattice of 15 keyword partitions and allows every possible combination of instances of the keywords `XML`, `Query`, `John` and `Smith`. The query of Figure 2b imposes a cohesiveness relationship on `John` and `Smith`. This modification renders several partitions of the full lattice of Figure 2a meaningless. For instance, in Figure 2b, the partition `[XJ, Q, S]` is eliminated, since an instance of `XML` cannot be combined with an instance of `John` unless the instance of `John` is already combined with an instance of `Smith`, as is the case in the partition `[XJS, Q]`. The cohesiveness relationship on `John` and `Smith` reduces the size of the initial lattice from 15 to 7. A second cohesiveness relationship between `XML` and `Query` further reduces the lattice to the size of 3, as shown in Figure 2c. Note that in this case, besides partitions that are not permitted because of the additional cohesiveness relationship (e.g., `[XJS, Q]`), some partitions may not be productive, which makes them useless. `[XQ, J, S]` is one such partition. The only valid combination of keyword instances that can be produced from this partition is `[XQ, JS]`, which is a partition that can be produced directly from the source partition `[X, Q, J, S]` of the lattice. The same holds also for the partition `[X, Q, JS]`. Thus, these two partitions can be eliminated from the lattice.

Algorithm description. Algorithm CohesiveLCA accepts as input a cohesive keyword query and the inverted lists of the query keywords and returns all LCAs which satisfy the cohesiveness relationships of the query, ranked on their LCA size.

The algorithm begins by building the lattice of stacks needed for the cohesive keyword query processing (line 2). This process will be explained in detail in the next paragraph. After the lattice is constructed, an iteration over the inverted lists (line 3) pushes all keyword instances into the lattice in Dewey code order starting from the source stack of the lattice, which is the only stack of coarseness level 0. For every new instance, a round of sequential processing of all coarseness levels is initiated (lines 6-9). At each step, entries are pushed and popped from the stacks of the current coarseness level. Each stack has multiple columns corresponding to and named by the keyword subsets of the relevant keyword partition. Each stack entry comprises a number of elements one for every column of the stack. The constructor PartialLCA produces a partial LCA element taking as parameters the Dewey code of a node, the term corresponding to a keyword partition, the size of the partial LCA and its provenance. Popped entries contain partial LCAs that are propagated to the next coarseness levels. An entry popped from the sink stack (located in the last coarseness level) contains a full LCA and constitutes a query result. After finishing the processing of all inverted lists, an additional pass over all coarseness levels empties the stacks producing the last results (line 10).

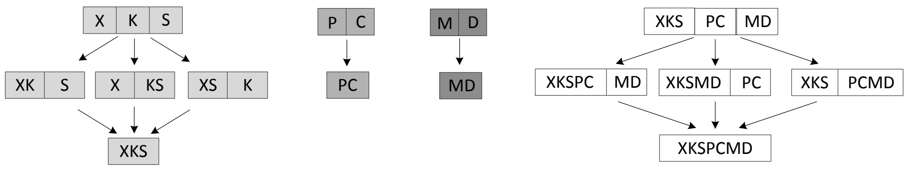
Procedure push() pushes an entry into a stack after ensuring that the top stack entry corresponds to the parent of the partial LCA to be pushed (lines 11-16). This process triggers pop actions of all entries that do not correspond to ancestors of the entry to be pushed. Procedure pop() is where partial and full LCAs are produced (lines 17-34). When an entry is popped, new LCAs are formed (lines 21-28) and the parent entry of the popped entry is updated to incorporate partial LCAs that come from the popped child entry (lines 29-34). The construction of new partial LCAs is performed by combining LCAs stored in the same entry.

Construction of the lattice. The key feature of CohesiveLCA is the dimensionality reduction of the lattice which is induced by the cohesiveness relationships of the input query. This reduction, as we also show in our experimental evaluation, has a significant impact on the efficiency of the algorithm. Algorithm CohesiveLCA does not naïvely prune the full lattice to produce a smaller one, but wisely constructs the lattice needed for the computation from smaller component sublattices. This is exemplified in Figure 3.

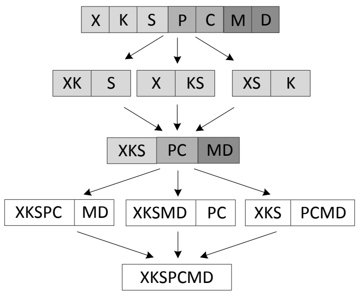
Consider the data tree depicted in Figure 1 and the query

Algorithm 1: CohesiveLCA

```
1  CohesiveLCA(Q: cohesive keyword query, invL: inverted lists)
2      buildLattice()
3      while currentNode ← getNextNodeFromInvertedLists() do
4          curPLCA ← PartialLCA(currentNode.dewey, currentNode.kw, 0, null)
5          push(initStack, curPLCA)
6          for every coarsenessLevel cL do
7              while pl ← next partial LCA of cL do
8                  for every stack S of cL containing pl.term do
9                      push(S, pl)
10     emptyStacks()
11 push(S: stack, pl: partial LCA)
12     while S.dewey not ancestor of pl.node do
13         pop(S)
14     while S.dewey ≠ pl.node do
15         addEmptyRow(S)
16     replaceIfSmallerWith(S.topRow, pl.term, pl.size)
17 pop(S: stack)
18     p ← S.pop()
19     if S.columns = 1 then
20         addResult(S.dewey, p[0].size)
21     if S.columns > 1 then
22         for i←0 to S.columns do
23             for j←i to S.columns do
24                 if p[i] and p[j] contain sizes and p[i].provenance ∩ p[j].provenance = ∅ then
25                     t ← findTerm(p[i].term, p[j].term)
26                     sz ← p[i].size+p[j].size
27                     prv ← p[i].provenance ∪ p[j].provenance
28                     pLCA ← PartialLCA(S.dewey, t, sz, prv)
29     if S is not empty and S.columns > 1 then
30         for i=0 to S.columns do
31             if p[i].size+1 < S.topRow[i].size then
32                 S.topRow[i].size ← p[i].size+1
33                 S.topRow[i].provenance ← {lastStep(S.dewey)}
34     removeLastDeweyStep(S.dewey)
```



