Object Semantics for XML Keyword Search

Thuy Ngoc Le #1, Tok Wang Ling #2, H. V. Jagadish *3, Jiaheng Lu †4

#National University of Singapore, *University of Michigan, †Renmin University of China
1,2{ltngoc,lingtw}@comp.nus.edu.sg, 3jag@umich.edu, 4jiahenglu@ruc.edu.cn

Abstract. It is well known that some XML elements correspond to objects (in the sense of object-orientation) and others do not. The question we consider in this paper is what benefits we can derive from paying attention to such object semantics, particularly for the problem of keyword queries. Keyword queries against XML data have been studied extensively in recent years, with several lowest-common-ancestor based schemes proposed for this purpose, including SLCA, MLCA, VLCA, and ELCA. It can be seen that identifying objects can help these techniques return more meaningful answers than just the LCA node (or subtree) by returning objects instead of nodes. It is more interesting to see that object semantics can also be used to benefit the search itself. For this purpose, we introduce a novel Nearest Common Object Node semantics (NCON), which includes not just common object ancestors but also common object descendants. We have developed `XRich`, a system for our NCON-based approach, and used it in our extensive experimental evaluation. The experimental results show that our proposed approach outperforms the state-of-the-art approaches in terms of both effectiveness and efficiency.

1 INTRODUCTION

XML has become a widely accepted standard for data storage and data exchange in many applications, such as electronic business[1], science[2], and text databases[3]. In addition, keyword search provides a simple and user-friendly query interface to access XML data in most applications. Therefore, keyword search on data-centric XML documents has attracted great interest. One of the most successful approaches to XML keyword search is the LCA semantics [5], which was inspired by the hierarchical structure of XML. Following this, many extensions of the LCA semantics such as SLCA [20], MLCA [13], ELCA [22] and VLCA [10] have been proposed to improve the effectiveness of the search.

1.1 Limitations with the LCA semantics

While the LCA semantics and its variants work well for many types of XML documents, unfortunately, in several scenarios, they still suffer from two limitations: they may return *meaningless* answers and *incomplete* sets of answers. Meaningless answers are returned when the LCA node (or its variants) just simply matches query keywords

1 http://www.ebxml.org
2 http://www.biodas.org/documents/spec-1.53.html
3 http://www-connex.lip6.fr/∼denoyer/wikipediaXML/

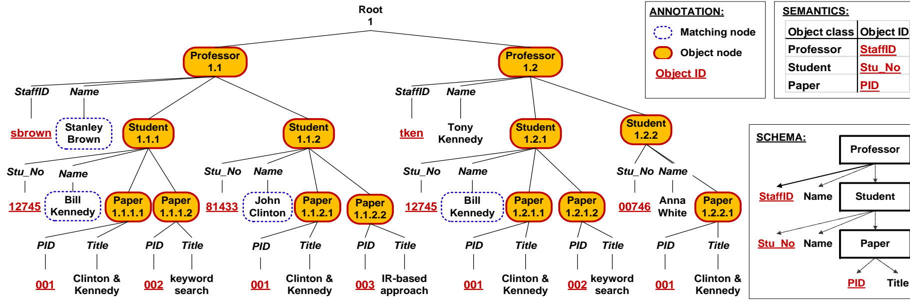


Fig. 1: An XML document with the corresponding schema and the discovered semantics

and does not provide any other additional information. More importantly, LCA-based approaches return incomplete sets of answers because they only search up from the matching nodes, i.e., the nodes containing keywords, for common ancestors, and never search down to find common information appearing as descendants. From now on, we use the term *common descendant* to refer to such common information. Example of these drawbacks can be seen in the context of the XML data tree in Fig. 1.

EXAMPLE 1 (Meaninglessness) *For query {*`Stanley, Brown`*}, an answer such as the value node* `Stanley Brown` *(in the left most) is meaningless since it does not provide any additional information about* `Stanley Brown`*. A meaningful answer should contain additional information about keywords, i.e., it should be the subtree rooted at node* `Professor(1.1)`*.*

EXAMPLE 2 (Incompleteness) *For query {*`Bill, John`*}, the keywords match two students:* `Student(1.1.1)` *and* `Student(1.1.2)` *respectively. Their common ancestor* `Professor(1.1)` *is an answer returned by LCA-based approaches. However, this is not complete. Object* <Paper:001>[4]*, which is represented by groups of nodes started at* `Paper(1.1.1.1)` *and* `Paper(1.1.2.1)`*, should also be returned as an answer. This paper is a common descendant of these student nodes. Intuitively, these students are not only supervised by the same professor* <Stanley:Brown>*, but also co-authors of the same paper* <Paper:001>*.*

The problem of incompleteness happens when the relationship between object classes is many-to-many ($m : n$). Then, the child object is duplicated each time it occurs in the relationship, and two nodes may have the same object as the child. In practice, $m : n$ relationships occur in many real XML datasets, including IMDb[5] and NBA[6] (used in experiments of prevalent XML research works such as [17, 16]). In IMDb, an actor or actress can play in many movies, and a company can produce several movies. In NBA, a player can play for several teams in different years. Moreover, due to the flexibility and exchangeability of XML, many relational datasets with $m : n$ relationships can be transformed to XML [8, 4] with duplication. Thus, it is very likely that LCA-based approaches will return an incomplete answer set for such databases.

---

[4] <Paper:001> denotes an object which belongs to object class `Paper` and has OID `001`.

