Finding Missing Answers due to Object Duplication in XML Keyword Search

Thuy Ngoc Le, Zhong Zeng, Tok Wang Ling

National University of Singapore
{ltngoc,zengzh,lingtw}@comp.nus.edu.sg

Abstract. XML documents often have duplicated objects, with a view to maintaining tree structure. Once object duplication occurs, two nodes may have the same object as the child. However, this child object is not discovered by the typical LCA (Lowest Common Ancestor) based approaches in XML keyword search. This may lead to the problem of missing answers in those approaches. To solve this problem, we propose a new approach, in which we model an XML document as a so-called XML IDREF graph so that all instances of the same object are linked. Thereby, the missing answers can be found by following these links. Moreover, to improve the efficiency of the search over XML IDREF graph, we exploit the hierarchical structure of the XML IDREF graph so that we can generalize the efficient techniques of the LCA-based approaches for searching over XML IDREF graph. The experimental results show that our approach outperforms the existing approaches in term of both effectiveness and efficiency.

1 Introduction

Since XML has become a generally accepted standard for data exchange over the Internet, many applications use XML to represent the data. Therefore, keyword search over XML documents has attracted a lot of interests. The popular approach for XML keyword search is the LCA (Lowest Common Ancestor) semantics [4], which was inspired by the hierarchical structure of XML. Following this, many extensions of the LCA semantics such as SLCA [17], MLCA [13], ELCA [19] and VLCA [10] have been proposed to improve the effectiveness of the search. However, since these approaches only search up to find common ancestors, they may suffer from the problem of missing answers as discussed below.

1.1 The problem of missing answers due to object duplication

XML permits nodes to be related through parent-child relationships. However, if the relationship type between two object classes is many-to-many without using IDREF, an object can occur at multiple places in an XML document because it is duplicated for each occurrence in the relationship. We refer such duplication as *object duplication*.

Example 1 *Consider an XML document in Figure 1 where the relationship type between student and course is many-to-many ($m : n$) and students are listed as children of courses. When a student takes two courses, this student is repeated under*

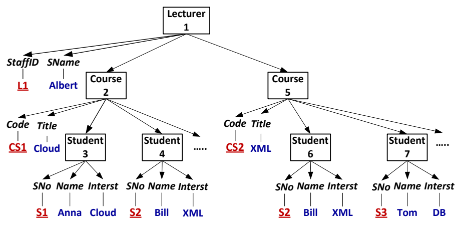


Figure 1: XML data tree

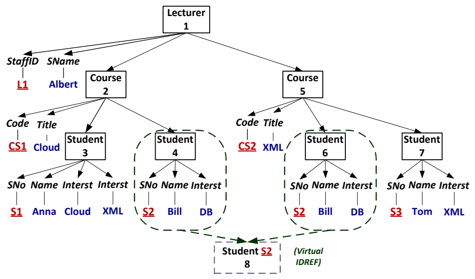


Figure 2: XML IDREF graph w.r.t. XML data tree in Figure 1

both courses. For example, both courses <Course:CS1>[1] *and* <Course:CS2> *are taken by* <Student:S2>*, which is repeated as the two groups of nodes, starting at node 4 and node 6 under the two courses. This causes the duplication of object* <Student:S2>*.*

When object duplication happens, two nodes may have the same object as the child. However, this *common child object* is not discovered by the LCA-based approaches because they only search up from matching nodes for common ancestors, but never search down to find common information appearing as descendants of matching nodes. We call this incident as the problem of *missing answers due to object duplication* which leads to loss of useful information as illustrated below.

Example 2 *Consider keyword query* {CS1, CS2} *issued against the XML data in Figure 1, where the keywords match object identifier of two courses (node 2 and node 5). The LCA-based approaches return only* <Lecturer:L1> *(node 1) as an answer. However, as discussed in Example 1, object* <Student:S2> *is the common student taking both matching courses and thus it should also be an answer. Intuitively, the two courses are not only taught by the same lecturer (*<Lecturer:L1>*), but also taken by the same student (*<Student:S2>*). As we can see, common information related to query keywords appearing as both ancestors and descendants are meaningful to users.*

Object duplication can be eliminated by ID/IDREF. However, to maintain the tree structure for ease of understanding, readability and retrieval, XML designers may duplicate objects instead of using ID/IDREF. In practice, object duplication is a common scheme for maintaining a view of tree structure. For example, suppose 300 students take course A. Among them, 200 students also take another course B. Then, if students are listed as children of courses, these 200 students are duplicated under both courses. Many real XML datasets, including IMDb[2] and NBA[3] (used in experiments of XML research works such as [16, 15]), contain object duplication. In IMDb, an actor or actress can play in many movies, and a company can produce several movies. In NBA, a

[1] <Course:CS> denotes an object which belongs to object class Course and has object identifier CS1.
[2] http://www.imdb.com/interfaces
[3] http://www.nba.com

player can play for several teams in different years. Moreover, due to the flexibility and exchangeability of XML, many relational datasets with many-to-many relationships can be transformed to XML [3] with object duplication in the resulting XML documents. Therefore, the problem of missing answers due to object duplication frequently happens in XML keyword search and necessitates to be solved.