(a) Component lattices for terms: (i)(`XML Keyword Search`), (ii)(`Paul Cooper`), (iii)(`Mary Davis`) and (iv)(`(XML Keyword Search)(Paul Cooper)(Mary Davis)`)



(b) Final lattice

Figure 3: Component and final lattices for the query `((XML Keyword Search) (Paul Cooper) (Mary Davis)))`

`((XML Keyword Search) (Paul Cooper) (Mary Davis)))` issued on this data tree. If each term is treated as a unit, a lattice of the partitions of three items is needed for the evaluation of the query. This is lattice (iv) of Figure 3a. However, the input of this lattice consists of combinations of keywords and not of single keywords. These three combinations of keywords each defines its own lattice shown in the left side of Figure 3a (lattices (i), (ii) and (iii)). The lattice to be finally used by the algorithm CohesiveLCA is produced by composing lattices (i), (ii) and (iii) with lattice (iv) and is shown in Figure 3b. This is a lattice of only 9 nodes, whereas the full lattice for 7 keywords has 877 nodes.

Function buildLattice() constructs the lattice for evaluating a cohesive keyword query. This function calls another function buildComponentLattice() (line 8). Function buildComponentLattice() (lines 18-21) is a recursive and builds all lattices for all terms which may be arbitrarily nested. The whole process is controlled by the *controlSets* variable which stores the keyword subsets admissible by the input cohesiveness relationships. This variable is constructed by the procedure constructControlSet() (lines 9-17).

Function buildLattice

```
1  buildLattice($Q$: query)
2      singletonTerms ← {keywords(Q)}
3      stacks.add(createSourceStack(singletonTerms))
       constructControlSet($Q$) for every control set cset in
       controlSets with not only singleton keywords do
4          stacks.add(createSourceStack(cset))
5      for every s in stacks do
6          buildComponentLattice($s$)
7  constructControlSet($qp$: query subpattern)
8      c ← new Set()
9      for every singleton keyword k in s do
10         c.add($k$)
11     for every subpattern sqp in s do
12         subpatternTerm ← constructControlSet(sqp)
13         c.add(subpatternTerm)
14     controlSets.add(c)
15     return newTerm(c)
16 buildComponentLattice($s$: stack)
17     for every pair t1, t2 of terms in s do
18         newS ← newStack(s, t1, t2)
           buildComponentLattice(newS)
```

3.1 Algorithm analysis

Algorithm CohesiveLCA processes the inverted lists of the keywords of a query exploiting the cohesiveness relationships to limit the size of the lattice of stacks used. The size of a lattice of a set of keywords with k keywords is given by the Bell number of k, B_k, which is defined by the recursive formula:

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

In a cohesive query containing t terms the number of sublattices is $t+1$ counting also the sublattice of the query (outer term). The size of the sublattice of a term with cardinality c_i is B_{c_i}. A keyword instance will trigger in the worst case an update to all the stacks of all the sublattices of the terms in which the keyword participates. If the maximum nesting depth of terms in the query is n and the maximum cardinality of a term or of the query itself is c, then an instance will trigger O(nB_c) stack updates. For a data tree with depth d, every processing of a partial LCA by a stack entails in the worst case d pops and d pushes, i.e., O(d). Every pop from a stack with c columns implies in the worst case $c(c-1)/2$ combinations to produce partial LCAs and c size updates to the parent node, i.e., O(c^2). Thus, the time complexity of CohesiveLCA is given by the formula:

$$O(dnc^2 B_c \sum_{i=1}^{c} |S_i|)$$

where S_i is the inverted list of the keyword i. The maximum term cardinality for a query with a given number of keywords depends on the number of query terms. It is achieved by the query when all the terms contain one keyword and one term with the exception of the innermost nested term which contains two keywords. Therefore, the maximum term cardinality is $k-t-1$ and the maximum nesting depth is t. Thus, the complexity of CohesiveLCA is:

$$O(dt(k-t-1)^2 B_{k-t-1} \sum_{i=1}^{k} |S_i|)$$

This is a paremeterized complexity which is linear to the size of the input (i.e., $\sum |S_i|$) for a constant number of keywords and terms.

	DBLP	XMark	NASA	PSD	Baseball
size	1.15 GB	116.5 MB	25.1 MB	683 MB	1.1 MB
maximum depth	5	11	7	6	5
# nodes	34,141,216	2,048,193	530,528	22,596,465	26,432
# keywords	3,403,570	140,425	69,481	2,886,921	1984
# distinct labels	44	77	68	70	46
# dist. label paths	196	548	110	97	46

Table 1: DBLP, XMark, NASA, PSD and Baseball dataset statistics

4. EXPERIMENTAL EVALUATION

We implemented our algorithm and we experimentally studied: (a) the effectiveness of the CohesiveLCA semantics and (b) the efficiency of the CohesiveLCA algorithm.