[5] http://www.imdb.com/interfaces

[6] http://www.nba.com

1.2 Our novel semantics based on object

To address limitations of the LCA semantics, we propose to use the concept of *object* in XML keyword search. In XML data, an object may be represented as different object *instances*, each of which corresponds to a group of nodes, rooted at a tag indicating object class, followed by a set of attributes and their associated values. We refer to this root node as *object node* and the others as *non-object nodes*. For example, object `<Paper:001>` has four instances starting from four object nodes `Paper(1.1.1.1)`,`Paper(1.1.2.1)`,`Paper(1.2.1.1)` and `Paper(1.2.2.1)` respectively. Other nodes such as `PID, 001, Title, Clinton& Kennedy` are non-object nodes. An object is identified by its *object class* and *OID (object identifier)*. Thus, *two instances represent the same object if they have the same object class and the same OID*. This is all the object orientation we rely upon. Using just this much, we show in this paper the benefits that accrue to keyword search.

Based on these object orientation concepts, we introduce a novel semantics, called *Nearest Common Object Node* (NCON) for XML keyword search. The NCON semantics has two key features. First, an NCON must be an *object node* rather than an arbitrary node. This reduces the number of meaningless answers. Second, an NCON can be either an LCOA *(lowest common object ancestor)* or an HCOD (*highest common object descendant*). Although LCOA is similar to LCA [5, 10, 20], the important difference is that an LCOA must be an object node. An HCOD (1) is a common object descendant of a set of keywords, and (2) has no ancestor that is also a common object descendant of that set of keywords. The second feature includes *common descendants* into the answer set. Let us revisit the motivating examples introduced above and see how our proposed NCON semantics helps.

EXAMPLE 3 (Example 1 Reprise) *LCOA `Professor(1.1)` is the object node of the non-object node `Stanley Brown`. Returning the former is meaningful, whereas returning the latter is meaningless. LCOA semantics will return the object node, rather than the non-object node.*

Several works such as XSeek [14], XReal[1], and [19] have attempted to solve the problem of meaningless answers by identifying entity (object), and they can obtain more meaningful answers in several cases. However, these works do not use OIDs as ours, and therefore do not always distinguish an object from an aggregation node, a composite attribute, and a multi-valued attribute.

EXAMPLE 4 (Example 2 Reprise) *We discover that `Paper(1.1.1.1)` and `Paper (1.1.2.1)` refer to the same object: paper <`PID:001`>, because they belong to the same object class `Paper` and have the same OID value `001`. HCOD will find this paper and return it as an answer. In contrast, LCA based approaches cannot detect this common paper because it appears as a descendant, not as an ancestor.*

For an XML document with ID/IDREF, graph-based approaches such as [2, 11, 7] can provide common descendants. However, to maintain the tree structure, XML designers may duplicate information instead of using ID/IDREF. Moreover, those graph-based approaches can find common descendants only if XML documents contain ID/IDREF. Otherwise, those graph-based approaches do not recognize instances of the same object. Therefore, they cannot find common descendants either. To the best of our knowledge, only [9] can detect such instances and find common descendants. Nevertheless, this work transfers an XML document to a graph which is similar to relational

database, and follows Steiner tree semantics. Thus, it suffers from the inefficiency and may return meaningless answers because matching nodes may not be (or weakly) related (will be shown in Section 5).

A final answer obtained by LCA-based approaches includes two parts: an LCA node and a presentation of the answer, e.g., a subtree or a path. Arguably, the presentation as a subtree may contain common descendants. However, LCA-based approaches do not explicitly identify them and it may be hard to identify them because this presentation contains a great deal of irrelevant information. In contrast, our NCON semantics can and does clearly identify both common ancestors and common descendants.

1.3 Our approach and contributions

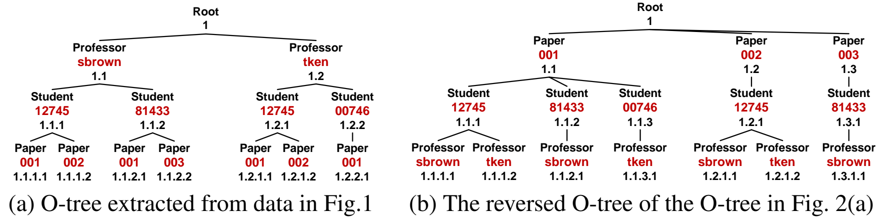
Like existing LCA-based approaches such as [20, 22, 10, 13], we work with *data-centric* XML documents *without ID/IDREF*, in which objects may be duplicated as different instances due to $m : n$ relationships. We follow the *NCON semantics* so that both common ancestors and common descendants can be answers. Finding common descendants is much more challenging than finding common ancestors. Given a set of matching nodes, unlike a common ancestor which appears as only one node, a common descendant may appear as many different nodes. Therefore, it requires more complex techniques of indexing and searching to find common descendants.

We propose a new search strategy which uses XML object tree *(O-tree)* rather than the whole XML document for the search. An O-tree is extracted from an XML data tree by keeping only object nodes and associating all non-object nodes (e.g., attributes and values) to the corresponding object nodes. This helps reduce the number of meaningless answers because only object nodes are returned. Moreover, this reduces the search space greatly since the number of nodes in O-tree is much smaller than those in the whole XML document (due to not counting non-object nodes). To find common descendants, we use a *reversed O-tree*, whose paths from the root to leafs are reversed from those of the given O-tree. Then, HCODs of the given O-tree can be found as LCOAs of the reversed O-tree. Fig. 2(b) shows the reversed O-tree w.r.t. the O-tree in Fig. 2(a).

We do not use ID/IDREF to connect instances of the same object because if we do so, XML data will be modeled as an XML graph instead of an XML tree. Searching over graph-structured data has been known to be equivalent to the group Steiner tree problem, which is NP-Hard [3]. In contrast, with the duo of the original and the reversed O-trees, we can leverage the efficient computation of finding common ancestors based on common prefix of Dewey labels as the LCA-based approaches [20, 22, 10] do.

In brief, we make the following contributions.

- Based on object identification, we introduce a novel semantics for XML keyword search, called NCON, which returns a more complete set of meaningful answers (Section 2).
- We propose an efficient search which uses the O-tree rather than the whole XML document, and reversed O-tree rather than ID/IDREF (Section 3 and Section 4).
- We have implemented all of our ideas in `XRich` system for evaluation. Although `XRich` has overhead from finding HCODs, experimental results show that it outperforms the state-of-the-art approaches in terms of both effectiveness and efficiency because it is still based on tree and works with O-trees rather than whole XML documents (Section 5).