For an XML document with ID/IDREF, graph-based approaches such as [11, 7] can provide missing answers due to object duplication. However, those graph-based approaches can find such missing answers only if all objects are covered by ID/IDREF mechanism. Otherwise, those graph-based approaches do not recognize instances of the same object appearing in different places in an XML document. In such cases, they cannot find missing answers either.

1.2 Our approach and contributions

In this paper, we propose an approach for keyword search over a data-centric XML document which can find missing answers due to object duplication. The input XML document in our approach can contain both objects under ID/IDREF mechanism and duplicated objects. For the latter, we propose a virtual object node to connect all instances of the same object via virtual IDREFs. The resulting model is called *XML IDREF graph*. "Virtual" here means we do not modify XML documents and ID/IDREF links are virtually created with the sole goal of finding missing answers.

A challenge appears when we have to deal with an XML IDREF *graph*, not a *tree* anymore. Searching over an arbitrary graph-structured data has been known to be equivalent to the group Steiner tree problem, which is NP-Hard [2]. In contrast, keyword search on XML tree is much more efficient thanks to the hierarchical structure of XML tree. This is because the search in an XML tree can be reduced to find LCAs of matching nodes, which can be efficiently computed based on node labels.

We discover that XML IDREF graph is a special graph. Particularly, it is an XML tree (with parent-child (PC) edges) plus a portion of *IDREF edges*. An IDREF edge is an edge from a referring node to a referred node. Although these nodes refer to the same object, we can treat them as having a parent-child relationship, in which the parent is the referring node and the child is the referred node. This shows that XML IDREF graph still has hierarchy, which enables us to generalize efficient techniques of LCA-based approaches (based on the hierarchy) for searching over our proposed XML IDREF graph. Thereby, we do not have to traverse the XML IDREF graph to process a keyword query.

Contribution. In brief, we make the following contributions.

- We argue that LCA-based approaches, which only search up to find common ancestors, may miss meaningful answers due to object duplication. To find such missing answers, we model an XML document as an XML IDREF graph, in which all instances of the same object are connected by a virtual object node.
- We discover the hierarchical structure of an XML IDREF graph which distinguishes it from an arbitrary graph. Based on this hierarchical structure, we can generalize techniques of the LCA-based approaches for an efficient search.
- The experimental results show that our approach outperforms both the graph-based and LCA-based approaches in term of both effectiveness and efficiency.

Roadmap. The rest of the paper is organized as follows. We introduce data model and answer model in Section 2. Our approach is described in Section 3. The experiment and evaluation are provided in Section 4. We review related works in Section 5. Finally, we conclude the paper in Section 6.

2 Data and answer model

2.1 Data model

In XML, an object can be referred to either by duplicating it under the referrer or by using ID/IDREF. The former causes object duplication whereas the latter does not. With ID/IDREF, an object has only one instance, and other objects refer to it via ID/IDREF. Without ID/IDREF, an object can be represented as many different instances. We propose virtual ID/IDREF mechanism, in which we assign a virtual object node as a hub to connect all instances of the same object by using virtual IDREF edges. The resulting model is called an XML IDREF graph which is defined as followed.

Definition 1 (XML IDREF graph) *An XML IDREF graph $G(V, E)$ is a directed, labeled graph where V and E are nodes and edges of the graph.*

- *$V = V_R \cup V_V$ where V_R and V_V are* real *and* virtual *nodes respectively. A real node is an object node in XML document. A virtual node is a virtual object node to connect all instances of the same object in XML document.*
- *$E = E_R \cup E_V$ where E_R and E_V are* real *edges and* virtual *edges respectively. A real edge (can be a real PC edge or real IDREF edge) is an edge between two real nodes. A virtual edge is the edge links an instance of a duplicated object (real node) to a virtual object node.*

For example, Figure 2 shows an XML IDREF graph with two virtual edges from `node 4` and `node 6` to a virtual object node (`node 8`) because `node 4` and `node 6` are instances of the same object `<Student:S2>`.