The experiments were conducted on a computer with a 1.6GHz dual core Intel Core i5 processor running Mac OS 10.8. The code was implemented in Java.

4.1 Datasets and queries

We used four real datasets: the bibliographic dataset DBLP[1], the astronomical dataset NASA[2], the Protein Sequence Database (PSD) [3] and the sports statistics dataset Baseball [4]. We also used a synthetic dataset, the benchmark auction dataset XMark[5]. These datasets cover different application areas and display various characteristics. Table 1 shows their statistics.

The DBLP is the largest and XMark the deepest dataset. For the effectiveness experiments, we used the real datasets DBLP, PSD, NASA and Baseball. For the efficiency evaluation, we used the DBLP, XMark and NASA datasets in order to test our algorithm on data with different structural and size characteristics. The keyword inverted lists of the parsed datasets were stored in a MySQL database.

We selected five cohesive keyword queries for each one of the four real datasets with an intuitive meaning. The queries display various cohesiveness patterns and involve 3-6 keywords. They are listed in Table 2. The binary relevance (correctness) and graded relevance assessments of all the LCAs were provided by five 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 data set 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.

4.2 Effectiveness of cohesive semantics

In this section we evaluate the effectiveness of cohesive semantics both as a filtering and as a ranking mechanism.

Filtering cohesive semantics. We compared the CohesiveLCA semantics with the *smallest* LCA (SLCA) [18, 40, 37, 10], *exclusive* LCA (ELCA) [16, 41, 42], *valuable* LCA (VLCA) [11, 20] and *meaningful* LCA [24] (MLCAs) filtering semantics. These are the best known filtering semantics discussed in the literature. An LCA is an SLCA if it is not an ancestor of another LCA in the data tree. An ELCA is an LCA of a set of keyword instances which are not in the subtree of any descendant LCA. An LCA is a VLCA if it is the root of an MCT which does not contain any label twice except when it is the label of two leaf nodes of the MCT. The MLCA semantics requires that for any two nodes n_a and n_b labeled by a and b, respectively, in an MCT, no node n'_b labeled by b exists which is more closely related to n_a (i.e., $lca(n_a, n'_b)$ is descendant of $lca(n_a, n_b)$). SLCA and ELCA semantics are based purely on structural characteristics, while VLCA and MLCA take also into account the labels of the nodes in the data tree.

	DBLP
Q_1^D	(proof (Scott theorem))
Q_2^D	((IEEE transactions communications) (wireless networks))
Q_3^D	((Lei Chen) (Yi Guo))
Q_4^D	((Wei Wang) (Yi Chen))
Q_5^D	((VLDB journal) (spatial databases))
	PSD
Q_1^P	((african snail) mRNA)
Q_2^P	((alpha 1) (isoform 3))
Q_3^P	((penton protein) (human adenovirus 5))
Q_4^P	(((B cell) stimulating factor) (house mouse))
Q_5^P	((spectrin gene) (alpha 1))
	NASA
Q_1^N	((ccd photometric system) magnitudes)
Q_2^N	((stars types) (spectral classification))
Q_3^N	((Astronomical (Data Center)) (Wilson luminosity codes))
Q_4^N	((year 1968) (Zwicky Abell clusters))
Q_5^N	((title Orion Nebula) (author Parenago))
	Baseball
Q_1^B	(Matt Williams (third base))
Q_2^B	(team (Johnson (first base)) (Wilson pitcher))
Q_3^B	(player surname (0 errors))
Q_4^B	(player (relief pitcher) (0 losses))
Q_5^B	(player (0 errors) (7 games))

Table 2: Queries for the effectiveness experiments on various datasets

Table 3 displays the number of results for each query and approach on the DBLP, PSD, NASA and Baseball datasets. Notice that, with the exception of SLCA and ELCA which satisfy a containment relationship (SLCA $\subseteq$ ELCA), all other approaches are pairwise incomparable. That is, one might return results that the other excludes and vice versa. The CohesiveLCA approach returns all the results that satisfy the cohesiveness relationships in the query. Since these relationships are imposed by the user, any additional result returned by another approach is irrelevant. For instance, for query Q_5^P, only 3 results satisfy the cohesiveness relationships of the user, and therefore, SLCA, VLCA, MLCA return at least 37 and ELCA at least 40 irrelevant results.

The CohesiveLCA semantics is a ranking semantics and ranks the results in layers based on their size. In order to