(a) O-tree extracted from data in Fig.1 (b) The reversed O-tree of the O-tree in Fig. 2(a)

Fig. 2: The original and reversed XML object trees (O-trees)

2 OBJECT SEMANTICS

We propose to use the semantics of object for XML keyword search. In this section, we first recall the concept of object in XML and the identification of object in Section 2.1. Based on object, we introduce Nearest Common Object Node semantics for answering XML keyword queries in Section 2.2.

2.1 Object identification

An *object* represents a real world entity. Recall that in XML, an object can be represented by multiple object *instances*, each of which corresponds to a group of nodes, starting at an *object nodes*, followed by *non-object nodes*. An object node may have other object nodes as its descendants. Among all nodes describing an object instance, the object node (i.e., the object class tagged node) is the most "important" because it is the root of the group containing these nodes. Hereafter, it is used as the representative for the entire object instance in unambiguous contexts.

Among non-object nodes, a special attribute (or a set of attributes) together with *object class* that can uniquely define an object is called *object identifier (OID)*. An OID can be a set of attributes because under several cases, a single attribute cannot uniquely identify an object (similar to a key in relational database may contain more than one attribute). OID is different from ID in XML schemas, e.g., DTD because an ID can always be an OID but not vice versa. With ID/IDREF, an OID is defined as an ID in XML schemas. However, in many cases, XML schemas may not have ID for objects such as lower objects in $m : n$ relationships in XML documents without ID/IDREF. Therefore, we need to identify OID in such cases.

Object identification has been studied by third party algorithms such as [14, 15, 12]. We apply the algorithm in [12] which has high accuracy (greater than 99%, 93% and 95% for discovering object class, OID and the overall process, respectively). Thus, for this paper, we assume this task has been done. For example, from the XML data Fig. 1, [12] identifies Professor, Student and Paper as object classes with the corresponding OIDs: StaffID, Stu_No and PID.

Once *object classes* and *OIDs* are identified, we can determine whether instances (or object nodes in other words) refer to the same object based on whether they have *the same object class and the same OID*. For example, object nodes Student(1.1.1) and Student(1.2.1) are of the same object <Student:12745> since they have the same object class Student and the same OID 12745. Multiple object nodes that refer to the same object are usually identical. However, XML does not enforce this. If we find two nodes that are not identical, they are still considered as referring to the same object as long as they have the same object class and OID value.

2.2 The Nearest Common Object Node (NCON) semantics

Based on object identification, we introduce the NCON semantics, which includes both common ancestors and common descendants. The purpose of this inclusion is to provide a more complete set of answers for XML keyword search. Our proposed NCON semantics can be built on the LCA semantics [5] or any of its variants such as SLCA [20], VLCA [10] and ELCA [22]. For simplicity of presentation, we provide the following definitions which are based on the LCA semantics. It is straightforward to make the necessary minor modifications required if any of the variant semantics are preferred.

Let $u \prec_a \ (\succ_a, \ \preceq_a, \ or \ \succeq_a) \ v$ denote that object node u is an ancestor (a descendant, an ancestor-or-self, or a descendant-or-self respectively) of object node v. A keyword k matches an object node u if k is contained by u or by any of non-object nodes associated with u. The NCON semantics and its two components, i.e., LCOA (Lowest Common Object Ancestor) and HCOD (Highest Common Object Descendant) are defined as follows.

Definition 1 (LCOA of a set of object nodes) *Object node u is the LCOA of a set of object nodes $\{u_1, \ldots, u_n\}$ if (1) $u \preceq_a u_i \ \forall i = 1..n$ and (2) there exists no object node $v \succ_a u$ s.t. $v \preceq_a u_i \ \forall i = 1..n$.*

An LCOA is similar to an LCA. However, an LCOA must be an object node while an LCA can be an arbitrary node. This difference enables the NCON semantics to reduce the number of meaningless answers.

Definition 2 (HCOD of a set of object nodes) *Given a set of object nodes $\mathbb{S} = \{u_1, \ldots, u_n\}$, the set of object nodes $\mathbb{H} = \{h_1, \ldots, h_n\}$ is an HCOD of $\mathbb{S}$ if*

- *all h_i's refer to the same object and*
- *$u_i \preceq_a h_i \ \ \forall i = 1..n$ and*
- *there exists no set of object nodes $\mathbb{H}' = \{h'_1, \ldots, h'_n\}$ where $h'_i \prec_a h_i \ \forall i = 1..n$ which satisfies the above two conditions.*

HCOD is the distinguishing feature of the NCON semantics. An HCOD contains a set of object nodes which refer to the same object. Each of them is a descendant of the corresponding matching object node. Note that a set of object nodes has only one LCOA but may have several HCODs because a node has only one parent but several children.

Definition 3 (An NCON of a set of object nodes) *An NCON of a set of object nodes $\mathbb{S}$ is either an LCOA of $\mathbb{S}$ or an HCOD of $\mathbb{S}$.*

Definition 4 (An NCON of a query) *An NCON of a keyword query $Q = \{k_1, \ldots, k_n\}$ is an NCON of a set of object nodes $\mathbb{S} = \{u_1, \ldots, u_n\}$ where u_i matches k_i.*

3 OVERVIEW OF OUR APPROACH

The problem tackled in this paper is to find the set of NCONs for a keyword query issued against a data-centric XML document without IDREF. This section provides an overview of our approach, including the ideas about object orientation and reversal mechanism, and the overview of the process. Detailed techniques will be given in Section 4.

3.1 Object orientation

Based on object nodes, we introduce the concept of XML object tree (O-tree) as follows.

