The VLDB Journal (2015) 24:441–465
DOI 10.1007/s00778-015-0384-3

REGULAR PAPER

Reasoning with patterns to effectively answer XML keyword queries

Cem Aksoy[1] · **Aggeliki Dimitriou**[2] · **Dimitri Theodoratos**[1]

Received: 10 November 2013 / Revised: 1 November 2014 / Accepted: 23 March 2015 / Published online: 10 April 2015

Abstract Keyword search is a popular technique for searching tree-structured data on the Web because it frees the user from knowing a complex query language and the structure of the data sources. However, the imprecision of the keyword queries usually results in a very large number of results of which only a few are relevant to the query. Multiple previous approaches have tried to address this problem. They exploit the structural properties of the tree data in order to filter out irrelevant results. This is not an easy task though, and in the general case, these approaches show low precision and/or recall and low quality of result ranking. In this paper, we argue that exploiting the structural relationships of the query matches locally in the data tree is not sufficient and a global analysis of the keyword matches in the data tree is necessary in order to assign meaningful semantics to keyword queries. We present an original approach for answering keyword queries which extracts structural patterns of the query matches and reasons with them in order to return meaningful results ranked with respect to their relevance to the query. Comparisons between patterns are realized based on different types of homomorphisms between patterns. As the number of patterns is typically much smaller than that of the of query matches, this global reasoning is feasible. We design an efficient stack-based algorithm for evaluating keyword queries on tree-structured data, and we also devise a heuristic extension which further improves its performance. We run comprehensive experiments on different datasets to evaluate the efficiency of the algorithms and the effectiveness of our ranking and filtering semantics. The experimental results show that our approach produces results of higher quality compared to previous ones and our algorithms are fast and scale well with respect to the input and output size.

Keywords XML keyword search · Keyword query semantics · Patterns · Ranking

1 Introduction

Keyword search has been established in the recent years as the most popular technique for searching the Web. Keyword search became initially popular as a technique for searching flat (unstructured) documents [2], but it soon expanded its popularity to structured [13] and semi-structured data [27]. In this paper, we focus on keyword search on tree-structured data which has become a standard format for exporting and exchanging data on the Web. The reason of the popularity of keyword search on tree-structured data is twofold: (a) the users can retrieve information from the Web without mastering a complex query language (e.g., XQuery) and (b) they can issue queries against the data without having full or even partial knowledge of the structure (schema). Therefore, the same query can be issued against multiple, differently structured data sources on the Web.

The advantages of keyword search come with a drawback. Keyword queries are inherently imprecise since they cannot specify structural constraints. As a consequence, they usually

Electronic supplementary material The online version of this article (doi:10.1007/s00778-015-0384-3) contains supplementary material, which is available to authorized users.

✉ Dimitri Theodoratos
dth@njit.edu

Cem Aksoy
ca64@njit.edu

Aggeliki Dimitriou
angela@dblab.ntua.gr

[1] New Jersey Institute of Technology, Newark, NJ, USA

[2] National Technical University of Athens, Athens, Greece

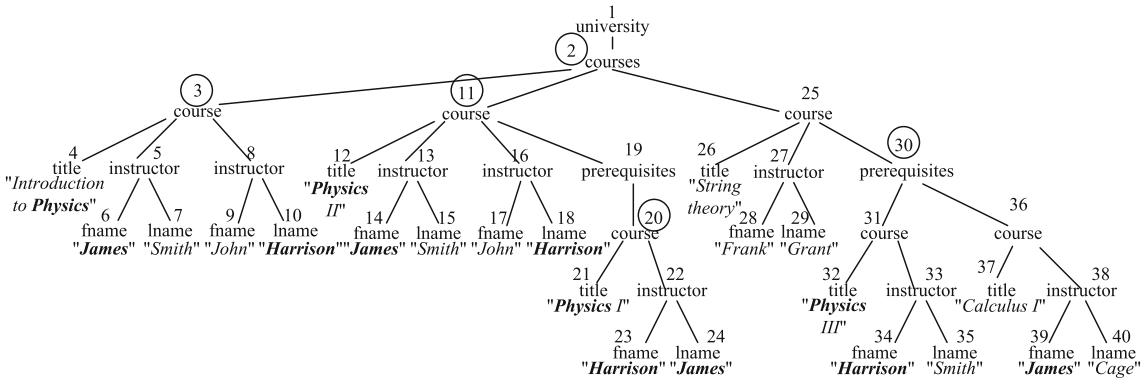


Fig. 1 An XML tree *T*

return a very large number of results of which only a tiny portion are relevant to the query.

The candidate results of a keyword query on an XML tree can be defined as the minimum connecting trees (MCTs) in the XML tree that contains an instance of all the keywords. The roots of the MCTs are the lowest common ancestors (LCA) of the included keyword instances and are often used to identify the candidate results [11,12,34,40,41].

Many recent works focus on addressing the problem of the multitude of candidate LCAs by appropriately assigning semantics to keyword queries on tree data. A number of these semantics, characterized as *filtering*, aim at filtering out a subset of the candidate LCAs that are irrelevant. Some filtering semantics prune LCAs based exclusively on *structural* information (e.g., SLCA semantics [12,25,40] and ELCA semantics [11,41]), while others take also into account *semantic* information, that is, the labels of the nodes in the XML tree (e.g., the valuable LCA or VLCA semantics [9,16], and the meaningful LCA or MLCA semantics [20]). Other recent works assign *ranking* semantics to keyword queries, that is, they rank the results aiming at placing on top those that are more relevant [3,7,9,11,30,38]. Ranking the results improves the usability of the system. In order to perform the ranking, these works exploit: (a) structural characteristics of the results and/or (b) statistical information or information theory metrics adapted to the tree structure of the data. All the ranking approaches rank the results based on some scoring function which assigns scores to the results.

The problems Although filtering approaches are intuitively reasonable for specific cases of data, they are ad hoc and they are frequently violated in practice resulting in low precision and/or recall [38]. Further, most ranking approaches are combined with filtering approaches, that is, they rank only the LCAs accepted by the respective filtering semantics, this way inheriting the low recall of the filtering semantics. This weakness is due to the fact that most existing filtering semantics do not examine and compare the way the keyword instances are combined in the XML tree to form tree patterns. Instead, most of them depend on the structural relationships of the keyword instances and LCAs locally in the XML tree. However, local relationships are not sufficient and a global view of the results is necessary in order to decide effectively on their relevance. Consider for instance the following example.

Example 1 Figure 1 shows a sample XML tree. This tree contains information about courses offered in a University. Consider also the keyword query {*Physics, James, Harrison*} against this XML tree. These keywords have multiple occurrences in the tree which are shown in bold. As one can see, each keyword has multiple instances (nodes that contain the keyword) in the tree. There are courses on Physics offered by one or two instructors whose name contains James and/or Harrison, and therefore, it is reasonable to assume that the user is looking for such courses. Assuming that our results are represented by LCA nodes, there are five candidate results whose node ids are shown in the figure encircled. Among them, only node 3, 11 and 20 are relevant results since node 30 represents the prerequisites of a course and node 2 represents the set of all the courses offered by the University. The ELCA semantics filters out candidate LCAs whose keyword instances are descendants of other descendant candidate LCAs, while SLCA prunes candidate LCAs that are ancestors of other candidate LCAs. In the context of our example, both ELCA and SLCA return the wrong result 30 and SLCA misses the correct result 11. One version of the MLCA semantics excludes candidate LCAs if the LCA of two of its keyword instances is not also an ELCA node of the labels of these instances. Therefore, it fails to return the correct result 3 because the LCA of the keyword `fname` (the label of the instance 6 of keyword *James*) and `lname` (the label of the instance 10 of keyword `Harrison`) is not also an ELCA of `fname` and `lname` (the label lname of node 10 comes closer to the label fname of node 9). Further, it incorrectly returns the node 30 (`prerequisites`).

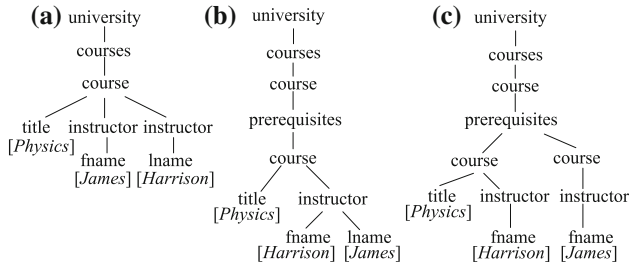


Fig. 2 Three patterns of the query *Physics, James, Harrison* on the XML tree of Fig. 1

The reason of these failures is that all these approaches are based on the relationship of the keyword instances locally (e.g., whether two LCAs have an ancestor–descendant relationship) and they miss a global view of how the keyword instances are combined in the data tree.

Our solution In this paper, we argue that a meaningful semantics for keyword queries does not depend on the local properties of the keyword instances in the XML tree but on the *patterns* the keyword instances define on the XML tree. Each pattern might and usually does represent many results. For instance, Fig. 2 shows three patterns for the query {*Physics, James, Harrison*} on the XML tree of Fig. 1. These patterns which are minimal trees rooted at the root of the XML tree and containing all the keywords indicate different ways the keyword instances in the XML tree are combined to form results. Pattern (a) represents the LCAs 3 and 11, pattern (b) the LCA 20 and pattern (c) the LCA 30.

Further, we argue that assigning simply a score to the results is not sufficient for producing a ranking of high quality. Rather, a ranking or a filtering of the results should be obtained by directly comparing the different structural and semantic properties of their patterns, that is, by *reasoning* on the patterns. For instance, by comparing the patterns of Fig. 2 one can easily see that pattern (c) is not relevant in the presence of patterns (a) and (b) and if one wants to rank the patterns in terms of relevance to the query, patterns (a) and (b) should precede pattern (c). These considerations apply to all the results these patterns represent. That is, the patterns act as representatives of the respective results.

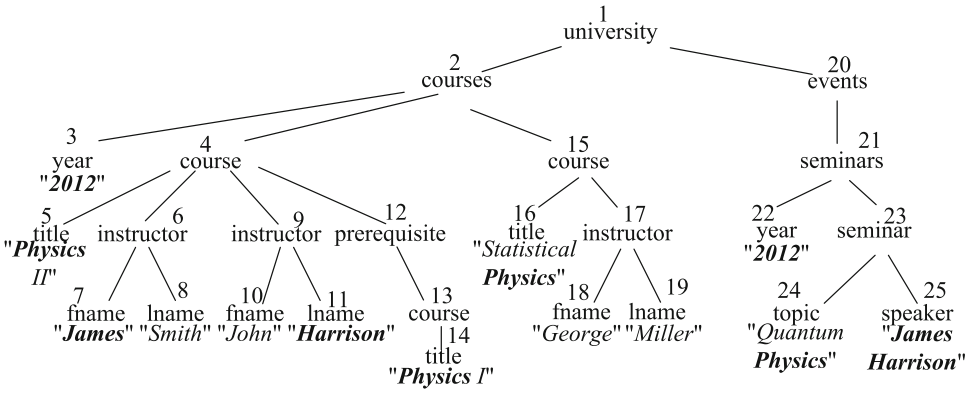
Different techniques could be devised to compare patterns. In the present work, this comparison is realized based on *homomorphisms* between patterns. As the number of patterns is typically much smaller than the number of the results they represent, we show that this comparison is computationally feasible.

A number of approaches define filtering or ranking semantics without relying on the structural relationships of the keyword instances locally in the XML tree. As an example, Cohen et al. [9] filter out a query match containing a pair of keyword instances linked through a path in the XML tree which has duplicate labels. Schema-level SLCA [15] excludes LCAs whose label paths from the root of the XML tree are a proper prefix of that of another LCA. More recently, Liu et al. [22] cluster query results based on the patterns they comply with and define filtering semantics based on the conceptual relationships between entity nodes (in the sense of the Entity-Relationship model) in the XML tree. Finally, Coherency Ranking (CR) [38] ranks query results based on an extension of the concepts of data dependencies and mutual information. None of these approaches define ranking or filtering semantics by globally comparing the structural and semantic properties of the query result patterns as we do in this paper.

Contribution The main contributions of this paper are the following:

- We define the answer of a keyword query based on the concept of instance tree (IT) which records not only how keyword instances are combined to form an MCT but also how the root of the MCT is linked to the root of the XML document. We introduce the concept of pattern of a keyword query on an XML tree which records the structural relationships between node labels and keywords in an IT. Every pattern represents the set of ITs in the XML tree that comply with it.
- In order to enable reasoning over patterns, we introduce two types of homomorphisms between patterns and we use them to define different kinds of homomorphism relations on patterns.
- Based on these relations, we organize the patterns of a keyword query into a graph of patterns which we leverage to determine a ranking for patterns and ranking semantics for queries thereof. We also provide filtering semantics for queries by selecting the top-k patterns in the ranking. Our semantics is named *XReason*.
- In order to bring all the previous filtering approaches to a common ground, we express their semantics in terms of ITs. We provide a comprehensive comparison of them, and we show how they differ from *XReason*.
- We design an efficient algorithm to implement *XReason* which comprises two modules: (a) a stack-based algorithm for computing the query patterns and the ITs that comply with them and (b) an algorithm that builds the pattern graph and generates a ranking for the patterns. Contrary to the previous algorithms implementing ranking semantics for keyword queries which rely on auxiliary structures on the datasets [3,9,11,38], our algorithm does not require a preprocessing of the datasets and uses only the inverted lists of the keywords in order to produce the query answers. We also design a heuristic extension of our algorithm which substantially

Fig. 3 An XML tree T



improves efficiency without compromising the quality of the results.
- We run comprehensive experiments on multiple datasets having different characteristics and compared our approach with previous ones in order to assess the effectiveness of *XReason* and the efficiency of our algorithms. Our results show that the *XReason* filtering and ranking semantics outperform previous approaches with respect to various metrics and our algorithms are fast and scale well with respect to the input and output size.

Outline The rest of the paper is organized as follows: Sect. 2 defines concepts that will be used throughout the paper. Section 3 introduces homomorphisms between patterns and homomorphism relations on patterns and shows how these are used to define the *XReason* ranking and filtering semantics. In Sect. 4, the differences in *XReason* from the previous filtering semantics are analyzed. The algorithm for implementing the *XReason* semantics is presented in Sect. 5. The details of our experimental evaluation and results are provided in Sect. 6. Finally, Sect. 7 presents the related work, and Sect. 8 contains concluding remarks and pointers to future work.

2 Preliminaries

2.1 Data model

As is usual, we view XML documents as ordered node labeled trees. Nodes represent and are labeled by elements and attributes. The nodes may have a content which is text. Edges represent element to element and element to attribute relationships. For any two nodes n and n' in an XML tree T, $n < n'$ ($n > n'$) denotes that n is an ancestor (descendant) of n' in T. Without loss of generality, we assume that the label of the root of the XML tree is unique. A function *label* on a node returns the label of that node. We want to allow keywords to match not only the content of a node but also its label. To this end, we define a function *value* on nodes which returns the set of words in the content *and* the label of the node. If $value(n)$ of a node n includes a keyword k, we say that n *contains* keyword k and that node n is an *instance* of k. We assume that XML tree nodes are enumerated using the Dewey encoding scheme [37]. The Dewey encoding scheme allows easily determining the LCA of multiple nodes and can be efficiently exploited by stack-based algorithms for computing query matches.