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

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

[3]http://pir.georgetown.edu/

[4]http://ibiblio.org/xml/books/biblegold/examples/baseball/

[5]http://www.xml-benchmark.org

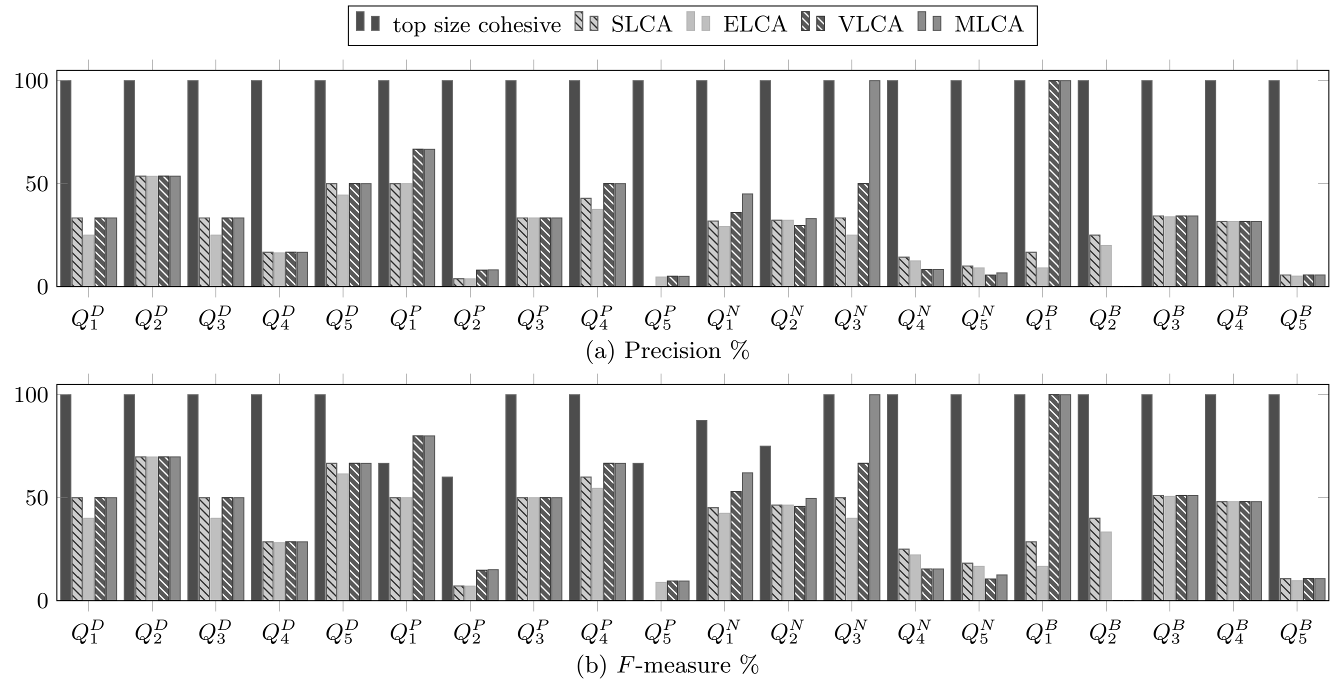


Figure 4: Precision and $\mathcal{F}$-measure of top size Cohesive LCA, SLCA and ELCA filtering semantics

compare its effectiveness with filtering semantics we restrict the results to the top layer (top-1-size results). Recall that these are the results with the minimum LCA size. The comparison is based on the widely used *precision (P)*, *recall (R)* and $\mathcal{F}$-measure$=\frac{2P\times R}{P+R}$ metrics [4]. Figure 4 depicts the results for the five semantics on the four datasets. Since all approaches demonstrate high recall, we only show the precision and $\mathcal{F}$-measure results in the interest of space.

The diagram of Figure 4a shows that CohesiveLCA largely outperforms the other approaches in all cases. Top-1-size CohesiveLCA shows perfect precision for all queries on all datasets. This is not surprising since, CohesiveLCA can benefit from cohesiveness relationships specified by the user to exclude irrelevant results. CohesiveLCA also shows perfect $\mathcal{F}$-measure on the DBLP and Baseball datasets. Its $\mathcal{F}$-measure on the PSD and NASA datasets is lower. This is due to the following reason: contrary to the shallow DBLP and Baseball datasets, the PSD and NASA datasets are deep and complex with a large amount of text in their nodes. This complexity leads to results of various sizes for most of the queries. Some of the relevant results are not of minimum size and they are missed by top-1-size CohesiveLCA. Nevertheless, any relevant result missed by top-1-size CohesiveLCA is retrieved by CohesiveLCA which returns all relevant results, as one can see in Table 4.

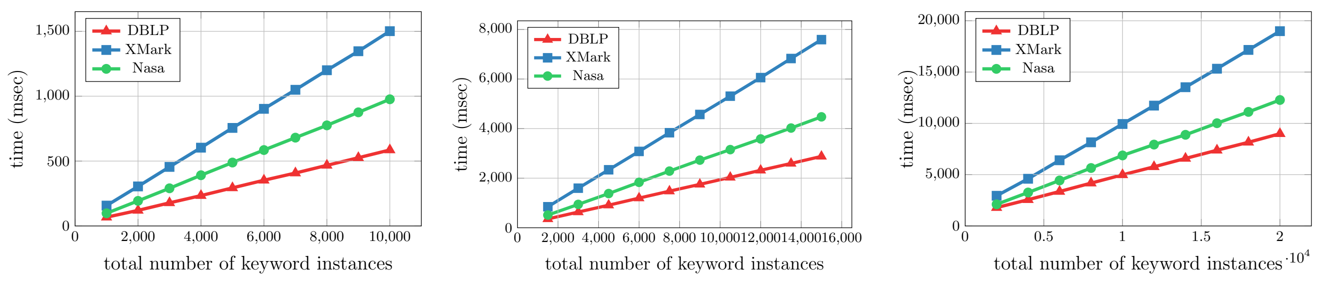
Table 4 summarizes the precision, recall and $\mathcal{F}$-measure values of the queries on all four datasets. The table displays values for the five filtering semantics but also for the CohesiveLCA semantics (without restricting the size of the results). Both CohesiveLCA and top-1-size CohesiveLCA outperform the other approaches in all three metrics. Top-1-size CohesiveLCA demonstrates perfect precision while CohesiveLCA with a slightly lower precision guarantees perfect recall. These remarkable results on the effectiveness of our approach are obtained thanks to the cohesiveness relation-