CONCEPT 1 (O-tree) *An O-tree OT is a tree extracted from an XML data tree DT by keeping all object nodes, and associating non-object nodes to the corresponding object nodes. For any object nodes u and v in DT having no other object nodes in between*[7]*, there is a parent-child edge between u and v in OT.*

For example, the O-tree extracted from the XML data in Fig. 1 is shown in Fig. 2(a), in which in each node, Dewey label is used to identify object node while object class and OID are used to identify object.

O-tree brings two important benefits to XML keyword search. First, an answer is more likely to be meaningful since a returned node is an object node in O-tree and it represents a whole object rather than just an attribute or a value. Second, the search space is dramatically reduced because the number of nodes of the extracted O-tree is much smaller than that of the corresponding XML data tree. Suppose that the average number of attributes for an object class is N, then the number of nodes in the XML document is at least $2 \times N$ times larger than that of O-tree (due to not counting attributes and values for the O-tree). This extensively reduces the complexity of the search.

3.2 Reversal mechanism

The set of NCONs includes LCOAs and HCODs. LCOAs can be found by any of existing LCA-based approaches such as [20, 21]. To find HCODs, the idea is a *reversal mechanism* by which HCODs of the given O-tree are turned into LCOAs in its reversed O-tree, which is defined as follows.

CONCEPT 2 (Reversed O-tree) *Given an O-tree OT, the reversed O-tree w.r.t. OT is an O-tree OT_R such that*

- *for each path of object nodes $/u_1/u_2/\ldots/u_{n-1}/u_n$ from the root to a leaf in OT, there is a corresponding reversed path $/u'_n/u'_{n-1}/\ldots/u'_2/u'_1$ in OT_R where each pair of object nodes u_i and u'_i refer to the same object, and*
- *there does not exist any pairs of nodes in OT_R such that they refer to the same object and they have the same list of objects as their ancestors, and*
- *there is no other object node in OT_R.*

For example, Fig. 2(b) shows the reversed O-tree derived from the O-tree in Fig. 2(a).

To derive a reversed O-tree, we need to determine whether two object nodes refer to the same object based on their object class and OID. The reversed O-tree is used with the sole goal of finding HCODs. Although there may be duplication in O-trees, such duplication does not affect the efficiency thanks to our index and search techniques. We assume updating does not frequently happen as LCA-based approaches assume. Otherwise, adding or deleting one node can lead to change Dewey labels of all nodes in an XML document in those approaches.

For XML data containing only binary relationships, LCOAs can be found from original O-tree while HCODs can be found from the reversed O-tree. Thus, although other O-trees, apart from the reversed O-tree, may capture the same information with the original O-tree, only the duo of the original and reversed O-tree is *self-sufficient* to

[7] There may exist non-object nodes between them such as an aggregational node or a grouping node [12].

return the complete set of NCONs. For n-ary relationship ($n \geq 3$), using the reversed O-tree can return more answers than LCA-based approaches, but the results still may not be complete. We leave the improvement of this kind of relationships for future work. Fortunately, such relationships are rare in XML in practice.

3.3 Overview of the process

The process of our approach, as shown in Fig. 3, comprises two components for *pre-processing* and *query processing*. Detailed techniques of these components will be discussed in Section. 4. For pre-processing, the three main tasks are extracting the O-tree from the input XML document, generating the reversed O-tree from the original O-tree and indexing.

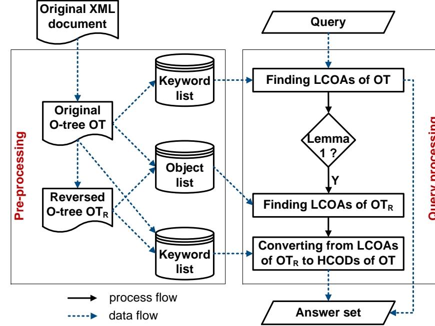


Fig. 3: Overview of the process

For query processing, we follow the reversal mechanism in which HCODs of the original O-tree are turned into LCOAs of the reversed O-tree. Therefore, our process has three steps: finding LCOAs in the original O-tree, finding LCOAs in the reversed O-tree, and converting LCOAs in the reversed O-tree to HCODs in the original O-tree. Our process is flexible in the sense that, it is independent of any LCA semantics adopted, and can be easily deployed to existing LCA-based approaches.

We observe that for several cases, using the reversed O-tree is not necessary because there is no HCOD. So we can optimize the processing with the following lemma.

Lemma 1. *Given an XML keyword query $Q = \{k_1, k_2, \dots, k_n\}$, the reversed O-tree does not provide any new answer if any of the following holds:*

- *Q has only one keyword.*
- *All keywords of Q match the same object.*
- *Keywords of Q may have multiple matches. For a set of matching object nodes $S = \{u_1, u_2, \dots, u_n\}$ where u_i matches keyword k_i, the reversed O-tree does not provide any new answer for S if there exist two different object nodes $u_i, u_j \in S$ which do not represent the same object such that they are the leaf nodes in the original O-tree.*

The first two conditions are intuitive. The rationale behind the third condition is that when u_i and u_j are the leaf nodes in the original O-tree, they become the highest nodes in the reversed O-tree with no ancestor beside the root. They do not have ancestor-descendant relationship for one of them to become a common ancestor either. Therefore, there is no common ancestor of these two nodes. Thus, there is no common ancestor of S. Hence, the reversed O-tree does not provide new answer.

4 DETAILED TECHNIQUES OF OUR APPROACH

Following Section 3, this section presents detailed techniques of our approach.

4.1 Generating the reversed O-tree

The process of generating the reversed O-tree from the original O-tree OT has two steps corresponding to two first conditions of Concept 2.