Figure 3 shows an XML tree which is a variation of the one shown in the introduction. Dewey codes are omitted for clarity. Plain numbers are used instead to identify the nodes.

2.2 Keyword queries

A *(keyword) query* Q is a set of keywords $\{k_1, k_2, \ldots, k_n\}$. Keyword queries are embedded to XML trees.

Definition 1 Let Q be a query and T be an XML tree. An *instance* of Q on T is an embedding of Q to T (i.e., a function from Q to the nodes of T that maps every keyword k in Q to an instance of k in T).

We overload the term "query instance," and we use it to refer both to the function that maps the query keywords to the tree nodes and to the images of the query keywords under this function. Note that two query keywords can be mapped to the same tree node.

Definition 2 Let Q be a query, T be an XML tree, and I be an instance of Q on T. The *instance tree (IT)* of I is the minimum subtree S of T such that: (a) S is rooted at the root of T and comprises all the nodes of I, and (b) every node n in S is annotated by the keywords which are mapped by I to n. The *minimum connecting tree (MCT)* of I is the minimum subtree of S that comprises the nodes of I.

Clearly, the root of the MCT is the *lowest common ancestor* (LCA) of the nodes of I in T.

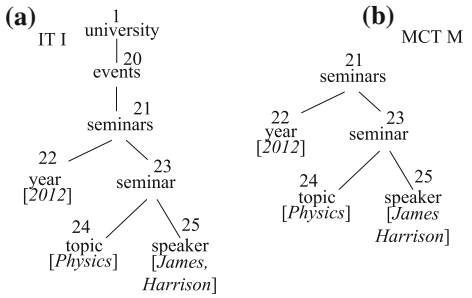


Fig. 4 **a** An IT and **b** its MCT

Consider the XML tree of Fig. 3 and the keyword query $Q = \{Physics, James, Harrison, 2012\}$. Figure 4a, b show the IT and the MCT, respectively, of the instance $\{(Physics, 24), (James, 25), (Harrison, 25), (2012, 22)\}$ of Q on T. Annotations are shown between square brackets by the nodes in the figure. The MCT of this IT is rooted at node 21 (the LCA of the keyword instances). The IT also contains the path from the root of T to node 21. In the following, we identify an IT and an MCT by their corresponding query instance.

Given a keyword query Q and an XML tree T, the set $\mathcal{C}$ of the ITs of all the instances of Q on T is the set of the candidate results of Q on T. The answer of a keyword query Q on an XML tree T is a *subset* of $\mathcal{C}$. Which specific subset forms the answer of a query depends on the semantics adopted. In our approach, the answer is determined by comparing the patterns (to be defined below) of the ITs. For the needs of this paper, this comparison is realized based on different types of homomorphisms. Next, we define formally these homomorphisms, and then, we use them to provide ranking and filtering semantics to the queries. Other semantics will be presented in terms of IT and compared with our approach in Sect. 4.

3 Query semantics

In order to define semantics for queries, we introduce patterns of ITs and homomorphisms between patterns and study their properties.

3.1 IT patterns

Definition 3 (*IT pattern*) A *pattern* P of a query Q on an XML tree T is a tree which is isomorphic (including the annotations) to an IT of Q on T. The MCT of a pattern P refers to P without the path that links the LCA of the annotated nodes to the root of P.

Multiple ITs of Q on T can share the same pattern. Figure 5 shows eight patterns (out of 15 in total) of the keyword query $Q = \{Physics, James, Harrison\}$ on the XML tree T of Fig. 3. All patterns except pattern P_8 have one IT. Pattern P_8 has two ITs which comply with it: the IT of the query instance $\{(Physics, 5), (James, 25), (Harrison, 25)\}$ and the IT of the query instance $\{(Physics, 16), (James, 25), (Harrison, 25)\}$.

The function $ann(n)$ on a node n of a pattern returns the annotation of n if the node is annotated or, an empty set, otherwise. We also define a function $size(P)$ which returns the number of edges of P. The size of pattern P_3 is 8, and that of its MCT is 7. The size of pattern P_1 is 9, and that of its MCT is 7.

3.2 Pattern homomorphism and homomorphism relation

Definition 4 (*Pattern homomorphism*) Let S and S' be two subtrees of patterns of a query on an XML tree. A *homomorphism* from S to S' is a function h from the nodes of S to the nodes of S' such that:

(a) for every node n in S, n and $h(n)$ have the same labels.
(b) if n_2 is a child of n_1 in S, $h(n_2)$ is a child of $h(n_1)$ in S' and
(c) for every node n in S, $ann(n) \subseteq ann(h(n))$.

Figure 6 shows the MCTs M, M' and M'' of three patterns of the query $Q = \{2012, James, Harrison\}$ on an XML tree. As we can see in this figure, there are homomorphisms from M to M' and from M' to M'' but not from M' to M or from M'' to M'. Observe that M' can be obtained from M (and M'' from M') by merging paths with the same sequence of labels starting from the same node and by unioning their annotations. For instance, M'' can be obtained from M' by merging the paths `seminars/seminar/speaker` from the node `speaker` and by unioning the annotations $[James]$ and $[Harrison]$. One can see that, in general, this is also a necessary condition for the existence of a homomorphism between two patterns.

Clearly, if there is a homomorphism from the MCT of a pattern P to the MCT of a pattern P', the keyword instances are more closely related in any instance of P' than in P. Thus, we consider the ITs of P' to be more relevant to the query than those of P.

Based on the existence of a homomorphism between two patterns, we can define a relation $\prec_h$ (called *homomorphism* relation) on patterns in order to compare their relevance to the query.

Definition 5 ($\prec_h$ *relation*) Let P and P' be two patterns of a query Q on an XML tree T. $P \prec_h P'$ iff there is a homomorphism from the MCT of P' to the MCT of P but not vice versa.

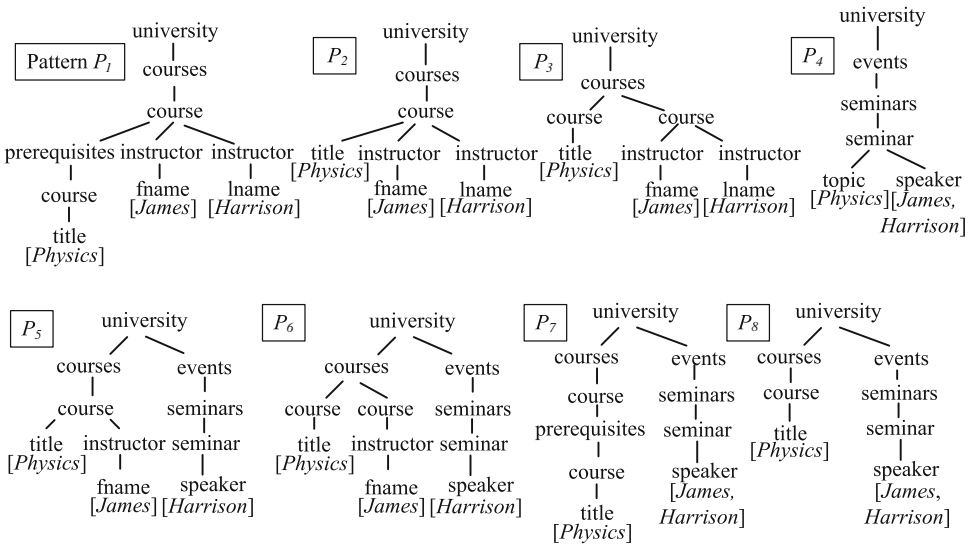


Fig. 5 Some patterns for $Q = \{Physics, James, Harrison\}$ on the tree of Fig. 3

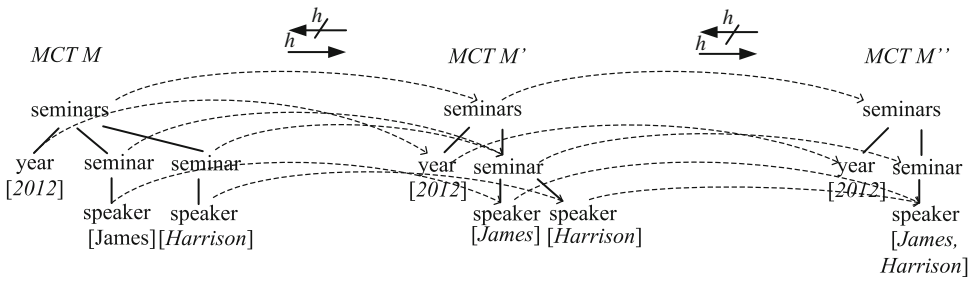


Fig. 6 Pattern MCTs M, M' and M'' and homomorphisms between them

For instance, for the patterns P, P' and P'' whose MCTs M, M' and M'', respectively, are shown in Fig. 6, $P'' \prec_h P' \prec_h P$, but $P' \not\prec_h P''$ and $P \not\prec_h P'$. That is, P' is more relevant than P, and P'' is more relevant than P'. Similarly, for the patterns P_5 and P_6 of our running example shown in Fig. 5, one can see that $P_5 \prec_h P_6$.

The following property which will be used later can be easily shown for the $\prec_h$ relation.

Proposition 1 *The relation $\prec_h$ on the set of patterns of a query on an XML tree is a strict partial order.*

Even though $\prec_h$ correctly characterizes relevance, it is not sufficient. Consider for instance the MCTs M_1 and M_2 of the patterns P_1 and P_2, respectively, shown in Fig. 7 (ignore the dashed arrows for the moment). Even though $P_2 \not\prec_h P_1$, P_2 is more relevant than P_1. Indeed, P_2 relates two instructors to a course they offer, while P_1 relates two instructors to a prerequisite of a course they offer. In the next section, we further exploit different kinds of relations in order to better capture this relevance relationship between patterns and their ITs thereof.

3.3 Path homomorphism and path homomorphism relations

In order to define additional relations on patterns, we introduce below a new type of homomorphism.

Definition 6 (*Path homomorphism*) Let p and p' be two paths of two patterns, such that p ends at a node n annotated by a keyword k. We say that there is a *path homomorphism* from p to p' if there is a function ph from the nodes of p to the nodes of p' such that:

(a) for every node n_1 in p, n_1 and $ph(n_1)$ have the same labels.
(b) if n_2 is a child of n_1 in p, $ph(n_2)$ is a child of $ph(n_1)$ in p', and
(c) $k \in ann(ph(n)) \cup label(ph(n))$.

Figure 7 shows the MCTs M_2 and M_1 of the corresponding patterns P_2 and P_1 (shown in Fig. 5) for the query $\{Physics, James, Harrison\}$ on the XML tree of Fig. 3. For every path from the root of M_2 to a node annotated by a keyword, there is a path homomorphism to a path in M_1 (the different types of dashed arrows indicate

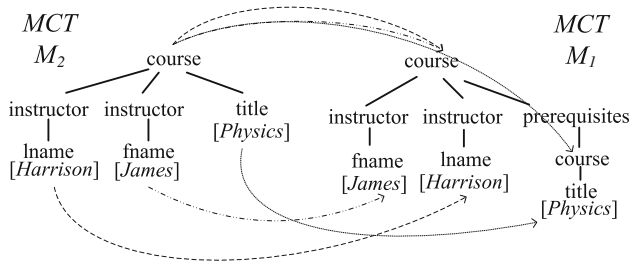


Fig. 7 Pattern MCTs M_2 and M_1 and three path homomorphisms from the paths of M_2 to paths of M_1

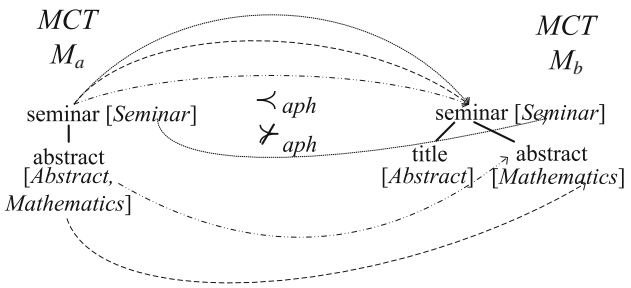


Fig. 8 Pattern MCTs M_a and M_b and three path homomorphism from the paths of M_a to paths of M_b

these path homomorphisms). However, the opposite is not true, that is, there is at least one path of M_1 (in fact, only the path `course/prerequisites/course/title` [*Physics*]) that does not have a path homomorphism to a path in M_2.

Our intuition is that if P and P' are two patterns of a query on an XML tree, and every path from the root of the MCT of P to a keyword annotated node has a path homomorphism to a path in the MCT of P', then the keyword instances in P are more meaningfully related than in P' because every sequence of labels from the LCA to a keyword instance in P also appears in P'.

In order to compare the relevance of query patterns, we use now path homomorphisms to define a relation $\prec_{aph}$ (called *all_path_homomorphism* relation) on patterns.

Definition 7 ($\prec_{aph}$ *relation*) Let P and P' be two patterns of a query Q on an XML tree T. $P \prec_{aph} P'$ iff the following two conditions hold:

(a) for every path p from the root of the MCT of P to a node annotated by a keyword, there is a path homomorphism from p to a path of the MCT of P'.
(b) Property (a) does not hold in the opposite direction, that is, from P' to P.

As an example, observe that for the pattern MCTs M_2 and M_1 of Fig. 7, $P_2 \prec_{aph} P_1$. That is, the $\prec_{aph}$ relation correctly characterizes P_2 as more relevant than P_1.

Consider also another example which involves mapping a keyword to a label or a value of a node: Fig. 8 shows the MCTs M_a and M_b of two query patterns P_a and P_b, respectively. As it can be seen from the annotations, the keyword query is $\{Seminar, Abstract, Mathematics\}$. $P_a \prec_{aph} P_b$ since the three paths `seminar`[*seminar*], `seminar/abstract`[*abstract*] and `seminar/abstract`[*mathematics*] in P_a have a path homomorphism to a path in P_b, but the path `seminar/ title`[*abstract*] of P_b does not have a path homomorphism to path in P_a. Here, again $\prec_{aph}$ correctly favors P_a over P_b. We now show a property of relation $\prec_{aph}$.

Proposition 2 *The relation $\prec_{aph}$ on the set of patterns of a query on an XML tree is a strict partial order.*

We next examine how $\prec_h$ and $\prec_{aph}$ are related. In Fig. 6, one can see that besides homomorphisms from M to M', and M' to M'', for every path p from the root to an annotated node of an MCT, there is a path homomorphism to a path in any one of the other MCTs. This is expected due to the following proposition.

Proposition 3 *Let M and M' be two pattern MCTs of a query on an XML tree. If there is a homomorphism from M to M', then: (a) for every path p from the root of M to a node annotated by a keyword, there is a path homomorphism from p to a path of M' and (b) for every path p' from the root of M' to a node annotated by a keyword, there is a path homomorphism from p' to a path of M.*

Nevertheless, if for every path p from the root of M to a node annotated by a keyword, there is a path homomorphism from p to a path of M', then there is not necessarily a homomorphism from M to M' or from M' to M.