dataset	query	# of results				
		CohesiveLCA	SLCA	ELCA	VLCA	MLCA
DBLP	Q_1^D	2	3	4	3	3
	Q_2^D	527	981	982	981	981
	Q_3^D	2	3	4	3	3
	Q_4^D	11	60	61	60	60
	Q_5^D	5	8	9	8	8
PSD	Q_1^P	3	2	3	3	3
	Q_2^P	14	78	79	88	85
	Q_3^P	2	4	4	3	3
	Q_4^P	4	7	8	6	6
	Q_5^P	3	40	43	40	40
NASA	Q_1^N	17	22	24	25	20
	Q_2^N	85	90	90	118	106
	Q_3^N	1	3	4	2	1
	Q_4^N	6	7	8	12	12
	Q_5^N	9	10	11	18	15
Baseball	Q_1^B	10	5	6	1	1
	Q_2^B	7	4	5	0	0
	Q_3^B	216	516	522	516	516
	Q_4^B	145	335	335	335	335
	Q_5^B	49	177	196	177	177

Table 3: Number of results of queries on various datasets

	CohesiveLCA	top-1-size CohesiveLCA	SLCA	ELCA	VLCA	MLCA
Precision %	67.4	100	25.1	27.6	32.6	35.7
Recall %	100	96.9	88.0	93.0	95.0	95.0
$\mathcal{F}$-measure %	76.8	92.8	39.8	36.8	44.4	46.8

Table 4: Average precision, recall and $\mathcal{F}$-measure values over all queries and datasets for all semantics



(a) 10 keywords queries (b) 15 keywords queries (c) 20 keywords queries

Figure 5: Performance of CohesiveLCA for queries with 10, 15 and 20 keywords varying the number of instances

ships which are provided effortlessly by the user.

Ranking cohesive semantics We evaluated our result ranking scheme by computing the Mean Average Precision (MAP) [4] and the Normalized Discounted Cumulative Gain (NDCG) [4] on the queries of Table 2. 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.

MAP (%)			
DBLP	PSD	NASA	Baseball
94	99	94	97

NDCG (%)			
DBLP	PSD	NASA	Baseball
100	99	98	100

Table 5: MAP and NDCG measurements on the four datasets for the queries of Table 2

Table 5 shows the MAP and NDCG values of the cohesive ranking on the queries of Table 2. The excellent values of NDCG show that the ranking in ascending order of the scores of the result LCAs (which take into account the partial LCA sizes and the cohesive term weights) is very close to the correct ranking of the results provided by the expert users. Most MAP values are slightly inferior to 100. This means that a small number of non-relevant LCAs are ranked higher than some relevant. However, the high NDCG values, guarantee that these LCAs are not highly located in the total rank.

4.3 Efficiency of the CohesiveLCA algorithm

In order to study the efficiency of our algorithm we run experiments to measure: (a) its performance scalability on the dataset size, (b) its performance scalability on the query maximum term cardinality and (c) the improvement in efficiency over previous approaches. We used collections of queries with 10, 15 and 20 keywords issued against the DBLP, XMark and NASA datasets. For each query size, we formed 10 cohesive query patterns. Each pattern involves a different number of terms of different cardinalities nested in various depths. For instance, a query pattern for a 10-keyword query is `(xx((xxxx)(xxxx)))`. We used these patterns to generate keyword queries on the three datasets. The keywords were chosen randomly. In order to stress our algorithm, they were selected among the most frequent ones. In particular, for each pattern, we generated 10 different keyword queries and we calculated their average evaluation time. We generated, in total, 100 queries for each dataset. For each query, we run experiments scaling the size of each keyword inverted list from 100 to 1000 instances with a step of 100 instances.

Performance scalability on dataset size. Figure 5 shows how the computation time of CohesiveLCA scales when the total size of the query keyword inverted lists grows. Each plot corresponds to a different query size (10, 15 or 20 keywords) and displays the performance of CohesiveLCA on the three datasets. Each curve corresponds to a different dataset and each point in a curve represents the average computation time of the 100 queries that conform to the 10 different patterns of the corresponding query size. Since the keywords are randomly selected among the most frequent ones this figure reflects the performance scalability with respect to the dataset size.

All plots clearly show that the computation time of CohesiveLCA is linear on the dataset size. This pattern is followed, in fact, by each one of the 100 contributing queries. In all cases, the evaluation times on the different datasets confirm the dependence of the algorithm's complexity on the maximum depth of the dataset: the evaluation on DBLP (max depth 5) is always faster than on NASA (max depth 7) which in turn is faster than on XMark (max depth 11).

It is interesting to note that our algorithm achieves interactive computation times even with multiple keyword queries and on large and complex datasets. For instance, a query with 20 keywords and 20,000 instances needs only 20 sec to be computed on the XMark dataset. These results are achieved on a prototype without the optimizations of a commercial keyword search system. To the best of our knowledge, there is no other experimental study in the relevant literature that considers queries of such sizes.

Performance scalability on max term cardinality. As we showed in the analysis of algorithm CohesiveLCA (Section 3.1), the key factor which determines the algorithm's performance is the maximum term cardinality in the input query. The maximum term cardinality determines the size of the largest sublattice used for the construction of the lattice ultimately used by the algorithm (see Figure 3). This dependency is confirmed in the diagram of Figure 6. We used queries of 10, 15 and 20 keywords with a total number of 6000 instances each on the DBLP dataset. The x axis shows the maximum term cardinality of the queries. The computation time shown by the bars (left y axis) is averaged over all the queries of a query size with the corresponding maximum cardinality. The curve displays the evolution of the size of the largest sublattice as the maximum term cardinality increases. The size of the sublattice is measured by the number of the stacks it contains (right y axis).