Step1: reversing object node paths. To reverse object node paths in OT, we traverse OT backward from each leaf node to the root to form a reversed path. Then, all reversed paths are connected to form the intermediate O-tree. Algorithm 1 presents this process. We use an *array-like-stack* S to store all object nodes in OT. An array-like-stack is an array in which *push* and *pop* operators are used in similar way to a stack while we still can access any element in S like an array. We traverse OT by depth first order and push visited object nodes into S. To handle the branches in the tree, we maintain the parent of each object node. Thus, we use the triple $\langle i, (objCls(i) : OID(i)), pre(i)\rangle$ to represent each object node i, where i is the *index* by depth first order (i starts from 1), $objCls(i)$ and $OID(i)$ are the *object class* and *OID* of i and $pre(i)$ is the *index* of the *parent* of i. Note that during the reversal, we associate relationship attributes (if any) to the lowest object node of the relationship it belongs to. Fig. 4 shows the intermediate O-tree w.r.t. the original O-tree in Fig. 2(a).

Algorithm 1: Reversing object node paths

```
   Input: The original O-tree OT
   Output: Intermediate O-tree OT_I
1  Variables: Array-like-Stack S: store object nodes in OT by DF order
2  for visited object node i ∈ OT by DF order do
3      S.Push (⟨i, (objCls(i) : OID(i), pre(i)⟩)
4  OT_I . Add (Root)
5  OT_I . NewBranch
6  while S ≠ ∅ do
7      ⟨i, (objCls(i) : OID(i), pre(i)⟩ ← S.Pop
8      OT_I . Add (objCls(i) : OID(i))
9      //pre(i) = 0:parent is current top element
10     if pre(i) = 0 then
11         OT_I . NewBranch
12     //pre(i) ≠ 0: parent has branches
13     if pre(i) ≠ i − 1 and pre(i) ≠ 0 then
14         k ← pre(i)
15         while k ≠ 0 do
16             Access element k ⟨k, (objCls(k) : OID(k), pre(k)⟩
17             OT_I . Add (objCls(k) : OID(k))
18             if pre(k) = 0 then
19                 OT_I . NewBranch
20             if pre(k) = k − 1 then
21                 k. Next
22             k ← pre(k)
```

Step 2: merging object nodes. To generate the reversed O-tree from reversed object node paths, we merge object nodes having the same set of ancestors. Particularly, at the first level of the intermediate O-tree, we merge branches where the starting object nodes refer to the same object. Then we recursively merge in the lower levels. Fig. 5 demonstrates merging processes w.r.t. the intermediate O-tree in Fig. 4.

Size of the reversed O-tree. In the worst case where there exist $1 : m$ relationships, the size of the reversed O-tree is $\frac{N \times h}{2 \times l}$ where N, h, l are the number of nodes, the height, and the least number of attributes of an objects in the original XML document. The number of object nodes in the original O-tree is N/l. All leaf nodes ($(N/l)/2$ nodes) become the nodes in the first level (after the root node). In the worst case where there is no duplication among them, there will be maximum $\frac{N \times h}{2 \times l}$ nodes.

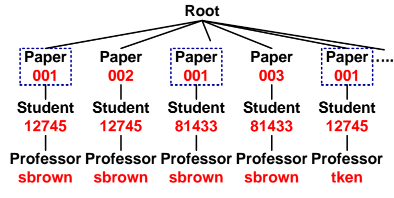
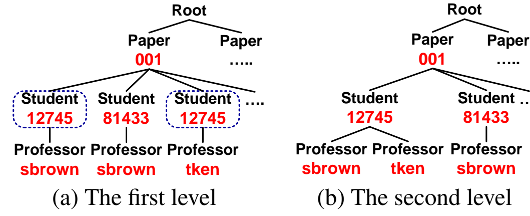


Fig. 4: The intermediate O-tree derived from the O-tree in Fig.2(a)



(a) The first level

(b) The second level

Fig. 5: Merging branches having the same set of ancestors

4.2 Indexes

Since a common descendant may appear as many different nodes, we need more complex kinds of index to accelerate finding common descendants.

Keyword list. Keyword list is to efficiently retrieve the set of matching objects[8] in the original XML document. Each keyword matches a list of objects ordered decreasingly by hierarchical level of objects. The space cost of the keyword list is $K \times M$ where K is the number of keywords in XML document and M is the maximum number of objects matching a keyword. Table 1 shows a part of keyword list of the XML data in Fig. 1.

Table 1: A part of keyword list of the XML data in Fig. 1

Keyword	Matching objects
Kennedy	<Professor:tken>, <Student:12745>, <Paper:001>
Clinton	<Student:002>, <Paper:001>
...	...

Object list. Object list is created for two purposes. It is used to determine whether two object nodes refer to the same object or not, and more importantly, to identify the set of object nodes in the reversed O-tree w.r.t. a given object. The latter will be used to find HCODs. Each object corresponds to a list of Dewey labels of its object nodes sorted by preorder numbering. The space cost of the object list is $M \times N$ where M is the total number of objects in the original O-tree and N is the maximum number of object nodes of an object. Part of the object list of the O-trees in Fig. 2 is given in Table 2.

Table 2: A part of object list of the O-trees in Fig. 2

Objects	Objects nodes in the original O-tree	Object nodes in the reversed O-tree
<Professor:sbrown>	1.1	1.1.1.1, 1.1.2.1, 1.2.1.1, 1.3.1.1
<Student:12745>	1.1.1, 1.2.1	1.1.1, 1.2.1
<Student:81433>	1.1.2	1.1.2, 1.3.1
<Paper:001>	1.1.1.1, 1.1.2.1, 1.2.1.1, 1.2.2.1	1.1
...	...	...

Reversed list. Given an object node in the reversed O-tree, reversed list is to trace back to corresponding object nodes in the original O-tree for final output presentation. It costs $N \times L$ where N is the number of object nodes in the reversed O-tree and L is the maximum number of object nodes in the original O-tree w.r.t. a given object node in the reversed O-tree. Table 3 shows a part of the reversed lists w.r.t. the O-trees in Fig. 2.