As a consequence of Proposition 3 and Definition 6, if $P \prec_h P'$, then $P \not\prec_{aph} P'$ and $P' \not\prec_{aph} P$.

Often, it is the case that an XML tree integrates data for entity types which are unrelated. For instance, the University XML tree of Fig. 3 involves courses and seminars. Instances of these entity types are not related except that they share the root of the XML tree as the only common ancestor. In such a context, the $\prec_{aph}$ relation does not help us compare effectively the relevance of patterns that involve labels from different entity types with patterns that involve labels from the same entity type.

Consider, for instance, the patterns P_4 and P_5 of Fig. 9 for the query $\{Physics, James, Harrison\}$ on Fig. 3. These patterns are not related with respect to $\prec_{aph}$. However, P_4 is more relevant than P_5 since it meaningfully brings together a speaker of a seminar with the topic of the seminar. In contrast, P_5 involves both a course and a seminar and brings together the speaker of a seminar with the instructor and title of a course.

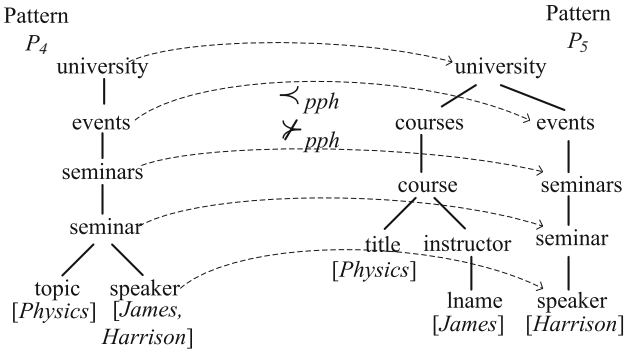


Fig. 9 A path homomorphism from a path of pattern P_4 to a path of P_5

Our intuition is that if two patterns share the same root-to-leaf path (including the annotations), the pattern P_1 whose MCT root R_1 is a descendant of the MCT root R_2 of the other pattern P_2 in this path (that is, R_1 is deeper than R_2 in the common path) more meaningfully relates the keyword instances than P_2.

In order to enable relevance comparisons between patterns that involve unrelated parts of an XML tree, we define below a relation $\prec_{pph}$ (called *partial_path_homomorphism* relation) on query patterns.

Definition 8 ($\prec_{pph}$ *relation*) Let P and P' be two patterns of a query Q on an XML tree T and p be a path from the root of P to an annotated node of P. $P \prec_{pph} P'$ iff there is a path homomorphism ph of p to a path in P' such that:

(a) the root of P is mapped by ph to the root of P'.
(b) the root of the MCT of P is mapped by ph to a node which is a descendant (not self) of the root of the MCT of P', and
(c) $P' \not\prec_h P$ and $P' \not\prec_{aph} P$.

Condition (c) is included for the purpose of guaranteeing the acyclicity of the relations $\prec_{aph}$ and $\prec_{pph}$ between two patterns.

Consider again the patterns P_4 and P_5 of Fig. 9. As shown in the figure, there is a path homomorphism from the path `university/events/seminars/seminar/speaker`[$Harrison$] of P_4 to the same path of P_5 and the image of the root of the MCT of P_4 (`seminar`) under this homomorphism is a descendant of the root of the MCT of P_5 (`university`). Therefore, $P_4 \prec_{pph} P_5$. That is, the $\prec_{pph}$ relation correctly finds P_4 to be more relevant than P_5.

Because the roots of the MCTs of two patterns related through a $\prec_{pph}$ relation are required to have a descendant relationship, it is easy to see that $\prec_{pph}$ relation has the following property.

Proposition 4 *The relation $\prec_{pph}$ is acyclic.*

3.4 *XReason* semantics

We use homomorphism relations to define filtering and ranking semantics to keyword queries called *XReason* semantics. We first define a precedence relation, $\prec$, on patterns which combines the three homomorphism relations[1].

Definition 9 Let P and P' be two patterns of a query Q in an XML tree T. $P \prec P'$ iff $P \prec_h P'$ or $P \prec_{aph} P'$ or $P \prec_{pph} P'$.

Based on the previous discussion, one can see that the following property holds for the precedence relation on patterns.

Proposition 5 *The relation $\prec$ on the set of patterns of a query on an XML tree is acyclic.*

Given the relation $\prec$ on the set of patterns of a query Q on an XML tree T, consider a directed graph $G_\prec$ such that: (a) the nodes of $G_\prec$ are the patterns of Q on T, and (b) there is an edge in $G_\prec$ from node P to node P' iff $P \prec P'$. Clearly, because of Proposition 5, $G_\prec$ is *acyclic*. Figure 10a shows the graph $G_\prec$ for the relation $\prec$ on the set of patterns of query $Q = \{Physics, James, Harrison\}$ on the XML tree of Fig. 3. There are 15 such patterns, and eight of them (patterns P_1 - P_8) are shown in detail in Fig. 5. The edges are labeled by letters h, a and/or p to indicate which of the relations, respectively, $\prec_h$, $\prec_{aph}$ and $\prec_{pph}$ relate its nodes. Transitive a-edges which are not p-edges are omitted to reduce the clutter. Since $G_\prec$ is acyclic, it has at least one source node (i.e., a node without incoming edges). The one of Fig. 10 has two source nodes (pattern P_2 and P_4).

In the graph of Fig. 10a, observe that all the nodes (patterns) can be partitioned in levels based on their maximum distance ($GLevel$) from a source node. For instance, in level 3 there are patterns P_3 and P_7. The patterns in one level can also be further distinguished based on the depth of their MCT root in the pattern ($MCTDepth$) and the size of their MCT ($MCTSize$). We create an order $\mathcal{O}$ for the patterns in $G_\prec$ which ranks them in: (a) ascending order of $GLevel$, (b) descending order of $MCTDepth$ and (c) ascending order of $MCTSize$. Note that two patterns might be placed at the same rank in $\mathcal{O}$. The order $\mathcal{O}$ does not distinguish between these patterns. In our running example, one can see that the 15 patterns of Fig. 10a are ordered with respect to $\mathcal{O}$ as shown in Fig. 10b.

Definition 10 (*Ranking XReason semantics*) According to the ranking XReason semantics, an answer of a query Q on an XML tree T is a list of the ITs of Q on T ranked in an order which complies with the order $\mathcal{O}$ of their patterns.

[1] The homomorphism relations can also be related to the concept of preference relation in measurement theory.

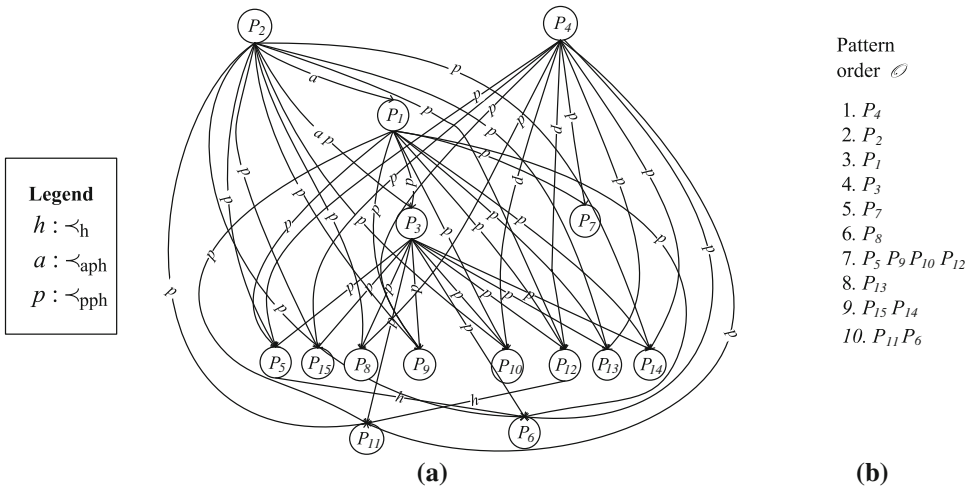


Fig. 10 **a** Graph $G_{\prec}$, **b** pattern order $\mathcal{O}$

In our running example, we have 15 patterns and 16 ITs. An ordering of these ITs which complies with the order $\mathcal{O}$ of patterns shown in Fig. 10b is an answer of the query $Q = \{Physics, James, Harrison\}$ on the XML tree of Fig. 3.

We use the patterns at the top-k levels of $G_{\prec}$ to define filtering semantics.

Definition 11 (*Filtering XReason semantics*) According to the filtering *XReason* semantics, the answer of Q on T is the set of ITs of the patterns of Q on T with the smallest k $GLevel$ values.

Parameter k is user defined. Usually, k is chosen to be equal to one (i.e., we choose the top-level patterns) in which case the answer of Q on T is the set of ITs whose patterns are source nodes in the $G_{\prec}$ graph.

Based on the previous definition and for k=1, the answer of query $Q = \{Physics, James, Harrison\}$ on the XML tree of Fig. 3 is the set of ITs which comply with patterns P_2 or P_4 (the two source nodes in graph $G_{\prec}$ of Fig. 10a). There are only two ITs which comply with these patterns—one for each pattern: the IT of P_2 with leaf nodes 5, 7 and 11 whose MCT is rooted at `course` and the IT of P_4 with leaf nodes 24 and 25 whose MCT is rooted at `seminar`.

It is interesting to note that according to *XReason* semantics, an IT might more meaningfully relate its keyword instances than another IT even though these ITs do not have any location proximity, that is, they do not share any node and their LCAs are not in descendant-or-self relationship. This feature departs from previous traditional filtering semantics (e.g., $ELCA$ [11,41], $SLCA$ [7,12,40], $MLCA$ [20]) where the location proximity is required in order to privilege one IT over another. In the next section, we analyze the filtering *XReason* semantics in relation to previous semantics.

4 Analysis of *XReason*

In this section, we compare the *XReason* filtering semantics with different previous filtering semantics from the literature. For determining the query answer with *XReason*, only ITs of the top level patterns in $G_{\prec}$ are retained. We show cases where the other approaches miss meaningful answers or return meaningless answers while *XReason* does not. This demonstration proves also that *XReason* is different than all the previous approaches.

Many approaches to keyword search in the literature return LCA nodes as answers to keyword queries. In this paper, the answer of a keyword query is a set of ITs (see Definition 2). The ITs more precisely capture the subtleties of the different semantics than the LCAs since the same LCA can be the root of multiple MCTs of different ITs for a query. In order to set up a common ground for comparison, we define the previous approaches in terms of ITs. We go through these approaches with an example, but we provide before a summary of their formal definitions. Let $Q = \{k_1, k_2, \ldots, k_n\}$ denote a keyword query and T denote an XML tree. Let also $LCAset(Q, T)$ be the set of LCAs of Q on T and $ITset(Q, T)$ be the set of ITs of the instances of Q on T. Table 1 provides formal definitions of the previous semantics in terms of ITs.

If the answer of a query according to semantics A is a subset of the answer of this query according to semantics B for any query and on any XML tree, we say that B contains A and we write $A \subseteq B$. Containment relationships between the different approaches based on the definitions of Table 1 are shown in Fig. 11. Containment relationships between some filtering approaches are also provided in [27]. However, the semantics defined in [27] are based on LCAs, while the semantics we defined in this

Table 1 Formal definitions of different previous filtering semantics in terms of ITs

Semantics	Definition of the answer of Q on XML tree T
$SLCA$ [7,12,40]	$\{t \mid t \in ITset(Q,T), n = root(MCT(t))$, and $\nexists\, t',\ n'\ (t' \in ITset(Q,T), n' = root(MCT(t'))$ and $n' > n)\}$
$ELCA$ [11,41]	$\{t \mid t \in ITset(Q,T), n = root(MCT(t))$, and $\nexists\, n'(n'$ is a node in $MCT(t), n \neq n'$ and $n' \in LCAset(Q,T)\}$
$VLCA$ [9,16] (*all pairs related*)	$\{t \mid t \in ITset(Q,T)$ and $\forall$ pair of annotated nodes n_i, n_j in t, $\nexists$ distinct nodes n_k, n_l in the path between n_i and n_j s.t. $label(n_k) = label(n_l)$ unless $n_k = n_i$ and $n_l = n_j\}$
$CVLCA$ [16]	$VLCAset(Q,T) \cap ELCAset(Q,T)$ where $VLCAset(Q,T)$ (resp. $ELCAset(Q,T)$) is the answer of Q on T according to $VLCA$ (resp. $ELCA$) semantics
$MLCA_{label}$ [20,38]	$\{t \mid t \in ITset(Q,T)$ and $\forall$ pair of annotated nodes n_i, n_j in $MCT(t)$ of two distinct keywords, $\nexists$ node n in T s.t. $label(n) = label(n_j)$ and $LCA(n_i,n) > LCA(n_i,n_j)$ in $T\}$
$MLCA_{value}$[20]=$pairwiseELCA$	$\{t \mid t \in ITset(Q,T)$ and $\forall$ pair of nodes n_i and n_j in $MCT(t)$ annotated by the keywords k_i and k_j, respectively, $\nexists$ instance n'_j of k_j in T s.t. $LCA(n_i,n'_j) > LCA(n_i,n_j)$ in $T\}$
$MLCA_{label+value}$ [20]	$\{t \mid t \in ITset(Q,T)$ and $\forall$ pair of nodes n_i and n_j in $MCT(t)$ annotated by the keywords k_i and k_j, respectively, $\nexists$ instance n'_j of k_j in T s.t. $label(n'_j) = label(n_j)$ and $LCA(n_i,n'_j) > LCA(n_i,n_j)$ in $T\}$
$MaxMatch$ [25]	$\{t \mid t \in SLCAset(Q,T)$ and $\forall$ node n in $MCT(t)$, $\nexists$ sibling node n' of n in T s.t. the set of keywords occurring in the subtree rooted at n in T is a proper subset of the set of keywords occurring in the subtree rooted at n' in $T\}$
$pairwiseSLCA$	$\{t \mid t \in ITset(Q,T)$ and $\forall$ pair of nodes n_i and n_j annotated by keywords k_i and k_j, respectively, in $MCT(t)$, $\nexists$ instances n'_i and n'_j of the same keywords in T s.t. $LCA(n'_i,n'_j) > LCA(n_i,n_j)$ in $T\}$
Schema-level SLCA [15]	$\{t \mid t \in ITset(Q,T)$ and $\nexists\, t' \in ITset(Q,T)$ s.t. the root-to-LCA label path of t is a proper prefix of the root-to-LCA label path of $t'\}$

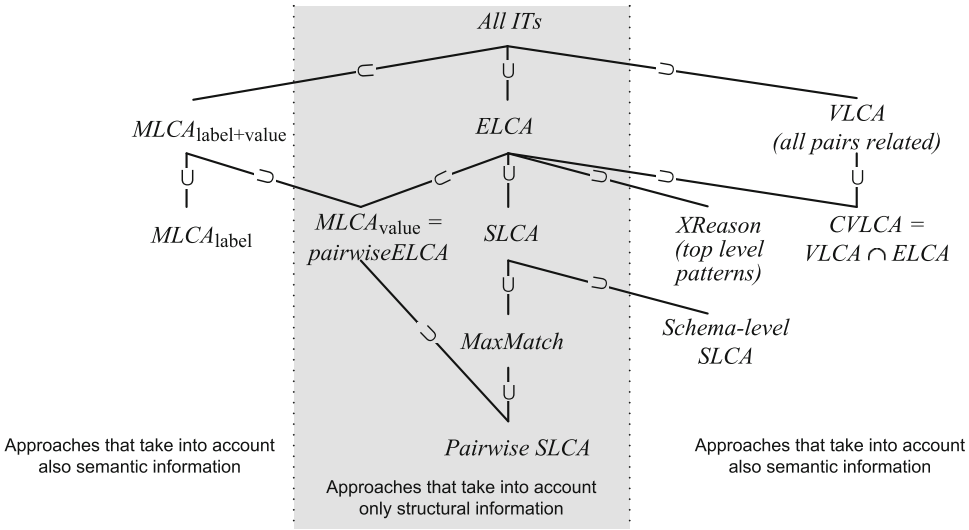


Fig. 11 Containment relationships between different filtering semantics

paper are based on ITs. Consider the query, $Q = \{Physics, James, Harrison\}$ on the XML tree of Fig. 12. With this query, the user requests information about a course or seminar on *physics* which is offered by *James Harrison* or by *James* and *Harrison*. The relevant ITs to Q are, $IT_1 = \{(Physics, 4), (James, 6), (Harrison, 10)\}$, $IT_2 = \{(Physics, 13), (James, 16), (Harrison, 15)\}$ and $IT_3 = \{(Physics, 40), (James, 41), (Harrison, 41)\}$. The LCAs of their keyword instances are (3, `course`), (12, `course`) and (39, `seminar`) and are labeled l_1, l_2 and l_3, respectively, in Fig. 12.