It is interesting to observe that the computation time depends primarily on the maximum term cardinality and to a much lesser extent on the total number of keywords. For instance, a query of 20 keywords with maximum term cardinality 6 is computed much faster than a query of 10 keywords with maximum term cardinality 7. This observation shows that as long as the terms involved are not huge, CohesiveLCA is able to efficiently compute queries with a very large number of keywords.

Performance improvement with cohesive relationships. In order to study the improvement in execution time brought by the cohesiveness relationships to previous algorithms, we compare CohesiveLCA with algorithms which compute a superset of LCAs and rank them with respect to their size. In these cases, since taking into account cohesiveness relationships filters out irrelevant results, the quality of the answer is improved. It is not meaningful to compare with other algorithms since their result sets are incomparable to that of CohesiveLCA (no result set is inclusive of the other) and they do not rank their results on their size. For the experiments we used the DBLP dataset—the results on the other datasets are similar.

There are two previous algorithms that can compute the full set of LCAs with their sizes: algorithm LCAsz [13, 14] and algorithm SA [17]. LCAsz is an algorithm that computes all the LCAs with their sizes which, similarly to CohesiveLCA, exploits a lattice of stacks. The SA algorithm [17]

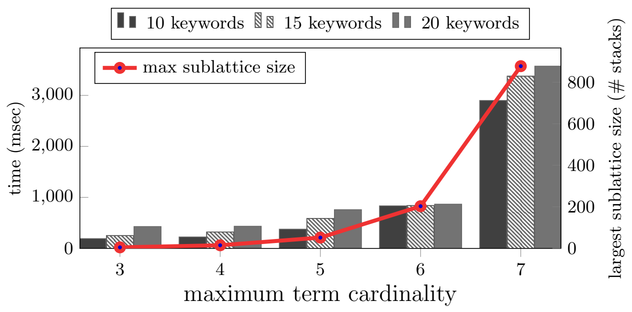


Figure 6: Performance of CohesiveLCA on queries with 6000 keyword instances for different maximum term cardinalities on the DBLP dataset

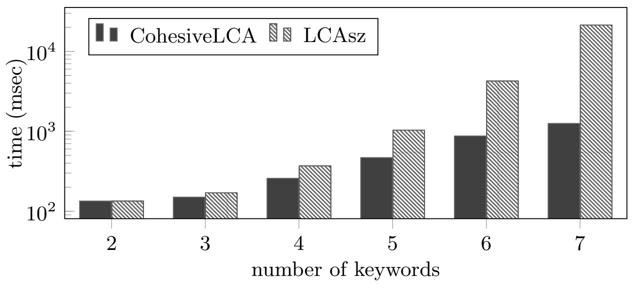


Figure 7: Improvement of CohesiveLCA over LCAsz varying the number of query keywords

computes all LCAs together with a compact form of their matching MCTs, called GDMCTs, which allows determining the size of an LCA. We implemented algorithm SAOne [17] which is a more efficient version of SA since it computes LCAs without explicitly enumerating all the GDMCTs.

Figure 7 compares the execution time of LCAsz and CohesiveLCA in answering keyword queries on the DBLP dataset varying the number of keywords. The execution time of LCAsz is averaged over 10 random queries with frequent keywords. Various cohesiveness relationships patterns were defined for CohesiveLCA (their number depends on the total number of keywords), and for each one of them 10 random queries of frequent keywords were generated. The execution time for CohesiveLCA is averaged over all the queries generated with the same number of keywords. In all cases, each keyword inverted list was restricted to 1000 instances.

As we can see in Figure 7, CohesiveLCA outperforms LCAsz. The improvement in performance reaches an order of magnitude for 6 keywords and increases for 7 keywords or more. Further, CohesiveLCA scales smoothly compared to LCAsz since, as explained above, its performance is dependent on the maximum term cardinality, as opposed to the total number of keywords that determines the performance of LCAsz.

Figure 8 compares the execution times of CohesiveLCA, LCAsz and SAOne for queries of 6 keywords varying the total number of keyword instances. The measurements are average execution times over multiple random queries and cohesiveness relationships patterns (for CohesiveLCA) as in the previous experiment. As one can see, CohesiveLCA clearly outperforms all previous approaches. LCAsz, in turn, largely outperforms SAOne which also scales worse than the other two

5. RELATED WORK

Keyword queries facilitate the user with the ease of freely forming queries by using only keywords. Approaches that evaluate keyword queries are currently very popular especially in the web where numerous sources contribute data often with unknown structure and where end users with no specialized skills need to find useful information. However, the imprecision of keyword queries results often in low precision and/or recall of the search systems. Some approaches combine structural constraints with keyword search [11]. Other approaches try to infer useful structural information implied by keyword queries by exploiting statistical information of the query keywords on the underlying datasets [5, 22, 38, 25, 7]. These approaches require a minimum knowledge

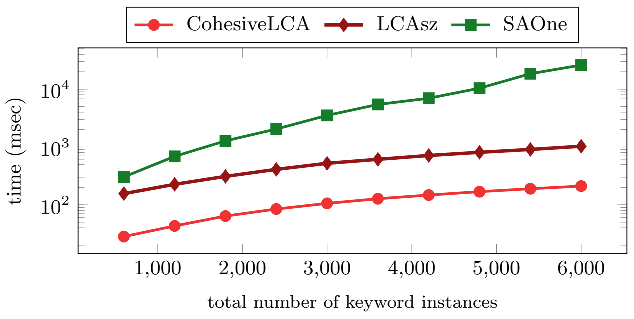


Figure 8: CohesiveLCA scaling comparison with other approaches for queries of 6 keywords

of the dataset or a heavy dataset preprocessing in order to be able to accurately assess candidate keyword query results. A great amount of previous work elaborates on keyword query evaluation on graph data (e.g., RDF databases or graphs extracted from Relational databases) [18, 12, 30, 7]. However, the focus of this work is on tree data.