Table 3: A part of reversed list of the O-trees in Fig. 2

Object nodes in reversed O-tree	Corresponding object nodes in original O-tree
1.1	1.1.1.1, 1.1.2.1, 1.2.1.1, 1.2.2.1
1.1.1	1.1.1, 1.2.1
...	...

[8] An object matches keyword k when any of its object node matches k.

4.3 QUERY PROCESSING

As shown in Fig. 3, to process a keyword query $Q = \{k_1, \ldots, k_n\}$, we have three steps.
Step1: finding LCOAs from the original O-tree OT. We can use any of existing LCA-based algorithms for this task. The list of object nodes matching keyword k_i can be retrieved from the *keyword list* and *object list*. Consider a set of matching object nodes $S = \{u_1, \ldots, u_n\}$ where u_i matches k_i. We denote $LCOA^O(S)$ and $LCOA^R(S)$ be the set of LCOAs for S w.r.t. the original O-tree OT and the reversed O-tree $OT_{_R}$, respectively.

Based on Lemma 1, we determine whether we need to find $LCOA^R(S)$ or not. Since the reversed O-tree is used without users' awareness, $LCOA^R(S)$ will be converted to HCODs w.r.t. the original O-tree.

Step 2: finding LCOAs of the reversed O-tree $\mathbf{OT_R}$. To find $LCOA^R(S)$, from S we identify the corresponding sets of object nodes on $OT_{_R}$. To do this, we look up the *object list*. Note that, there may be more than one corresponding set in $OT_{_R}$. After that, we can apply the same algorithm with the algorithm of finding $LCOA^O(S)$.

Step 3: converting $\mathbf{LCOA^R}$(S) into HCODs of OT. An LCOAs v of $OT_{_R}$ corresponds a set $Superset$ of nodes in OT, which can be found by looking up *reversed list*. HCODs is the subset $H = \{h_1, \ldots, h_n\}$ of $Superset$ where h_i is a descendant of u_i.

All ideas discussed above about finding HCODs for a given set of matching object nodes $S = \{u_1, \ldots, u_n\}$ are presented in Algorithm 2.

Algorithm 2: Finding $HCODs(S)$

Input: A set of matching object nodes $S = \{u_1, \ldots, u_n\}$
Object list
Reversed list
Output: $HCODs(S)$

1 **for** *each* u_i **do**
2     $S_i \leftarrow$ looking up object nodes of $OT_{_R}$ in object list
3 $LCOA^R(S) \leftarrow LCOA(S_1, \ldots, S_n)$
4 **for** *each* $v \in LCOA^R(S)$ **do**
5     $Superset \leftarrow$ looking up object nodes of OT w.r.t. v in reversed list
6   **for** *each matching object node* u_i *in* S **do**
7         $h_i \leftarrow e \in Superset$ and $e \succeq_a u_i$
8     $HCOD(S)$.Add($\{h_1, \ldots, h_n\}$)

Complexity. The cost of finding HCODs(S) is dominated by the cost of looking up a node in object list and reversed list. In the worst case, it is $\log(m) \times \log(n)$ for the former, where m and n are the number of objects matched query keywords and the maximum number of object nodes w.r.t. an object. For the later, it is $\log(N) \times \log(L)$ where N and L has similar meanings in reversed list.

Presentation of an answer. To avoid irrelevant information, we present an answer as a path from a returned NCON, i.e., a (set of) object node(s) to matching object nodes.

Fig. 6 shows the process and outputs for a set of matching nodes of query {`Clinton, Kennedy`} issued again the XML data in Fig. 1, in which one final output corresponds to an LCOA and the other corresponds to an HCOD.

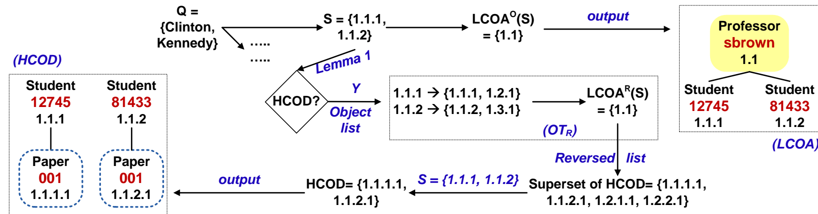
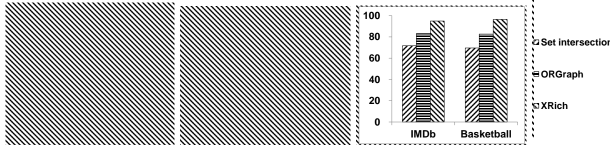


Fig. 6: Process and output of query {`Clinton, Kennedy`}



(a) Precision (b) Recall (c) F-measure

Fig. 7: Effectiveness Evaluation

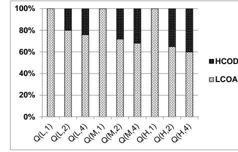


Fig. 8: Percentage of HCODs in NCONs

5 EXPERIMENT

We have developed `XRich`, a system for XML keyword search, based on our proposed approach. `XRich` was implemented using Java and was used for experimental evaluation. This section evaluates `XRich` on three aspects including efficiency, effectiveness and quality of the generated reversed O-tree.

5.1 Experimental Setup

Environment. Experiments were performed on a dual-core Intel Xeon CPU 3.0GHz running Windows XP operating system with 4GB of RAM and a 320GB hard disk.

Datasets. We pre-processed two real datasets including **IMDb**[9], and **Basketball**[10]. We used the subsets with the sizes 150MB and 86MB for IMDb and Basketball respectively. IMDb dataset contains information about movies, actors, actresses, companies, and etc. An actor or actress can play for many movies, and a company can produce for several movies. Basketball dataset contains information about coaches, teams, players where a player and a coach can work for different teams in different years. Table 4 gives more statistics of the datasets.