$SLCA$ [7,12,40] eliminates an IT whose LCA is an ancestor of another IT's LCA. Therefore, it fails to return IT_1 because there is another IT, IT_2, whose LCA (node 12) is a descendant of IT_1's LCA (node 3). Missing correct results reduces the recall of the approach. Moreover, $SLCA$ returns

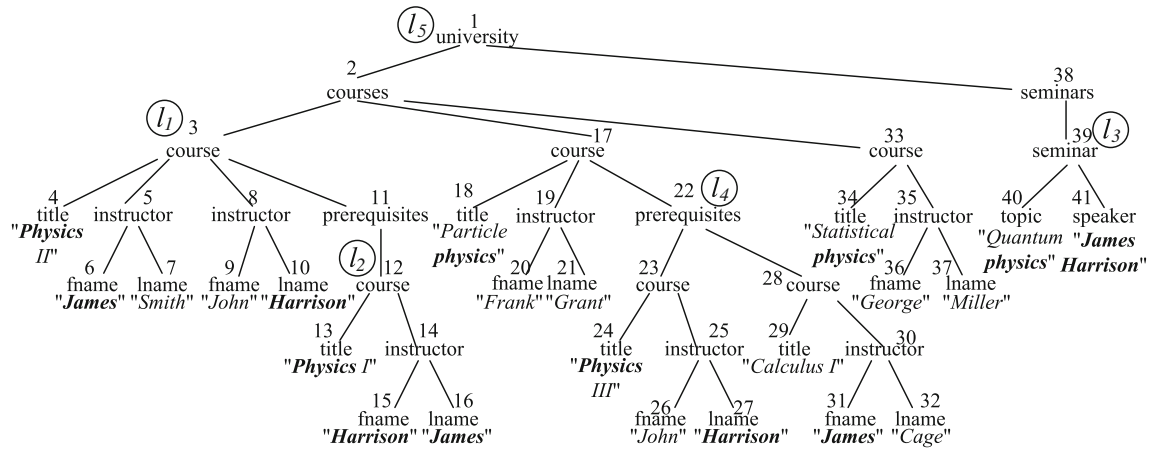


Fig. 12 An XML tree

$IT_4 = \{(Physics, 24), (James, 31), (Harrison, 27)\}$ with LCA (22, `prerequisites`) which is an irrelevant result since it involves two different courses. Irrelevant results affect the precision of $SLCA$.

$ELCA$ [11,41] returns ITs whose MCT does not comprise an LCA of the query keywords other than the MCT root. For this reason, IT_2 is a result for $ELCA$. IT_1 is also a result of $ELCA$ even though it is not a result of $SLCA$ since its MCT does not comprise any LCA of the query keywords other than (3, `course`). This is an improvement in $ELCA$ over $SLCA$. However, it fails to eliminate wrong results. For instance, $ELCA$ returns IT_4 as there does not exist another IT with a descendant LCA. In some cases, $ELCA$ has no way of eliminating an irrelevant answer. This weakness affects its precision.

The $VLCA$ semantics [9,16] takes into account also the labels of the nodes in the XML tree. It eliminates an IT if for some pair of annotated nodes, the path between them comprises two distinct nodes with the same label and at least one of them is an internal node in the path. $VLCA$ fails to return IT_1 because nodes 9 and 10 have the same label, `instructor`. Even though $VLCA$ is intuitive in specific cases, it is prone to missing relevant results in the general case. In addition, it is not able to eliminate irrelevant results when they do not contain duplicate labels. For instance, it fails to eliminate $IT_5 = \{(Physics, 34), (James, 41), (Harrison, 41)\}$ (whose LCA (1, `university`) is labeled as l_5 in Fig. 12) which is irrelevant as it links information from a course and a seminar through the root of the XML tree.

$CVLCA$ [16] returns an answer which is a subset of $VLCAset$ by enforcing a stricter rule. If an IT is in the $CVLCAset$: (a) the IT is in $VLCASet$, and (b) its MCT does not comprise an LCA of the query keyword instances other than the MCT root. Condition (b) amounts to forcing the IT to be part of the $ELCAset$. Therefore, an IT is returned by $CVLCA$ if and only if it is in the intersection of $VLCAset$ and $ELCAset$. Since $CVLCA$ is a subset of $VLCAset$ and $ELCAset$, it inherits the recall problems from both approaches. For instance, it fails to return IT_1 which as mentioned above does not belong to $VLCAset$.

The $MLCA$ semantics [20] requires all keyword instances in a result IT to be pairwise meaningfully related. However, the original definition of the meaningfulness relationship in [20] does not distinguish between a keyword matching the content of a node and the label of a node. Therefore, there are three ways to interpret this relationship in the XML keyword search context, which lead to three alternative definitions for $MLCA$ semantics, named here $MLCA_{label}$, $MLCA_{value}$ and $MLCA_{label+value}$. $MLCA_{label}$ and $MLCA_{label+value}$ take the labels of the nodes in the XML tree into account. According to $MLCA_{label}$, two keyword instances n_i and n_j are meaningfully related if there exists no other node n_k in T, with the same label as n_i, such that $LCA(n_j, n_k) > LCA(n_i, n_j)$ in T. In the example of Fig. 12, $MLCA_{label}$ misses IT_1 as it does not meaningfully relate nodes (6, `fname`) and (10, `lname`). Also, it fails to eliminate IT_5 as all pairs of keyword instances in IT_5 are meaningfully related. $MLCA_{value}$ is purely structural, that is, it is independent of the node labels in the XML tree. According to $MLCA_{value}$, two instances n_i and n_j of the keywords k_i and k_j, respectively, are meaningfully related if there exists no other instance n_k of k_i in T such that $LCA(n_j, n_k) > LCA(n_i, n_j)$ in T. One can see that $MLCA_{value}$ is equivalent to $pairwiseELCA$. The $pairwiseELCA$ semantics returns an IT if the LCA of any two annotated instances of two distinct keywords in it is also an ELCA of these two keywords in T. $MLCA_{value}$ fails to eliminate the irrelevant result IT_4. Finally, according to $MLCA_{label+value}$, two instances n_i and n_j of the keywords k_i and k_j, respectively, are meaningfully related if there exists no other instance n_k of k_i in T such that $label(n_k) = label(n_i)$ and $LCA(n_j, n_k) > LCA(n_i, n_j)$ in T. In other words, $MLCA_{label+value}$ defines the mean-

ingfulness relationship by imposing the conditions of both $MLCA_{label}$ and $MLCA_{value}$. Since $MLCA_{label+value} = MLCA_{label} \cup MLCA_{value}$, it inherits the precision problems of $MLCA_{label}$ and $MLCA_{value}$.

$MaxMatch$ [25] refines $SLCA$ by excluding ITs accepted by $SLCA$ when they involve "disqualified" keyword instances in the XML tree. $MaxMatch$ is a purely structural semantics since it uses structural criteria to identify disqualified nodes. The formal definition is shown in Table 3. The query answer of $MaxMatch$ is a subset of that of SLCA. One can see that $MaxMatch$ is less restrictive than $pairwiseSLCA$ which accepts an IT if the LCA of any two annotated instances of two distinct keywords in it is also an SLCA of these two keywords in the XML tree. Since, $MaxMatch$ is contained in $SLCA$, it inherits the bad recall of $SLCA$. For instance, in our running example, it misses the relevant IT_1. Moreover, it fails to eliminate IT_4 which is irrelevant since, as mentioned earlier, it groups together two distinct courses.

Schema-level SLCA [15] also refines $SLCA$ by excluding ITs accepted by $SLCA$ leveraging both structural and semantic information. It excludes an IT accepted by $SLCA$ if its root-to-LCA label path is a proper prefix of that of another IT. By definition, *Schema-level SLCA* is contained in $SLCA$ and as such it demonstrates the poor recall performance of $SLCA$, but it can even worsen it by excluding relevant ITs retained by $SLCA$. Figure 1 (and not Fig. 12) shows such an example for query $\{Physics, James, Harrison\}$ where the relevant IT whose LCA is node 3 is rejected even though it is accepted by $SLCA$.

Our approach, *XReason*, successfully returns all relevant ITs and eliminates the irrelevant ones. Results IT_1, IT_2 and IT_3 conform to top-level patterns in the $G_{\prec}$ graph, and they are retained. The irrelevant IT_4 is eliminated because the pattern P_1 of IT_1 precedes the pattern P_4 of IT_4 with respect to the $\prec_{aph}$ relation ($P_1 \prec_{aph} P_4$). The pattern P_5 of IT_5 is preceded by the pattern P_3 of IT_3 with respect to the $\prec_{pph}$ relation ($P_3 \prec_{pph} P_5$). Thus, the IT_5 is eliminated. All other irrelevant ITs are eliminated using $\prec_{pph}$.

Note that, as shown in Fig. 11, $ELCA$ contains *XReason*. To see this, let us assume that an IT I of Q on T is not in $ELCAset(Q, T)$. Then, there is an IT J in T whose LCA occurs in the MCT of I without coinciding with the LCA of I. Therefore, I and J share a common root-to-annotated-node path and $root(MCT(J)) > root(MCT(I))$. As a consequence, the pattern P_J of J precedes the pattern P_I of I with respect to $\prec_{pph}$ ($P_J \prec_{pph} P_I$), and thus, IT I (and all the ITs of P_I) is eliminated from the answer of Q on T according to *XReason*. This proves that *XReason* $\subseteq ELCA$.

The $\prec_{pph}$ relation is particularly useful for eliminating irrelevant ITs that link keyword instances through the root of the XML tree and are almost in all cases meaningless. In Sect. 5.3, we present a heuristic extension of our algorithms which eliminates patterns connecting keyword instances through the root of the XML tree in order to improve performance.

5 Algorithms

Our implementation of *XReason* comprises two components. The first one uses a stack-based algorithm to generate the query patterns and their associated ITs. The second one constructs the precedence graph $G_{\prec}$ based on the $\prec_h$, $\prec_{aph}$ and $\prec_{pph}$ relations and ranks the patterns and their respective ITs. We have maintained these processes separate in our system in order to be able to modify the semantics of query answers (pattern graph construction and pattern ranking) but also to include additional metrics in producing a ranking for the patterns, if desired. We also present in this section an improvement of the pattern generation algorithm which avoids generating patterns that are not meaningful and would be placed in low ranks based on the homomorphism relations, thereby substantially improving the performance of the system.

5.1 Pattern generation

The algorithm that extracts the query patterns is named *PatternStack* and is outlined in Algorithm 1. PatternStack takes as input the keyword query and the inverted lists of the keyword instances (XML tree nodes) for the query keywords. It returns the patterns of the query answers associated with their ITs.

PatternStack does not wait to extract patterns until after all the result ITs of the query are computed. Instead, it follows a *dynamic* approach: it incrementally computes the patterns on the fly while computing the result ITs and links the ITs to their respective patterns. A notable feature of PatternStack is that it does not require auxiliary structures for computing patterns and ITs.

Every entry in an input inverted list consists of the Dewey code of a keyword instance and the label path from the root of the XML tree to this instance. In order to reduce space consumption, the labels are numerically encoded. The inverted lists are produced with a single pass of the XML document. Patterns in PatternStack are tree structures. Every node in a pattern is associated with the set of keywords which annotate this node and its descendants in the pattern. This keyword set is encoded as a bitmap over the list of all keywords. During the pattern construction phase, PatternStack constructs patterns which do not involve all the keywords and are called partial patterns. These patterns are represented similarly to complete patterns. Partial patterns are progressively augmented into complete patterns. Partial and complete patterns are identified by ids (*pids*). Figure 13 shows (among other concepts which will be explained later) how patterns are represented by PatternStack.