The task of locating the nodes in a data tree which most likely match a keyword query has been extensively studied in [11, 18, 17, 20, 27, 41, 19, 22, 10, 13, 6, 23, 26, 38, 32, 2, 14, 1]. All these approaches use LCAs of keyword instances as a means to define query answers. The *smallest* LCA (SLCA) semantics [40, 28] validates LCAs that do not contain other descendant LCAs of the same keyword set. A relaxation of this restriction is introduced by *exclusive* LCA (ELCA) semantics [16, 41], which accepts also LCAs that are ancestors of other LCAs, provided that they refer to a different set of keyword instances.

In a slightly different direction, semantic approaches account also for node labels and node correlations in the data tree. *Valuable* LCAs (VLCAs) [11, 20] and *meaningful* LCAs [24] (MLCAs) aim at "guessing" 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 [38].

The efficiency of algorithms that compute LCAs as answers to keyword queries depend on the query semantics adopted. By design they exploit the adopted filtering semantics to prune irrelevant LCAs early on in the computation. Stack based algorithms are naturally offered to process tree data. In [16] a stack-based algorithm that processes inverted lists of query keywords and returns ranked ELCAs was presented. This approach ranks also the query results based on precomputed tree node scores inspired by PageRank [8] and IR style keyword proximity in the subtrees of the ranked ELCAs. An algorithm that computes all the LCAs ranked on LCA size is presented in [13, 14]. In [40], two efficient algorithms for computing SLCAs are introduced, exploiting special structural properties of SLCAs. This approach also introduces an extension of the basic algorithm, so that it returns all LCAs by augmenting the set of already computed SLCAs. Another algorithm for efficiently computing SLCAs for both AND and OR keyword query semantics is developed in [37]. The Indexed Stack [41] and the Hash Count [42] algorithms improve the efficiency of [16] in computing ELCAs. Finally, [5, 6] elaborate on sophisticated ranking of candidate LCAs aiming primarily on effective keyword query answering.

Filtering semantics are often combined with (i) structural and semantic correlations [16, 21, 10, 38, 11, 2], (ii) statistical measures [16, 11, 21, 10, 22, 38] and (iii) probabilistic models [38, 32, 23] to perform a ranking to the results set. Nevertheless, such approaches require expensive preprocessing of the dataset which makes them impractical in the cases of fast evolving data and streaming applications.

The most well known ranking models in the literature of IR assume term independency [4]. Extensions of the basic ranking models such as the generalized vector model [39] and the set based vector model [34] represent queries and documents using sets of terms. However, the ranking scheme in these models requires preprocessing of the data collection to compute $tf * idf$ style metrics for a representative subset of the term vocabulary. The work in [35] enhances keyword queries with structure to extract information from knowledge bases. However, their approach targets graph databases with semantic information and their queries are schema dependent. In contrast, our cohesive queries only contain groupings of the keywords expressing cohesiveness relationships which are not related to any schema construct. As such, the same cohesive query can be issued against any type of dataset (flat text documents, trees, graphs, etc.).

6. CONCLUSION

Current approaches for assigning semantics to keyword queries on tree data cannot cope efficiently or effectively with the large number of candidate results and produce answers of low quality. The convenience and simplicity offered to the user by the keyword queries cannot offset this weakness. In this paper, we claim that the search systems cannot guess the user intent from the query and the characteristics of the data to produce high quality answers on any type of dataset and we introduce a cohesive keyword query language which allows the users to naturally and effortlessly express cohesiveness relationships on the query keywords. We design an algorithm which builds a lattice of stacks to efficiently compute cohesive keyword queries and rank the results leveraging cohesiveness relationships to reduce the lattice dimensionality. A theoretical analysis and experimental evaluation show that our approach outperforms previous approaches in producing answers of high quality and scales smoothly succeeding to evaluate efficiently queries with a very large number of frequent keywords on large and complex datasets where previous algorithms for flat keyword queries fail.

We are currently working on alternative ways for defining semantics for cohesive keyword queries on tree data and in particular in defining skyline semantics which considers all the cohesive terms of a query in order to rank the query results.

7. REFERENCES

[1] C. Aksoy, A. Dimitriou, and D. Theodoratos. Reasoning with Patterns to Effectively Answer XML Keyword Queries. *VLDB Journal, Vol. 24, Issue 3, Springer*, pages 441–465, 2015.

[2] C. Aksoy, A. Dimitriou, D. Theodoratos, and X. Wu. XReason: A Semantic Approach that Reasons with Patterns to Answer XML Keyword Queries. In *DASFAA*, pages 299–314, 2013.

[3] S. Amer-Yahia and M. Lalmas. XML Search: Languages, INEX and Scoring. *SIGMOD Record*, 35(4):16–23, 2006.

[4] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval - the concepts and technology behind search*. Pearson Education Ltd., England, 2011.