Table 4: Statistics of datasets

Dataset	No. of nodes	No. of object nodes	No. of object classes	No. of keywords	**Data size**
IMDb	2,501,780	387,422	6	291,004	150M
Basketball	1,035,940	100,140	3	123,100	86M

Query set. We randomly generated 120 queries from document keywords. To avoid meaningless queries, we filtered out generated queries which do not contain any value keyword, such as queries contains only tags, or prepositions, or articles, e.g., query $\{$actor, the, to$\}$. 87 remaining queries include 34 and 53 queries for Basketball and IMDb datasets respectively.

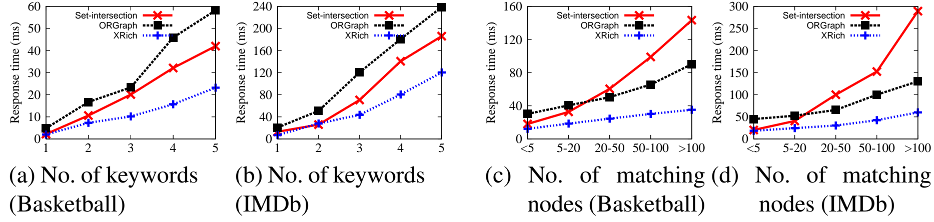
Compared Algorithms. We compared `XRich` with an LCA-based approach to show the advantages of our approach over LCA-based approaches. We chose Set-intersection [21] because it is recent and it outperforms other LCA-based approaches in term of efficiency. We also compare `XRich` with ORGraph[9] because it can also find common descendants. ORGraph converts XML document to a graph similar to relational database and is based on the Steiner tree semantics.

Metrics. To measure the efficiency, we compared the running time of approaches. We selected five (among 87) queries for each kind of queries, e.g., 2-keyword query. For each query, we ran it ten times to get the average response time. We finally reported the average response time of five queries for one kind of query.

To evaluate the effectiveness, we used standard *Precision* ($\mathcal{P}$), *Recall* ($\mathcal{R}$), and *F-measure* ($\mathcal{F}$) metrics. *F-measure* is the harmonic mean of precision and recall, and is calculated as $\mathcal{F}_\alpha = \frac{(1+\alpha^2)\times\mathcal{P}\times\mathcal{R}}{\alpha^2\times\mathcal{P}+\mathcal{R}}$. Here we choose $\alpha = 1$ to evenly weight to precision

[9] http://www.imdb.com/interfaces

[10] http://www.databasebasketball.com/stats_download.htm



(a) No. of keywords (Basketball) (b) No. of keywords (IMDb) (c) No. of matching nodes (Basketball) (d) No. of matching nodes (IMDb)

Fig. 9: Efficiency evaluation

and recall. Other values of α provides similar results. We randomly selected a subset (32 queries) of 87 generated queries for effectiveness evaluation. To compute precision and recall, we conducted surveys on the above 32 queries and the tested datasets. We asked 15 students in major of computer science to interpret 32 queries. Due to ambiguity of queries, a student may interpret a query in different ways. Common interpretations from at least 12 out of 15 (80%) students are considered as common intuitions. We then manually reformulate these interpretations into schema-aware XQuery queries and use their results as the ground truth.

5.2 Effectiveness Evaluation

Effectiveness. Fig. 7 shows the effectiveness of all compared approaches. As seen, `XRich` achieves high precision and recall (both are higher than 96%). Compared to Set-intersection, `XRich` outperforms Set-intersection both in term of recall and precision because `XRich` returns common descendants while Set-intersection does not; and a returned node of `XRich` corresponds an object node rather than an arbitrary node of Set-intersection. The difference in terms of recall (more than 25%) is higher than in term of precision. `XRich` improves both precision and recall, but the more important contribution is improving recall.

Compared to ORGraph, based on undirected Steiner tree, ORGraph has a lightly higher recall than `XRich`, however, `XRich` significantly outperforms ORGraph in term of precision because beside common descendants, ORGraph may also return many meaningless answers in sense that it is hard (or even impossible) to interpret such answers because the matching nodes have weak or no relationships. Therefore, if precision and recall is evenly weighed, the F-measure of `XRich` is higher than that of ORGraph as shown in Fig. 7(c).

Percentage of HCODs in NCONs. Fig. 8 shows the percentage of HCODs and LCOAs in NCONs for 9 queries containing $1-4$ keywords. Low (L), medium (M) and high (H) frequencies of keywords correspond to the number of matching objects between 1-100, 100-1000, and above 1000, respectively. $Q(f, k)$ denotes a query containing k keywords with frequency f. For 1-keyword queries, there is 0% HCOD because the reversed tree provides no new answers for such cases. For other queries, the high percentage of HCODs (20% - 40%) shows the importance of finding HCODs. The higher k and f are, the higher that percentage is.

5.3 Efficiency Evaluation

Efficiency. The response time of approaches is shown in Fig. 9, in which we varied the number of query keywords and the number of matching nodes. Although XRich has overhead from finding HCODs, it still outperforms the other algorithms because it searches over the O-tree which is much smaller than the XML document and only uses the reversed O-tree when necessary. Set-intersection runs slower because it works with

	Basketball	IMDb
Quality of extracted O-tree (%)	100	99.5
Quality of reversed O-tree (%)	100	99.5
Time to extract O-tree (min)	4	10.5
Time to generate reversed O-tree (min)	3.5	5.8

Fig. 10: Extracting original O-tree and generating reversed O-tree

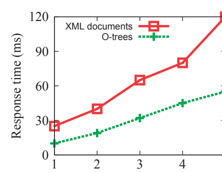


Fig. 11: O-tree vs. XML data tree

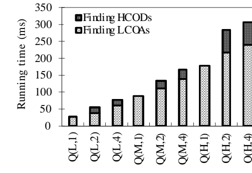


Fig. 12: Overhead of finding HCODs

the whole large XML document. ORGraph runs also slower because ORgraph follows undirected Steiner tree semantics, which would lead exponential computation [3].