Algorithm 1: PatternStack algorithm.

```
1  PatternStack(k_1, ..., k_n: keyword query, invL: inverted lists)
2      pattern[] patterns /* Array of patterns. The array
           indexes are pattern ids */
3      int[] completePatterns /* Array of complete patterns' ids
           */
4      int[][][] jointPatterns /* Mappings from pairs of patterns
           to their joint patterns */
5      int[][] parentPatterns /* Mappings from patterns to their
           parent patterns */
6      s = new Stack()
7      while currentNode = getNextNodeFromInvertedLists() do
8          while s.topNode is not ancestor of currentNode do
9              pop(s)
10         while s.topNode is not parent of currentNode do
11             push(s, ancestor of currentNode at
12                 s.topNode.depth+1, "")
13         push(s, currentNode, keyword)
14     while s is not empty do
15         pop(s)
16 pop(Stack s)
17     for tempId = s.top.patterns.next() do
18         if temp is complete then
19             s.top.removePatternId(tempId)
20             completePatterns.add(tempId)
21             patterns[tempId].addLCA(s.top.dewey())
22         else
23             childPatterns.add(extendToParent(tempId))
24     s.pop()
25     newPatternIds = constructNewPatterns(s.top.patterns,
26         childPatterns)
27     s.top.addPatternIds(newPatternIds)
28 push(Stack s, Node n, String keyword)
29     newP = new Pattern(n.labelId, flags.set(id(keyword)))
30     newPid = addToPatternsIfNotExists(newP)
31     newPatternIds = constructNewPatterns(s.top.patterns,
32         array(newPatternId))
33     s.top.unionPatternIds(newPatternIds)
34 int[] constructNewPatterns(int[] currentPatternIdsA, int[] currentPatternIdsB)
35     foreach currentPatternIdsA as idA do
36         foreach currentPatternIdsB as idB do
37             if patterns[idA].keywordFlags AND patterns[idB].keywordFlags
               == 0 then
38                 if jointPatterns[min(idA, idB), max(idA, idB)] is set then
39                     newPatterns.add(jointPatterns[min(idA,
40                         idB), max(idA, idB)])
41                 else
42                     newP = new Pattern(joinRoots(
43                         patterns[idA], patterns[idB]))
44                     newP.keywordFlags =
45                         patterns[idA].kwFlags OR
46                         patterns[idB].kwFlags
47                     newPid =
48                         addToPatternsIfNotExists(newP)
49                     newPatterns.add(newPid)
50                     jointPatterns[min(idA, idB), max(idA,
51                         idB)] = newPid
52     return newPatterns
53 int extendToParent(int childPid)
54     if parentPatterns[childPid] is set then
55         return parentPatterns[childPid]
56     else
57         childP = copyOf(patterns[childPid])
58         parentP = new Pattern(parentLabel(childP.label),
59             childP.kwFlags)
60         childP.label = tail(childP.label)
61         parentP.addChild(childP)
62         parentPid = patterns.add(parentP)
63         parentPatterns[childPid] = parentPid
```

PatternStack processes nodes from the keyword inverted lists in document order. The algorithm uses a stack to progressively construct the patterns of a query on an XML tree in a bottom-up way. Each stack entry corresponds to a node of the XML tree and is associated with the set of all the MCTs that involve its descendant keyword instances. These MCTs can be complete (i.e., they involve instances of all the query keywords) or partial (i.e., they involve instances of a proper subset of the query keywords). They are represented in the stack entry by their corresponding partial or complete patterns. For each pattern, only an id (*pid*) is stored.

In order for a node n to be pushed into a stack, the top stack node should be the parent of n. This is guaranteed by appropriate pops of non-ancestor nodes and pushes of all the ancestors of n (lines 8–12). For each new node, a partial pattern MCT is constructed (line 29). If this pattern has been constructed before, it appears in the list $patterns$ and its pid is known. Otherwise, it is added to $patterns$ and gets a new pid (line 30). Then, it is combined with all the existing patterns of the top stack node to build new patterns. The resulting new pattern set is unioned with the old pattern set of the top stack node (lines 31–33).

During a pop action, all complete patterns are removed from the top stack entry (lines 18–20). The Dewey code of the top stack node is associated with all complete patterns, since it is the root of the MCTs of the ITs corresponding to these complete patterns (line 21). The remaining (partial) patterns are extended to the parent of the top node (line 23, 53–63). Note that only the root node of a pattern MCT is associated with a path of label ids, while the rest of the nodes are associated only with their own label id (Fig. 13b). This scheme is implemented in lines 58–60. The annotation of the former root is also propagated to the new one (line 58). The top entry is popped (line 24), and its extended patterns are combined with the patterns of the parent stack entry to produce new ones (line 25). The patterns are combined through an operation called *joinRoots* (line 42) which simply merges their roots. The resulting pattern is called *jointPattern*. Figure 13 shows examples of *joinRoots* operations on partial patterns. Finally, the extended patterns together with newly combined ones are added to the parent stack entry.

Algorithm PatternStack considers any alternative way of combining partial patterns into a complete pattern only once. It stores the patterns in the $patterns$ array and keeps only their pids in the stack entries. Every time two patterns are combined to produce a new one, the pids of the combined patterns and that of the jointPattern are kept in the table $jointPatterns$ (line 4). Another table ($parentPatterns$ in line 5) associates each pattern MCT $childP$ with the pattern $parentP$ produced when $childP$ is augmented with an edge to the parent node of its root (lines 53–63). These two tables are consulted before any pattern is constructed

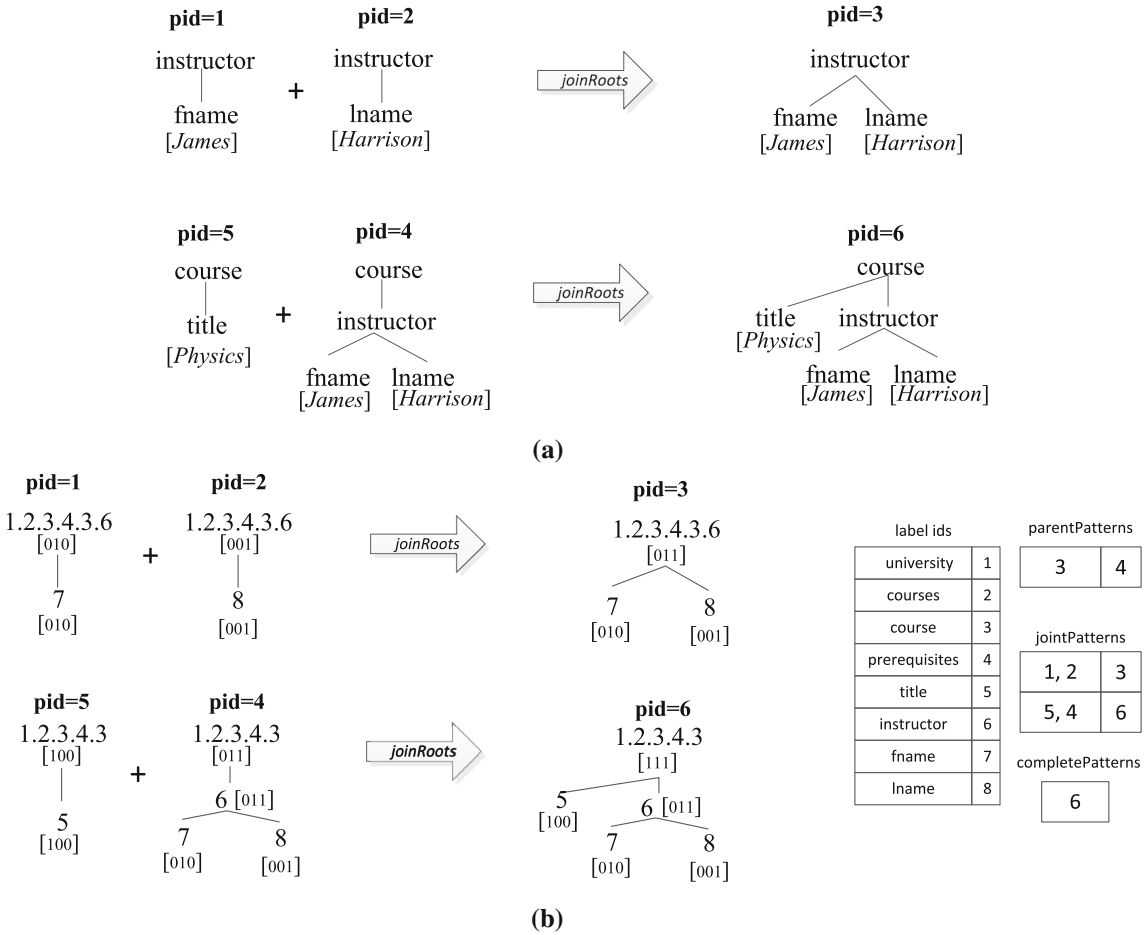


Fig. 13 PatternStack pattern encoding and combination for $Q = \{Physics, James, Harrison\}$ on the data tree of Fig. 12. **a** Partial patterns, **b** encoded partial patterns

(lines 38, 54) in order to avoid extending a pattern which has already been extended or combining two patterns that have already been combined. Function addToPatternsIfNotExists() (lines 30, 48) checks if a newly constructed pattern has been constructed before. This function compares the new pattern only with stored patterns having: (a) the same root label path and (b) the same root annotation. The comparison is performed exploiting a unique string representation of each pattern, which uses the label encodings and the annotation bitmaps of the pattern nodes. Function checkIfExistsOrAddToPatterns() returns the existing pid or the new pid assigned to the new pattern added to the array $patterns$.

The example of Fig. 13 shows how PatternStack combines and extends patterns. Patterns P1 and P2 are combined to construct P3. Pattern P3 is extended to produce P4. Pattern P4 combined with P5 produces P6. The figure shows also the encoded labels and label paths as well as the bitmap keyword annotations.

The previous discussion in this section justifies the following proposition.

Proposition 6 *The PatternStack algorithm correctly computes all the patterns of a keyword query on the set of inverted lists of an XML tree.*

Let d be the depth of the data tree and $|S|$ be the total number of nodes in the inverted lists. Each insertion of a node from the inverted lists may require at most d pops from and d pushes onto the stack. Let p be the number of partial patterns a stack entry can contain. When a node is pushed onto the stack, it may be combined with at most p patterns of the parent stack entry. This takes $O(p)$ time. When a node is popped from the stack, all its partial patterns are extended with an edge to their parent node (the new top entry in the stack). Assuming all the patterns of the popped node are partial, this takes $O(p)$. They are also combined with the partial patterns of the parent node to produce new patterns which takes $O(p^2)$. The whole process takes $O(|S|dp^2)$.

5.2 Graph construction and ranking

The second component of our system constructs the precedence graph $G_{\prec}$ and ranks the patterns. This is implemented

by algorithm *PatternGraph*. Algorithm *PatternGraph* takes as input the patterns produced by *PatternStack* and incrementally constructs $G_{\prec}$ by checking for the existence of the homomorphism relations $\prec_h$, $\prec_{aph}$ and $\prec_{pph}$ between each new pattern and the patterns in $G_{\prec}$. Then, it uses the graph to rank the patterns as described in Sect. 3.4. There are two sources of complexity in this process: (a) checking for the existence of the $\prec_h$, $\prec_{aph}$ and $\prec_{pph}$ relations between two patterns (which involves checking for the existence of the different types of homomorphisms) and (b) applying these checks to a large number of pairs of patterns.

In order to deal with (a), *PatternGraph* exploits the properties of the $\prec_h$, $\prec_{aph}$ and $\prec_{pph}$ relations as this is shown by the next four propositions.

Property 1 Let P and P' be two patterns of a query on an XML tree. If the number of root-to-leaf paths of P is greater than the number of root-to-leaf paths of P', then $P \not\prec_h P'$.

The proof of Property 1 is derived from the fact that the annotation (subset of the keywords) of an annotated node n' in P' which is mapped by a homomorphism to a node n in P should be a subset of the annotation of n. Since the numbers of annotating keywords in two patterns are equal, the number of root-to- leaf paths in P cannot exceed that of P'.

Property 2 Let P and P' be two patterns of a query on an XML tree. If the height of the MCT of P is greater than that of the MCT of P', then $P \not\prec_{aph} P'$.

Clearly, if the height of the MCT of P is greater than that of the MCT of P', the longest path in the MCT of P cannot be mapped by a path homomorphism to any path in the MCT of P', and therefore, $P \not\prec_{aph} P'$.

Property 3 Let P and P' be two patterns of a query on an XML tree. If the labels of nodes annotated by the same keyword in P and P' are not the same, then $P \not\prec_h P'$, $P \not\prec_{aph} P'$, $P' \not\prec_h P$ and $P' \not\prec_{aph} P$.

Indeed, it is easy to see that there is no homomorphism from the MCT of P to that of P' and path homomorphisms from the paths of the MCT of P to those of the MCT of P' if the labels of their nodes annotated by the same keywords are not the same.

Property 4 Let P and P' be two patterns of a query on an XML tree. $P \not\prec_{pph} P'$, if one of the following conditions does not hold: (a) the LCA depth of P' is smaller than the LCA depth of P, (b) the root-to-LCA path of P' is a prefix of the root-to-LCA path of P and (c) the maximum length MCT path in P' is longer than the minimum length MCT path in P.

The proof of Property 4 can be directly derived from the definition of path homomorphism and the $\prec_{pph}$ relation.

Based on these properties, *PatternGraph* avoids initiating in most cases the checking for the existence of homomorphisms or path homomorphisms between patterns by storing numeric information (e.g., the number of root-to-leaf paths, the height of the MCT, the LCA depth) with every pattern at construction time.

To support checking for path homomorphisms when this is needed, the MCT root-to-annotated-node paths, including the annotations, are represented as strings. Then, checking for the existence of path homomorphisms reduces to string matching starting with the annotations.

In order to address the complexity related to the large number of checks between patterns (which are needed in order to determine the existence of edges between nodes), *PatternGraph* avoids constructing some paths in the graph $G_{\prec}$ which do not affect the final ordering of the patterns. This is based on the following proposition.

Proposition 7 *Let P and P' be two nodes in the graph $G_{\prec}$. If there is an edge from P to P' in $G_{\prec}$ (that is $P \prec P'$), any edge from the ancestors of P to P' does not alter the ordering of the patterns produced by $G_{\prec}$.*

The reasoning of the previous proposition is that in order to determine the ordering $\mathcal{O}$ of the patterns, their level in the graph $G_{\prec}$ ($GLevel$) needs to be computed. The $GLevel$ value of a pattern is the maximum distance of the pattern from a source node in the graph. Therefore, transitive edges do not increase this distance and need not be added to the graph or if present, they can be removed.

Graph $G_{\prec}$ is stored in the form of an adjacency list. For each pattern (node) P, a list of pointers to the parent nodes and a list of pointers to the child nodes are maintained. We also maintain the list of source nodes and the list of sink nodes in $G_{\prec}$. When a new pattern is considered, *PatternGraph* checks the existence of homomorphisms starting with a sink node of $G_{\prec}$ and proceeds in a bottom-up way. The sink node list supports the bottom-up computation, and the source node list is used for easily detecting the minimal patterns at the end of the process.

The outline of algorithm *PatternGraph* is shown in Algorithm 2. Algorithm *PatternGraph* compares every pattern P_{new} in the input list $PList$ of patterns with the patterns P_{old} in the L_{sink} list by calling procedure **Compare** (lines 2–8). If $P_{old} \prec P_{new}$, an edge from P_{old} to P_{new} is added to $G_{\prec}$ and L_{sink} is updated if needed (lines 17–20). If $P_{new} \prec P_{old}$ or P_{new} and P_{old} are incomparable w.r.t. $\prec$, procedure **Compare** is recursively called for all parents P_0 of P_{old} (lines 22–25). If $P_0 \prec P_{new}$ and $P_{new} \prec P_{old}$, the transitive edge from P_0 to P_{old} is removed (lines 24–25). If $P_{new} \prec P_{old}$ an edge from P_{new} to P_{old} is added to $G_{\prec}$ and L_{source} is updated if needed (lines 26–29). Further optimizations based on Proposition 7 (not shown in the outline of Algorithm 2)

are implemented in *PatternGraph* to avoid adding transitive edges.