[5] Z. Bao, T. W. Ling, B. Chen, and J. Lu. Effective XML Keyword Search with Relevance Oriented Ranking. In *ICDE*, pages 517–528, 2009.

[6] Z. Bao, J. Lu, T. W. Ling, and B. Chen. Towards an Effective XML Keyword Search. *IEEE Trans. Knowl. Data Eng.*, 22(8):1077–1092, 2010.

[7] S. Bergamaschi, F. Guerra, M. Interlandi, R. T. Lado, and Y. Velegrakis. Combining user and database perspective for solving keyword queries over relational databases. *Inf. Syst.*, 55:1–19, 2016.

[8] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks*, 30(1-7):107–117, 1998.

[9] O. C. L. Center. Dewey Decimal Classification, 2006.

[10] L. J. Chen and Y. Papakonstantinou. Supporting top-K Keyword Search in XML Databases. In *ICDE*, pages 689–700, 2010.

[11] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. In *VLDB*, pages 45–56, 2003.

[12] A. Dass, C. Aksoy, A. Dimitriou, and D. Theodoratos. Keyword pattern graph relaxation for selective result space expansion on linked data. In *ICWE*, pages 287–306, 2015.

[13] A. Dimitriou and D. Theodoratos. Efficient keyword search on large tree structured datasets. In *ACM KEYS*, pages 63–74, 2012.

[14] A. Dimitriou, D. Theodoratos, and T. Sellis. Top-k-size keyword search on tree structured data. *Inf. Syst.*, 47:178–193, 2015.

[15] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, pages 436–445, 1997.

[16] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *SIGMOD Conference*, pages 16–27, 2003.

[17] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword Proximity Search in XML Trees. *IEEE TKDE*, 18(4):525–539, 2006.

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

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

[20] G. Li, J. Feng, J. Wang, and L. Zhou. Effective Keyword Search for Valuable LCAs over XML documents. In *CIKM*, pages 31–40, 2007.

[21] G. Li, C. Li, J. Feng, and L. Zhou. SAIL: Structure-aware Indexing for Effective and Progressive top-k Keyword Search over XML Documents. *Inf. Sci.*, 179(21):3745–3762, 2009.

[22] J. Li, C. Liu, R. Zhou, and W. Wang. Suggestion of Promising Result Types for XML Keyword Search. In *EDBT*, pages 561–572, 2010.

[23] J. Li, C. Liu, R. Zhou, and W. Wang. Top-k Keyword Search over Probabilistic XML Data. In *ICDE*, pages 673–684, 2011.

[24] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, pages 72–83, 2004.

[25] X. Liu, L. Chen, C. Wan, D. Liu, and N. Xiong. Exploiting structures in keyword queries for effective XML search. *Inf. Sci.*, 240:56–71, 2013.

[26] X. Liu, C. Wan, and L. Chen. Returning Clustered Results for Keyword Search on XML Documents. *IEEE TKDE*, 23(12):1811–1825, 2011.

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

[28] Z. Liu and Y. Chen. Reasoning and Identifying Relevant Matches for XML Keyword Search. *PVLDB*, 1(1):921–932, 2008.

[29] Z. Liu and Y. Chen. Processing Keyword Search on XML: a Survey. *WWW*, 14(5-6):671–707, 2011.

[30] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: Give me an example of what you need. *PVLDB*, 7(5):365–376, 2014.

[31] B. Mozafari, K. Zeng, L. D'Antoni, and C. Zaniolo. High-performance complex event processing over hierarchical data. *ACM TDS*, 38(4):21, 2013.

[32] K. Nguyen and J. Cao. Top-k Answers for XML Keyword Queries. *WWW*, 15(5-6):485–515, 2012.

[33] P. Ogden, D. B. Thomas, and P. Pietzuch. Scalable XML query processing using parallel pushdown transducers. *PVLDB*, 6(14):1738–1749, 2013.

[34] B. Pôssas, N. Ziviani, W. M. Jr., and B. A. Ribeiro-Neto. Set-based vector model: An efficient approach for correlation-based ranking. *ACM Trans. Inf. Syst.*, 23(4):397–429, 2005.

[35] J. Pound, I. F. Ilyas, and G. E. Weddell. Expressive and flexible access to web-extracted data: a keyword-based structured query language. In *ACM SIGMOD Conference*, pages 423–434, 2010.

[36] A. Schmidt, M. L. Kersten, and M. Windhouwer. Querying XML Documents Made Easy: Nearest Concept Queries. In *ICDE*, pages 321–329, 2001.

[37] C. Sun, C. Y. Chan, and A. K. Goenka. Multiway SLCA-based Keyword Search in XML Data. In *WWW*, pages 1043–1052, 2007.

[38] A. Termehchy and M. Winslett. Using Structural Information in XML Keyword Search Effectively. *ACM Trans. Database Syst.*, 36(1):4, 2011.

[39] S. K. M. Wong, W. Ziarko, and P. C. N. Wong. Generalized vector space model in information retrieval. In *Proceedings of the 8th annual international ACM SIGIR 1985*, pages 18–25, 1985.

[40] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *SIGMOD Conference*, pages 527–538, 2005.

[41] Y. Xu and Y. Papakonstantinou. Efficient LCA based keyword search in XML data. In *EDBT*, pages 535–546, 2008.

[42] R. Zhou, C. Liu, and J. Li. Fast ELCA Computation for Keyword Queries on XML Data. In *EDBT*, pages 549–560, 2010.