Overhead of finding HCODs. Fig. 12 shows the overhead of finding HCODs for 9 queries discussed in Fig. 8. As shown, it is around 24.7% of the total time, which is not double thanks to Lemma 1.

Impact of object on efficiency. Fig 11 shows the response time of `XRich` when it searches over the O-trees versus the corresponding XML documents. It shows that it runs much faster with the O-trees, especially when the number of keywords increases because the size of the O-trees is much less than that of the XML documents. We randomly chose IMDb because Basketball dataset provides similar results.

5.4 Quality of the extracted and reversed O-trees

To test the quality of the O-tree extracted from XML document, we check the accuracy of the object class and OID discovery. To test the reversed O-tree, we computed the ratio of the number of satisfied object nodes over the total number of object nodes in the reversed O-tree. The satisfied nodes are those in the reversed O-tree that satisfy the reversed schema (object class) which is manually generated. The results are given in Table 10. As can be seen, the quality of the reversed O-tree depends on the quality of the O-tree extracted from XML document, which is very high since our technique can discover object class and OID with high accuracy. Once the O-tree is extracted, the reversed O-tree can be derived accurately. The cost of these processes is not expensive since this computation is performed offline and only once.

6 RELATED WORK

LCA-based approaches. XRANK [5] proposes a stack based algorithm to efficiently compute LCAs. XKSearch [20] defines Smallest LCAs (SLCAs) to be the LCAs that do not contain other LCAs. Meaningful LCA (MLCA) [13] incorporates SLCA into XQuery. VLCA [10] and ELCA [22] introduces the concept of valuable/exclusive LCA to improve the effectiveness of SLCA. MESSIAH [18] handles cases of missing values in optional attributes. Although extensive works have been done on improving the effectiveness, these works may return incomplete answer sets since they find only common ancestors but not common descendants.

Graph-based approaches. Graph-based approaches can be classified based on the semantics such as the Steiner tree [2], distinct root [6] and subgraph [11, 7]. The Stener tree semantics can return common descendants if the XML document contains ID/IDREF, but it also returns meaningless answers as well. Distinct root semantics is similar to LCA semantics and cannot find common descendant. Sub-graph semantics provides more information for answers but still miss common descendants if ID/IDREF is not used in XML documents.

Object-oriented approaches. Object have been introduced in XSeek [14], XReal [1], and [19]. However, none of the above works considers OID. Thus, they may not distinguish an object and a composite attribute and/or a multi-valued attribute.

7 CONCLUSION

This paper shows advantages of object identification in XML keyword search. Based on object identification, we introduced the NCON semantics for XML keyword search, by which an answer corresponds to an object and the answer set includes not only common ancestors but also common descendants. We also proposed an approach based on the NCON semantics and use both the original and the reversed O-trees to find answers. Experimental results showed that `XRich` outperforms LCA-based and graph-based approaches in terms of both effectiveness and efficiency. Therefore, the approach could be a promising direction for XML keyword search to return a more complete set of meaningful answers. More broadly, this paper demonstrates the benefit of object orientation in XML. In future work, we will explore how other XML processing can similarly benefit and how to handle n-ary ($n \geq 3$) relationships.

References

1. Z. Bao, T. W. Ling, B. Chen, and J. Lu. Efficient XML keyword search with relevance oriented ranking. In *ICDE*, 2009.
2. B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in database. In *ICDE*, 2007.
3. S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. *Networks*, 1971.
4. J. Fong, H. K. Wong, and Z. Cheng. Converting relational database into XML documents with DOM. *Information & Software Technology*, 2003.
5. L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.
6. H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD*, 2007.
7. M. Kargar and A. An. Keyword search in graphs: finding r-cliques. *PVLDB*, 2011.
8. J. Kim, D. Jeong, and D.-K. Baik. A translation algorithm for effective RDB-to-XML schema conversion considering referential integrity information. *Journal Inf. Sci. Eng.*, 2009.
9. T. N. Le, H. Wu, T. W. Ling, L. Li, and J. Lu. From structure-based to semantics-based: Effective XML keyword search. *ER*, 2013.
10. G. Li, J. Feng, J. Wang, and L. Zhou. Effective keyword search for valuable LCAs over XML documents. In *CIKM*, 2007.
11. G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: Efficient and adaptive keyword search on unstructured, semi-structured and structured data. In *SIGMOD*, 2008.
12. L. Li, T. N. Le, H. Wu, T. W. Ling, and S. Bressan. Discovering semantics from data-centric XML. *DEXA*, 2013.
13. Y. Li, C. Yu, and H. V. Jagadish. Schema-free XQuery. In *VLDB*, 2004.
14. Z. Liu and Y. Chen. Identifying meaningful return information for XML keyword search. In *SIGMOD*, 2007.
15. L. Ribeiro and T. Härder. Entity identification in XML documents. In *Grundlagen von Datenbanken*, 2006.
16. Y. Tao, S. Papadopoulos, C. Sheng, and K. Stefanidis. Nearest keyword search in XML documents. In *SIGMOD*, 2011.
17. A. Termehchy and M. Winslett. EXTRUCT: using deep structural information in XML keyword search. *PVLDB*, 2010.
18. B. Q. Truong, S. S. Bhowmick, C. E. Dyreson, and A. Sun. MESSIAH: missing element-conscious slca nodes search in XML data. In *SIGMOD*, 2013.
19. H. Wu and Z. Bao. Object-oriented XML keyword search. In *ER*, 2011.
20. Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *SIGMOD*, 2005.
21. J. Zhou, Z. Bao, W. Wang, T. W. Ling, Z. Chen, X. Lin, and J. Guo. Fast slca and elca computation for XML keyword queries based on set intersection. In *ICDE*, 2012.
22. R. Zhou, C. Liu, and J. Li. Fast ELCA computation for keyword queries on XML data. In *EDBT*, 2010.