Algorithm 2: PatternGraph algorithm.

```
1  PatternGraph(PList: list of patterns)
2      foreach P in PList do
3          if G≺ is empty then
4              L_sink.add(P)
5              L_source.add(P)
6          else
7              foreach P' in L_sink do
8                  Compare(P, P')
9          if P.parents is empty then
10             L_source.add(P)
11         if P.children is empty then
12             L_sink.add(P)
13         Reset all visited flags
14 Compare(P_new, P_old)
15     if P_old.visited=false then
16         P_old.visited=true
17         if P_old ≺ P_new then
18             Add edge(P_old, P_new) to G≺
19             if P_old in L_sink then
20                 L_sink.remove(P_old)
21         else
22             foreach P_0 in P_old.parents do
23                 Compare(P_new, P_0)
24                 if P_0 ≺ P_new and P_new ≺ P_old then
25                     remove edge(P_0, P_old) from G≺
26             if P_new ≺ P_old then
27                 add edge(P_new, P_old) to G≺
28                 if P_old in L_source then
29                     L_source.remove(P_old)
```

The previous discussion in this section justifies the next proposition.

Proposition 8 *Algorithm PatternGraph correctly constructs the graph $G_{\prec}$ given a set of patterns as input.*

Let m be the number of patterns. Procedure **Compare** compares each pattern to at most m other patterns. That is, it performs $O(m^2)$ comparisons. Comparing one pattern to another for the $\prec_h$ relation can be done in linear time on the size of the pattern assuming every node is annotated by the keywords of its descendant nodes and sibling nodes are ordered based on the annotating keywords. Thus, this takes $O(dk)$, where d is the depth of the XML tree and k is the number of keywords. Comparing two patterns for the $\prec_{pph}$ relation takes $O(dk^2)$ since a node annotated by keyword k_i is allowed to map to a node which is not annotated by k_i but labeled by k_i. Finally, comparing two patterns for the $\prec_{aph}$ relation takes $O(d^2k^2)$ since, in this case, a path can even map different subpaths of another path. Therefore, the PatternGraph algorithm constructs the graph $G_{\prec}$ in $O(m^2d^2k^2)$ time. In practice, by exploiting the properties of the homomorphism relations mentioned earlier, the algorithm very efficiently avoids most of these comparisons and checks.

After the graph $G_{\prec}$ is constructed, the order $\mathcal{O}$ is extracted by using the $GLevel$, $MCTDepth$ and $MCTSize$ values of each pattern. $MCTDepth$ and $MCTSize$ are calculated and stored during the generation of the patterns. In order to calculate the $GLevel$ values, the graph $G_{\prec}$ is traversed and the $GLevel$ value for each pattern is set to the maximum $GLevel$ value of its parents incremented by one.

The filtering semantics for *XReason* depends on the number k of top levels in the graph $G_{\prec}$. The patterns (nodes) that satisfy this condition (and their ITs thereon) are obtained through a depth first traversal of $G_{\prec}$ up to level k.

5.3 An extension of PatternStack

In this section, we present an extension of *PatternStack*. This extension is based on the observation that the patterns in which the MCT root coincides with the pattern root are usually meaningless. Indeed, these are patterns that link the keyword instances through the root of the XML tree which suggests that these keyword instances are not meaningfully related. We name these patterns *root* patterns. *Root* patterns usually represent a large percentage of all the patterns. Similar remarks about the meaninglessness of *root* patterns have been made in the context of different semantics [38] in order to avoid their processing.

The semantics of *XReason* is expected to capture the meaninglessness of such patterns and eventually rank them in low ranks (in the case of ranking semantics) or exclude them from the query answer (in the case of filtering semantics). Both $\prec_{aph}$ and $\prec_{pph}$ relations (but in particular the $\prec_{pph}$ relation) are effective in pushing down in the $G_{\prec}$ graph the *root* patterns. Further, $MCTDepth$ and $MCTSize$ rank low the *root* patterns among patterns with the same $GLevel$ value.

Nevertheless, even though we do not expect to have an improvement in the effectiveness of *XReason* from the pruning of *root* patterns, terminating their construction before they are even generated substantially improves the performance of *PatternStack*. Further, invoking *PatternGraph* on a much smaller number of patterns greatly reduces the number of pattern comparisons and the execution time of that algorithm too.

It is important to note that the extended ordering $\mathcal{O}'$ of the non-*root* patterns produced by the extended *PatternStack* complies with the original ordering $\mathcal{O}$ of all the patterns produced by *PatternStack*. That is, the extended *PatternStack* does not alter the order of the remaining patterns. In order to support this claim, we first show the following proposition.

Proposition 9 *Let P_1 and P_2 be two patterns. If $P_1 \prec P_2$ and P_1 is a* root *pattern then P_2 is also a* root *pattern.*

The previous proposition can be derived from the properties of the homomorphism relations. Indeed, if P_1 is a *root* pattern and P_2 is a non-*root* pattern, then $P_1 \not\prec_{pph} P_2$, while if $P_1 \prec_h P_2$ or $P_1 \prec_{aph} P_2$ the MCT roots of P_1 and P_2 should share the same label (the unique label of the root of the XML tree), a contradiction.

Proposition 9 states that if a pattern in the $G_{\prec}$ graph is a *root* pattern, all its descendant patterns are also *root* patterns. Consider the ordering $\mathcal{O}$ of the patterns induced by the $G_{\prec}$. If we eliminate all *root* patterns and their incident edges from $G_{\prec}$, the order between the remaining non-*root* patterns will be preserved in the ordering $\mathcal{O}'$ of the patterns induced by the resulting graph $G'_{\prec}$ (which is the graph produced by the extended *PatternStack*). This is formalized by the next proposition.

Proposition 10 *Let P_1 and P_2 be two non-*root *patterns. If P_1 precedes P_2 in the original ordering $\mathcal{O}$ then P_1 also precedes P_2 in the extended ordering $\mathcal{O}'$.*

Therefore, the extended *PatternStack* algorithm can be safely used to improve the performance of *PatternStack*.

The modification of PatternStack for the implementation of the extension under discussion is confined to the pop() procedure of Algorithm 1. The extended PatternStack treats in a different way the nodes that are children of the document root. The patterns that are rooted on them are not extended to their parent (i.e., the document root) in order to construct new patterns. Thus, if the stack contains exactly two entries (i.e., the document root and one of its children), lines 23, 25 and 27 are not executed. This way, the children of the root node may contribute only complete patterns, while their partial ones are not further processed.

6 Experimental evaluation

We performed experiments to measure the efficiency and effectiveness of *XReason* as a filtering and ranking system. We compare the quality of our results to that of previous approaches.

In contrast to the IR domain [8], there is no standard benchmark to evaluate the effectiveness of keyword search on data-oriented XML [38].

We use Mondial, SIGMOD, NASA and DBLP datasets for the experiments which are obtained from the UW XML Data Repository[2]. Statistics for these datasets are depicted in Table 2. We do not distinguish between elements and attributes, and we represent attributes in these datasets as elements. For the effectiveness experiments, we use Mondial and SIGMOD datasets which are often used for this purpose in XML keyword search research [3,22]. We used NASA and DBLP datasets which are larger for the scalability experiments. NASA and DBLP datasets show different characteristics: DBLP is a large but shallow dataset, whereas NASA is a relatively smaller but deep dataset. Because of these different characteristics, we can measure the efficiency of our algorithms in a representative environment. The experiments were conducted on a 2.9 GHz Intel Core i7 machine with 3 GB memory running Ubuntu.

Table 2 Mondial, SIGMOD, DBLP and NASA dataset statistics

	Mondial	SIGMOD	DBLP	NASA
Size	1 MB	467 kB	127 MB	23 MB
No. of nodes	69,846	15,263	3,736,406	523,963
No. of distinct tags	50	12	50	70
No. of distinct label paths	119	12	145	111
Average depth	3.00	4.60	1.93	4.56
Maximum depth	5	6	5	7

We first introduce the metrics we use for the experimental evaluation; then, we present our results on effectiveness for both filtering and ranking semantics, and finally we present our efficiency experiments.

6.1 Metrics

Since the ranking approaches we consider may view a number of results as equivalent (i.e., having the same rank), we extend below the metrics that are usually used to measure the quality of ranking. In order to determine the ground truth for the effectiveness experiments, we employed five expert users who are not involved in this project which characterized the query patterns as relevant or irrelevant to the query. The relevancy of each pattern (relevant or irrelevant) was determined by the majority of the characterizations of the expert users.

Filtering experiments Since *XReason* works with patterns, if a pattern is among those with the smallest k $GLevel$ values, all candidate results that conform to it are regarded as relevant and are returned to the user. We use *precision* and *recall* to measure the effectiveness of filtering semantics. *Precision* is the ratio of the number of relevant results in the result set of the system to the total number of results returned by the system. *Recall* is the ratio of the number of relevant results in the result set of the system to the total number of relevant results.

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

Ranking experiments For the ranking experiments, we employ two metrics: Mean Average Precision (MAP) and precision@N.

MAP is the mean average precision of a set of queries with average precision of a query being the average of precision scores after each relevant result of the query is retrieved. As a ranking effectiveness metric, MAP takes the order of the results into account.

We extend MAP so that it takes into account equivalence classes of results. An equivalence class in a ranked list is a set of all the results which have the same rank. Different orderings of these results in the ranked list would affect the value of MAP [32]. For this reason, we define and compute *worst* and *best* versions for MAP. In the *worst* (resp. *best*) version, the ranked list is assumed to have the correct results ranked at the end (resp. beginning) of each equivalence class. This extension allows us to compute upper and lower bounds for the ranking metrics between which the scores of all the possible rankings lie. We denote these metrics as MAP_{worst} and MAP_{best}. We also computed the *expected* MAP value by averaging over all the queries, the average AP of the possible rankings of the results of every query. This latter metric is denoted MAP_{exp}. Clearly, $MAP_{worst} \leq MAP_{exp} \leq MAP_{best}$. When the possible rankings were too numerous to be computed exhaustively, we used sampling to compute MAP_{exp}.

In order to assess the effect of answer set size on precision in ranking experiments, we also measure precision with a cutoff point for the number N of results which is called precision@N (P@N). Similar to MAP, we consider two versions of P@N: $P@N_{worst}$ and $P@N_{best}$ and also compute the expected $P@N$ value.

6.2 Effectiveness of filtering semantics

For the filtering experiments, we compare *XReason* with $k = 1$ (that is, we consider only the patterns with the smallest $GLevel$ value) with three well-known baseline approaches: SLCA [7,12,40], ELCA [11,41] and VLCA [9,16]. We also compare with two more recent approaches, XReal [3,4] and the CR approach [38].

In order to allow the comparison of *XReason*, which returns ITs and not simply LCAs with the other approaches, we use the definitions of SLCA, ELCA and VLCA semantics in terms of ITs provided in Table 1. XReal infers promising result node types (label paths from the root) and ranks and returns the nodes that match these node types. In order to compare XReal with *XReason*, we adjusted XReal in Table 3 so that it returns ITs and we named this new approach *ITReal*. For a query Q on an XML tree T, $ITset(Q, T)$ denotes the set of ITs of the instances of Q on T. $XReal_{Nodes}$ denotes the set of nodes in T that match the node type inferred by XReal.

Table 3 Definitions of XReal semantics in terms of ITs

Approach	Definition of answer of Q on T
$ITReal$	$\{t \mid t \in ITset(Q,T), n = root(MCT(t))$, and $\exists\, n'\ \ (n' \in XReal_{Nodes}$ and $n' < n)\}$

CR does not need an adjustment as it returns subtrees similar to the ITs of *XReason*. CR can be directly compared to *XReason* because, like *XReason*, it partitions the results into patterns. It also characterizes the relevance of the patterns, not of individual results. Since CR excludes root patterns, we consider in the effectiveness experiments non-root patterns. As in [38], the factor $f(n)$ of NTC is set to $n^2/(n-1)^2$ and the threshold of NTC is set to zero.

We run 20 queries on each of the datasets shown in Table 4 (Mondial and SIGMOD). Most of the queries are chosen from previous papers: M1–M7 and S1–S8 from [26], M8–M12 and S9–S11 from [22], M13 from [23], M14–M16 from [25], and S12–S15 from [19]. Table 4 also shows the total number of patterns (results) and the number of relevant patterns (results) for each query. Precision scores of the different approaches for all the queries on the Mondial and SIGMOD datasets are shown in Figs. 14 and 15, respectively. Recall scores are only shown for the queries on Mondial in Figure 16. The recall scores for the queries on SIGMOD are all equal to 1.0 for all approaches with the exception of *XReason* on query S2 which is 0.95. Table 5 provides average precision and recall scores for all approaches on both datasets.

All approaches show good recall with CR and ITReal topping the list and *XReason* and ELCA doing almost equally well. However, in terms of precision ITReal and CR display the worst scores closely followed by ELCA and SLCA. Both ITReal and CR, as *XReason*, are also ranking systems and they can use their ranking capacity to reduce the negative effect of a large size answer set on precision. For this reason, in the next section we also measure P@N. As we can see, *XReason* significantly outperforms all the other approaches ($p < 0.05$) with the exception only of the VLCA approach on SIGMOD dataset ($p < 0.12$) in terms of precision and shows almost perfect recall on both datasets.

6.3 Effectiveness of ranking semantics

In order to evaluate the effectiveness of the ranking semantics of *XReason*, we computed the queries of Table 4 under *XReason*, ITReal and CR semantics on the datasets and we measured best and worst bounds and expected values for MAP and P@N.

Table 6 shows the MAP scores. *XReason* outperforms CR and largely outperforms ITReal. Since *XReason* ranks the

Table 4 Queries used in the experiments

Dataset	Query ID	Keywords	No. of relevant patterns	No. of patterns	No. of relevant results	No. of results
Mondial	M1	*torneaelv, country, province*	1	1	1	1
	M2	*roman, catholic, percentage, united, states*	1	8	2	36
	M3	*population, 87, albania, city*	1	7	12	504
	M4	*organization, name, members*	2	2	10,111	10,111
	M5	*country, government, republic*	2	17	115	3579
	M6	*country, ethnicgroups, german*	2	14	19	3066
	M7	*city, washington, province*	3	37	6	149,352
	M8	*france, territory*	1	1	3	3
	M9	*lake, located*	2	3	97	124
	M10	*singapore, country*	3	4	4	6
	M11	*religions, christian, muslim*	2	6	86	111
	M12	*province, houston, dallas*	1	6	1	294
	M13	*belarus, population*	2	2	4	4
	M14	*united, states, birmingham, population*	2	12	6	3198
	M15	*ethnicgroups, chinese, indian, capital*	2	6	16	183
	M16	*country, muslim*	2	8	101	2209
	M17	*international, monetary, fund, established*	1	1	1	1
	M18	*government, democracy, muslim*	2	2	17	17
	M19	*jewish, percentage*	2	9	17	109
	M20	*japan, tokyo, population*	2	10	4	244
SIGMOD	S1	*author, position, 01, harry, article*	1	217	2	74,392,500
	S2	*jim, gray, title, initpage, endpage*	2	67	19	8,976,784
	S3	*initpage, 3, endpage, 7*	1	65	4	56,210
	S4	*author, nicolas*	1	3	2	197
	S5	*article, title, author*	2	10	3738	11,309,368
	S6	*initpage, 7, article, endpage*	2	55	20	1,940,838
	S7	*volume, 11, article*	1	9	12	934
	S8	*asuman, pinar, article*	1	5	4	135
	S9	*directions, database, research*	1	5	1	366
	S10	*jennifer, widom, jeffrey, d, ullman*	1	47	2	294
	S11	*relational, model, author, date*	1	10	1	238
	S12	*karen, title*	1	2	2	51
	S13	*anthony, data*	1	2	2	55
	S14	*article, data, john*	1	5	3	7801
	S15	*database, volume, number*	1	3	347	374
	S16	*divesh, srivastava, database*	1	5	2	174
	S17	*michael, stonebraker, postgres*	1	6	3	23
	S18	*database, systems, security*	1	5	1	417
	S19	*christos, faloutsos, signature, files*	1	2	1	2
	S20	*efficient, maintenance, materialized, views, subrahmanian*	1	24	1	36

correct results almost always higher than the incorrect ones, it has almost perfect MAP scores.

Best and worst P@N scores are shown in Figs. 17 and 18. For a given query and a given approach, best and worst scores are shown on the same column with worst scores superimposing best scores, i.e., if the scores are the same, only worst scores are visible. $N = 10$ for all datasets because most of the queries have few correct results. The average $P@10_{exp}$

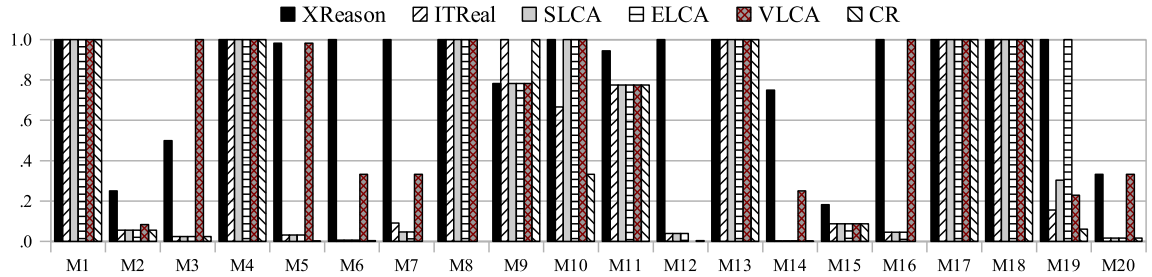


Fig. 14 Precision scores for the queries of Table 4 on the Mondial dataset

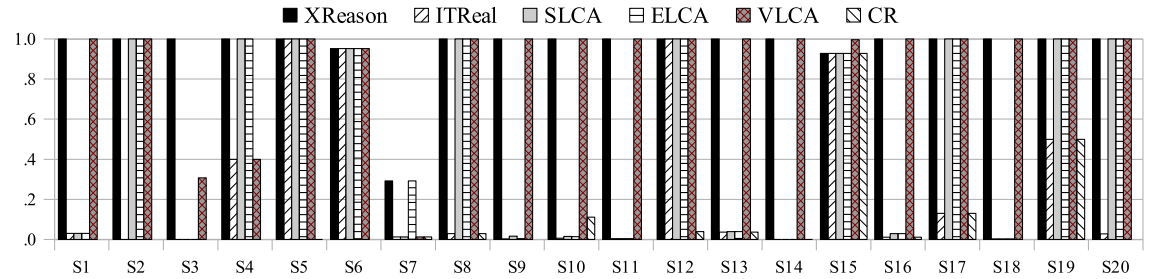


Fig. 15 Precision scores for the queries of Table 4 on the SIGMOD dataset

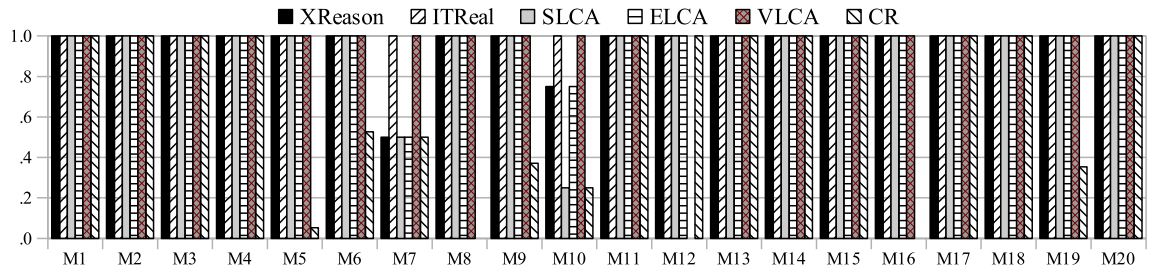


Fig. 16 Recall scores for the queries of Table 4 on the Mondial dataset

Table 5 Average precision and recall scores for the queries of Table 4

Dataset	Metric	*XReason*	*ITReal*	*SLCA*	*ELCA*	*VLCA*	*CR*
Mondial	*Avg. Prec.*	0.84	0.45	0.46	0.50	0.66	0.47
	Avg. Rec.	0.96	1.00	0.94	0.96	0.95	1.00
SIGMOD	*Avg. Prec.*	0.96	0.25	0.50	0.52	0.88	0.10
	Avg. Rec.	1.00	1.00	1.00	1.00	1.00	1.00

Table 6 Best and worst MAP scores for the queries of Table 4

Dataset	Semantics	MAP_{worst}	MAP_{best}	MAP_{exp}
Mondial	*XReason*	0.97	0.97	0.97
	ITReal	0.48	0.76	0.51
	CR	0.71	0.71	0.71
SIGMOD	*XReason*	1.00	1.00	1.00
	ITReal	0.34	0.63	0.37
	CR	0.90	0.90	0.90

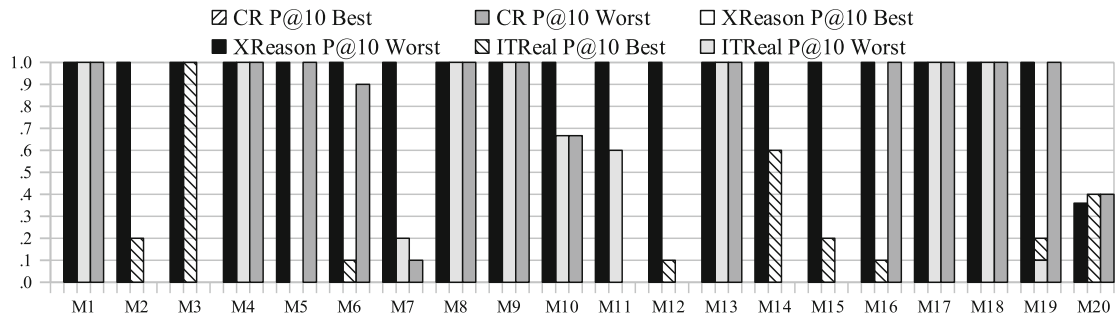


Fig. 17 Best and worst P@10 scores for the queries of Table 4 on Mondial dataset

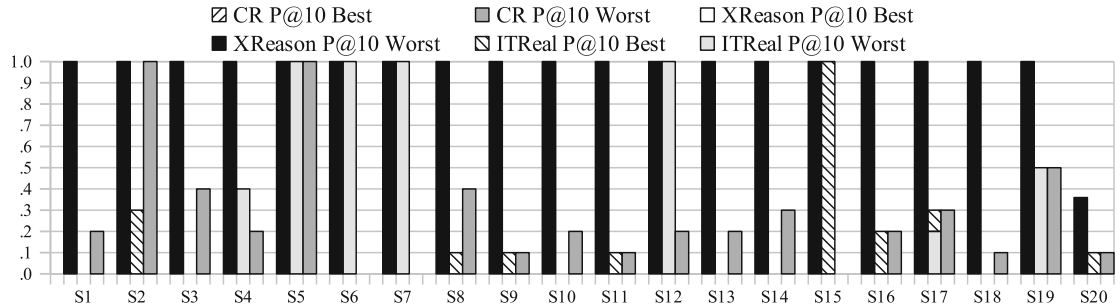


Fig. 18 Best and worst P@10 scores for the queries of Table 4 on SIGMOD dataset

Table 7 Average $P@10_{exp}$ scores for the queries of Table 4

Dataset	*XReason*	*ITReal*	*CR*
Mondial	0.97	0.44	0.60
SIGMOD	1.00	0.27	0.28

scores are displayed in Table 7. For ITReal, limiting the result set size did not have a significant effect on the precision for most of the queries, which means that some incorrect results are ranked high in the result list. *XReason* has perfect P@10 scores in almost all cases and obtains statistically significant differences with respect to the ranking comparison systems ($p < 0.01$).

6.4 Efficiency

In order to evaluate the efficiency of the algorithms: (a) we compared the computation time of our original algorithm to a naïve algorithm (b) we run scalability experiments for the original and the extended algorithms and (c) we compared the original versus the extended algorithm.

The naïve algorithm for implementing the *XReason* semantics generates all the ITs of the query using the inverted lists of the keywords and iterates over them to extract the patterns. Then, it checks for the existence of homomorphisms between the pairs of patterns in a straightforward way and computes the query results. Figure 19 shows the computation times of the odd numbered queries of Table 4 for our original algorithm and the naïve algorithm on the Mondial and SIGMOD datasets. We only report on half of the queries to save space. Note that the y-axis is in logarithmic scale. Times exceeding 10,000 s for the naïve algorithm are shown with arrows. Our algorithm is at least two orders of magnitude faster than the naïve algorithm in most of the cases. The average computation times for our algorithm over 20 queries on Mondial and SIGMOD datasets are 0.50 and 0.51 s, respectively. This performance is acceptable for real-time search systems even without additional optimizations.

For our scalability experiments, we used DBLP and NASA datasets. For the original algorithm, we measured the computation time in relation with the output size (the number of ITs). The original algorithm depends heavily on the output size. For the experiments, we use the most frequent four, five and six keywords to form three different queries from the randomly chosen keywords shown in Table 8. Table 8 lists the keywords in descending order of their frequencies. For each query, we truncated the keyword inverted lists at 20, 40, 60 and 80 % of their length. The computation times are presented in Fig. 20. Obviously, the higher the number of keywords, the higher the number of ITs returned. As we can see in Fig. 20, the original algorithm scales well even though the number of ITs increases very fast with the percentage of the inverted lists being used.

For the scalability experiments on the extended algorithm, we measured the computation time in relation to the input size (the total number of keyword instances in the XML tree of all the query keywords). The extended algorithm depends

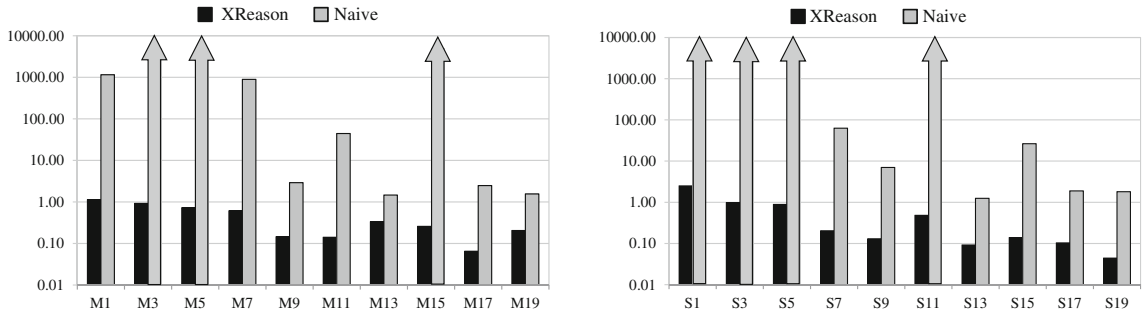


Fig. 19 *XReason* execution times (in s) for the queries of Table 4. **a** Mondial, **b** SIGMOD

Table 8 Queries used in scalability experiments

Algorithm	Dataset	Keywords
Original algorithm	DBLP	*osawa, sculptured, cricket, marriages, erhebung, ilpo*
	NASA	*iccd, brightnesses, colloquium, hendry, perugia, attribute*
Extended algorithm	DBLP	*srinivas, elias, masao, divyakant, sums, lectures*
	NASA	*medium, dr, seen, heii, oxygen, comparing*

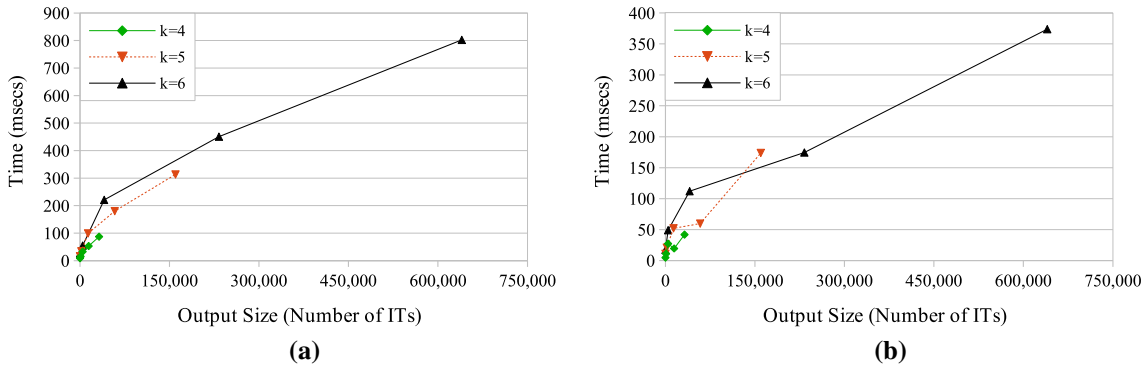


Fig. 20 Computation time versus output size for the original algorithm using queries with 4, 5 and 6 keywords. **a** NASA dataset, **b** DBLP dataset

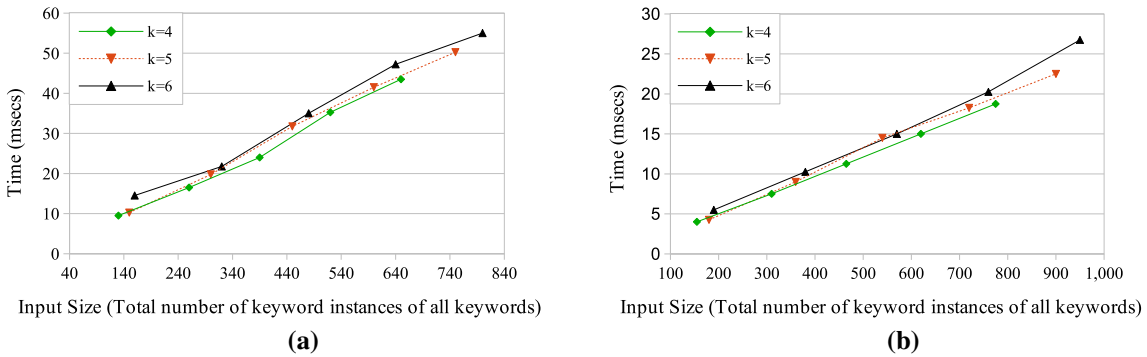


Fig. 21 Computation time versus input size for the extended algorithm using queries with 4, 5 and 6 keywords. **a** NASA dataset, **b** DBLP dataset

mainly on the input size since it eliminates non-*root* patterns and returns a restricted number of ITs. We selected queries from Table 8 and truncated keyword inverted lists as we did in the previous scalability experiment. We show the measured computation times in Fig. 21. As shown in the figure, the extended algorithm scales smoothly (it is almost linear). This is due to the fact that the extended algorithm prunes partial patterns early on in the computation before they become complete patterns as long as they are rooted at the root of the XML tree.

For our experiments on comparing the original and the extended algorithm, we run five queries on the DBLP

Table 9 Queries on the DBLP dataset used compare the performance of the original versus the extended algorithm

Query ID	Keywords
Q1	*xml, keyword, search*
Q2	*query, analysis*
Q3	*sequence, alignment*
Q4	*collaborative, filtering, recommendation*
Q5	*dynamic, incremental, clustering*

dataset to measure the number of ITs, the number of generated patterns and the computation time of the two algorithms. The queries are shown in Table 9. We selected real-world queries in order to guarantee that they return a reasonable number of non-*root* patterns and therefore to highlight the differences in the performance of two algorithms. The results are shown in Fig. 22. In Fig. 22a, b, we show the number of ITs and the number of generated patterns for each query, respectively. The y-axes are in logarithmic scale. The number of *root* patterns is significantly greater than the number of non-*root* patterns for all queries. The same remark applies to the number of *root* and non-*root* ITs. In Fig. 22c, we show the computation time of both algorithms. As we can see in this figure, the extended algorithm significantly outperforms the original algorithm. These results show that the extended algorithm is a good substitute of the original one in real-world applications.

7 Related work

Keyword search on semi-structured databases is a complicated task. Semi-structured databases such as XML usually do not follow a strict schema, so combining data from different parts of the database is more challenging than in fully structured databases [15]. Several papers elaborate on filtering and ranking semantics for keyword search on XML data. The results are usually modeled as LCAs of the keyword matches or subtrees of the XML tree which contains a query match. Most of the filtering semantics are based exclusively on the structural properties of the results and only few of them take into account the semantic information (that is the labels of the nodes). The SLCA [36,40], ELCA [11], XSearch [9], VLCA [16] and MLCA [20] and their properties are extensively reviewed in Sect. 4. In [39], tree pattern queries are extracted from the structural summary and those which present a meaningful result are used. MaxMatch [25] groups SLCA nodes and eliminate some irrelevant keyword matches under these subtrees by considering additional rules. Consistency and monotonicity concepts are also introduced in MaxMatch as an axiomatic framework for evaluation of keyword search semantics. Kong et al. [14] improve over MaxMatch by working over all LCAs instead of only SLCAs and address MaxMatch's false positive and redundancy issues.

Ranking semantics for answering XML keyword queries return a ranked list of results with respect to their relevance to the user query. XRank [11] uses a variation of PageRank algorithm to rank the results. XSearch [9] ranks the results using a tf-idf function [33] adapted to the tree structure of XML documents. XReal [3] introduces a similarity function to rank nodes with respect to their similarity to the query. Termehchy and Winslett [38] exploit mutual information to calculate coherency ranking measures for ranking the query answer. Nguyen and Cao [30] use mutual information to compare results and to define a dominance relationship between results for ranking. SAIL [17] introduces the concept of minimal-cost trees and identifies the top-k answers by using link analysis and keyword-pair relevancy.

Some works focus on developing efficient algorithms for XML keyword search semantics. Algorithms for finding SLCAs and ELCAs for a keyword query are presented in [40–43]. Hristidis et al. [12] develop efficient algorithms for finding a compact representation of the result subtrees. In [10], a multi-stack algorithm to return a size-ranked result list to a keyword query is presented. Chen and Papakon-

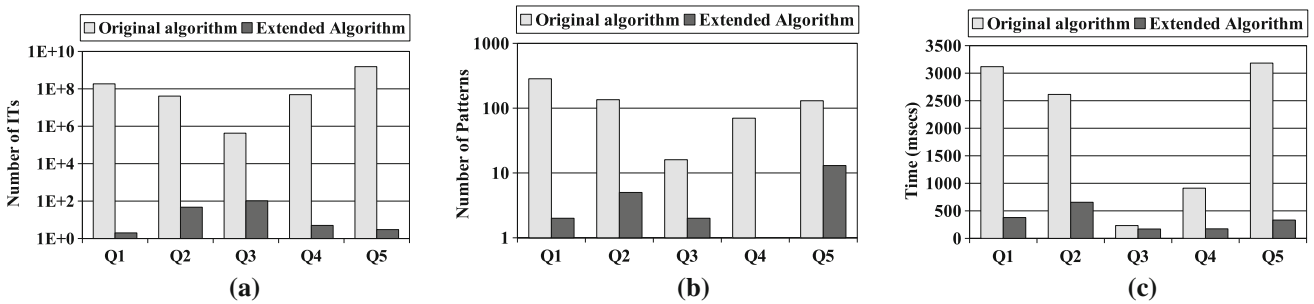


Fig. 22 **a** Number of ITs, **b** number of generated patterns and **c** the computation time of the original and the extended algorithm on the DBLP dataset using the queries of Table 9

stantinou [7] introduce algorithms to support top-k SLCA and ELCA calculation.

Additionally, different problems within the context of XML keyword search have been addressed in some studies. XReal [3,4] and XBridge [18] propose approaches to find the user intented result type. XReal [3,4] uses a variation of tf-idf for finding the candidate result type of a query. XBridge [18] uses a scoring measure for the types as well, but it also takes into account the structure of the results while scoring the type. XSeek [23] utilizes entity, attribute or connection nodes to decide upon the nodes to be returned in the results. XMean [22] and Liu and Chen [26] address the problem of clustering XML keyword search results. XMean [22] uses patterns of the results for defining clusters for the results. In order to facilitate browsing the results, a relaxation graph for the patterns are created. Liu and Chen [26] built on XSeek [23] to detect the nodes to be included in the results and the results are clustered by using the types of keyword instance nodes (i.e., entity or attribute). Context-sensitive keyword search on XML is addressed in [6]. The context is defined in the form of a path in the XML tree, and the results are ranked by taking into account the specified context. Materialized views for supporting the evaluation of XML keyword queries have been proposed in [24,35]. Keyword query refinement and/or keyword suggestion techniques in the XML keyword search context are studied in [28,31]. Most of these problems have been summarized in [27].

Keyword search has also been addressed in structured databases [5,21,29]. As in the semi-structured databases, keyword search techniques applied on the Web cannot be applied directly on relational databases. This is due to the fact that the information in relational databases are spread over multiple tables and the structure of the data should also be utilized during the keyword search [29]. Relational databases and the query answers are usually modeled as graphs [5,13], and some information retrieval techniques have been adapted for assigning semantics to keyword queries [29].

A preliminary version of part of the work presented in this paper was presented in [1]. With respect to [1], the present work introduces a new homomorphism relation to define the *XReason* semantics, presents new propositions about homomorphisms, relations and algorithm correctness, and compares *XReason* to all previous filtering semantics. Further, it introduces a new algorithm to generate the pattern graph, a new heuristic extension of *PatternStack* that prunes the pattern generation while improving performance and new experimental results to support the effectiveness of *XReason* and the efficiency of our algorithms.

8 Conclusion

We have proposed *XReason*, a novel approach for providing ranking and filtering semantics to keyword queries on XML data which is based on reasoning with patterns. The patterns record the structural and semantic characteristics of the query matches. In order to reason with patterns, we introduced homomorphisms between patterns which are leveraged to define homomorphism relations on patterns. Our approach benefits from a global view of the query matches and avoids the pitfalls of previous semantics which rely on comparing query matches locally in the XML tree or rank them based simply on a scoring function. By reasoning with patterns whose number is typically very small compared to the number of query matches, we make this global and multi-feature comparison feasible. We designed an efficient stack-based algorithm to implement *XReason*, and we also devised a heuristic extension to improve its performance. Contrary to most previous algorithms, ours works with the keyword inverted lists and does not require any auxiliary data structures and preprocessing of the data. Our experimental studies over several real datasets show that *XReason* outperforms previous approaches in precision, precision@N, recall and mean average precision. Further, they showed that our algorithms are fast and scale smoothly, and therefore, our approach is computationally feasible and can be applied in practice.

In the present work, we did not exploit techniques based on statistical information like tf-idf and PageRank which are extensively used in information retrieval. This is an orthogonal issue. As a future work, we are planning to study how our approach can be combined with these techniques in application areas where this combination can be beneficial.

References

1. Aksoy, C., Dimitriou, A., Theodoratos, D., Wu, X.: XReason: a semantic approach that reasons with patterns to answer XML keyword queries. In: DASFAA, pp. 299–314 (2013)
2. Baeza-Yates, R.A., Ribeiro-Neto, B.A.: Modern Information Retrieval. ACM Press/Addison-Wesley, New York (1999)
3. Bao, Z., Ling, T.W., Chen, B., Lu. J.: Effective XML keyword search with relevance oriented ranking. In: ICDE, pp. 517–528 (2009)
4. Bao, Z., Lu, J., Ling, T.W., Chen, B.: Towards an effective XML keyword search. IEEE Trans. Knowl. Data Eng. **22**(8), 1077–1092 (2010)
5. Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword searching and browsing in databases using banks. In: ICDE, pp. 431–440 (2002)
6. Botev, C., Shanmugasundaram, J.: Context-sensitive keyword search and ranking for XML. In: WebDB, pp. 115–120 (2005)
7. Chen, L.J., Papakonstantinou, Y.: Supporting top-K keyword search in XML databases. In: ICDE, pp. 689–700 (2010)

8. Clough, P., Sanderson, M.: Evaluating the performance of information retrieval systems using test collections. Inf. Res. **18**(2) (2013). http://InformationR.net/ir/18-2/paper582.html
9. Cohen, S., Mamou, J., Kanza, Y., Sagiv, Y.: XSEarch: a semantic search engine for XML. In: VLDB, pp. 45–56 (2003)
10. Dimitriou, A., Theodoratos, D.: Efficient keyword search on large tree structured datasets. In: KEYS, pp. 63–74 (2012)
11. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: ranked keyword search over XML documents. In: SIGMOD, pp. 16–27 (2003)
12. Hristidis, V., Koudas, N., Papakonstantinou, Y., Srivastava, D.: Keyword proximity search in XML trees. IEEE Trans. Knowl. Data Eng. **18**(4), 525–539 (2006)
13. Hristidis, V., Papakonstantinou, Y.: Discover: keyword search in relational databases. In: VLDB, pp. 670–681 (2002)
14. Kong, L., Gilleron, R., Mostrare, A.L.: Retrieving meaningful relaxed tightest fragments for XML keyword search. In: EDBT, pp. 815–826 (2009)
15. Lee, K.-H., Whang, K.-Y., Han, W.-S., Kim, M.-S.: Structural consistency: enabling XML keyword search to eliminate spurious results consistently. VLDB J. **19**(4), 503–529 (2010)
16. Li, G., Feng, J., Wang, J., Zhou, L.: Effective keyword search for valuable LCAs over XML documents. In: CIKM, pp. 31–40 (2007)
17. Li, G., Li, C., Feng, J., Zhou, L.: SAIL: structure-aware indexing for effective and progressive top-k keyword search over XML documents. Inf. Sci. **179**(21), 3745–3762 (2009)
18. Li, J., Liu, C., Zhou, R., Wang, W.: Suggestion of promising result types for XML keyword search. In: EDBT, pp. 561–572 (2010)
19. Li, J., Wang, J.: XQSuggest: an interactive XML keyword search system. In: DEXA, pp. 340–347 (2009)
20. Li, Y., Yu, C., Jagadish, H.V.: Schema-free XQuery. In VLDB, pp. 72–83 (2004)
21. Liu, F., Yu, C., Meng, W., Chowdhury, A.: Effective keyword search in relational databases. In: SIGMOD, pp. 563–574 (2006)
22. Liu, X., Wan, C., Chen, L.: Returning clustered results for keyword search on XML documents. IEEE Trans. Knowl. Data Eng. **23**(12), 1811–1825 (2011)
23. Liu, Z., Chen, Y.: Identifying meaningful return information for XML keyword search. In: SIGMOD, pp. 329–340 (2007)
24. Liu, Z., Chen, Y.: Answering keyword queries on XML using materialized views. In: ICDE, pp. 1501–1503 (2008)
25. Liu, Z., Chen, Y.: Reasoning and identifying relevant matches for XML keyword search. PVLDB **1**(1), 921–932 (2008)
26. Liu, Z., Chen, Y.: Return specification inference and result clustering for keyword search on XML. ACM Trans. Database Syst. **35**(2), 10:1–10:47 (2010)
27. Liu, Z., Chen, Y.: Processing keyword search on XML: a survey. World Wide Web **14**(5–6), 671–707 (2011)
28. Lu, Y., Wang, W., Li, J., Liu, C.: XClean: providing valid spelling suggestions for XML keyword queries. In: ICDE, pp. 661–672 (2011)
29. Luo, Y., Lin, X., Wang, W., Zhou, X.: Spark: top-k keyword query in relational databases. In: SIGMOD, pp. 115–126 (2007)
30. Nguyen, K., Cao, J.: Top-k answers for XML keyword queries. World Wide Web **15**(5–6), 485–515 (2012)
31. Pu, K.Q., Yu, X.: Keyword query cleaning. PVLDB **1**(1), 909–920 (2008)
32. Raghavan, V., Bollmann, P., Jung, G.S.: A critical investigation of recall and precision as measures of retrieval system performance. ACM Trans. Inf. Syst. **7**(3), 205–229 (1989)
33. Salton, G., Buckley, C.: Term-weighting approaches in automatic text retrieval. Inf. Process. Manag. **24**(5), 513–523 (1988)
34. Schmidt, A., Kersten, M., Windhouwer, M.: Querying XML documents made easy: nearest concept queries. In: ICDE, pp. 321–329 (2001)
35. Shao, F., Guo, L., Botev, C., Bhaskar, A., Chettiar, M., Yang, F., Shanmugasundaram, J.: Efficient keyword search over virtual XML views. VLDB J. **18**(2), 543–570 (2009)
36. Sun, C., Chan, C.Y., Goenka, A.K.: Multiway SLCA-based keyword search in XML data. In: WWW, pp. 1043–1052 (2007)
37. Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C.: Storing and querying ordered XML using a relational database system. In: SIGMOD, pp. 204–215 (2002)
38. Termehchy, A., Winslett, M.: Using structural information in XML keyword search effectively. ACM Trans. Database Syst. **36**(1), 4 (2011)
39. Theodoratos, D., Wu, X.: An original semantics to keyword queries for XML using structural patterns. In: DASFAA, pp. 727–739 (2007)
40. Xu, Y., Papakonstantinou, Y.: Efficient keyword search for smallest LCAs in XML databases. In: SIGMOD, pp. 537–538 (2005)
41. Xu, Y., Papakonstantinou, Y.: Efficient LCA based keyword search in XML data. In: EDBT, pp. 535–546 (2008)
42. Zhou, J., Bao, Z., Wang, W., Ling, T.W., Chen, Z., Lin, X., Guo, J.: Fast SLCA and ELCA computation for XML keyword queries based on set intersection. In: ICDE, pp. 905–916 (2012)
43. Zhou, R., Liu, C., Li, J.: Fast ELCA computation for keyword queries on XML data. In: EDBT, pp. 549–560 (2010)