XML permits some objects under ID/IDREF mechanism and some other objects with duplication co-exist in an XML document. In this case, the resulting XML IDREF graph has two types of IDREF: real and virtual. Thus, an XML IDREF graph may have three types of edges: PC edge, real IDREF edge and virtual IDREF edge.

The hierarchical structure of XML IDREF graph. We observe that an XML IDREF graph still has hierarchy with parent-child (PC) relationships represented as containment edges (PC edges) or referenced edges. This is because nodes in a referenced edge can be considered as having PC relationship, in which the parent is the referring node and the child is the referred node.

Importance of the hierarchical structure of XML IDREF graph. Once we discover the hierarchical structure of an XML IDREF graph, we can inherit the efficient search techniques of LCA-based approaches which based on the hierarchical structure of XML tree. Thereby, we do not have to traverse the XML IDREF graph to process a keyword query as graph-based search does. This brings a huge improvement on efficiency. Without the property of the hierarchy, generally, in graph-based search, matching nodes will be expanded to all directions until they can connect to one another. In theory, there can be exponentially many answers under the Steiner tree based semantics: $O(2^m)$ where m is the number of edges in the graph. The graph-based search has been well known to be equivalent to the group Steiner tree problem, which is *NP-Hard* [2].

Generating XML IDREF graph. An object instance in XML is usually represented by a group of nodes, rooted at the object class tagged node, followed by a set of attributes and their associated values to describe its properties. In this paper, we refer to the root of this group node as an *object node* and the other nodes as *non-object nodes*. Hereafter, in unambiguous contexts, we use object node as the representative for a whole object instance, and nodes are object nodes by default. For example, matching node means matching object nodes. Among non-object nodes, *object identifier* (OID) can uniquely identify an object. To generate an XML IDREF graph from an XML document, we need to detect object instances of the same object. Since an object is identified by object class and OID, we assume that two object instances (object nodes as their representatives) are of the same object if they belong to the same *object class* and have the same *OID*.

We assume that the data is consistent and we work on a single XML document. Data integration, data uncertainty, and heterogeneous data are out of the scope of this work. Object classes and OIDs can be discovered from XML schema and data by our previous works [12], which achieve high accuracy (greater than 98% for object classes and greater than 93% for OIDs). Therefore, we assume the task of discovering object class and OID has been done. Interested readers can find more details in [12].

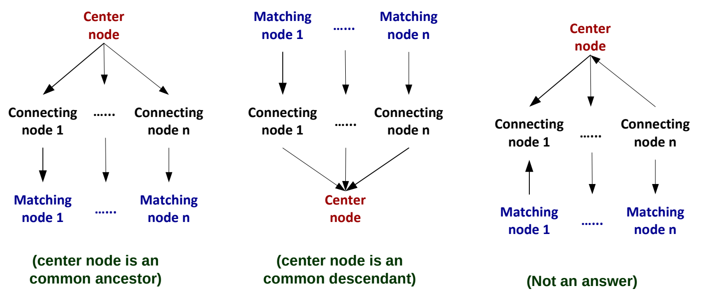
2.2 Answer model

Consider a n-keyword query $Q = \{k_1, \ldots, k_n\}$. An answer to Q contains three kinds of nodes: *matching nodes*, *center nodes* and *connecting nodes*. A matching node contains keyword(s). A center node connects all matching nodes through some intermediate nodes (called connecting nodes). Based on the hierarchical structure of XML IDREF graph, there exist ancestor-descendant relationships among nodes in an XML IDREF graph. Therefore, we can define an answer to Q as follows:

Definition 2 *Given a keyword query $Q = \{k_1, \ldots, k_n\}$ to an XML IDREF graph G, an answer to Q is a triplet $\langle c, \mathbb{K}, \mathbb{I} \rangle$, where $c, \mathbb{K}$, and $\mathbb{I}$ are called the* center node*, the set of* matching nodes *and the set of* connecting nodes *(or intermediate nodes) respectively. $\mathbb{K} = \bigcup_1^n u_i$ where u_i contains k_i. An answer satisfies the following properties:*

- *(P1: Connective) For every i, c is an ancestor of u_i or for every i, c is a descendant of u_i, i.e., c is either a common ancestor or a common descendant of u_i's.*
- *(P2: Informative) For any answer $\langle c', \mathbb{K}', \mathbb{I}' \rangle$ where $\mathbb{K}' = \bigcup_1^n u'_i$ and u'_i contains k_i:*
  - *if c and c' are both common* ancestors *of u_i's, and of u'_i's respectively, and c' is a* descendant *of c, then $\forall i\ u_i \notin \mathbb{K}'$.*
  - *if c and c' are both common* descendant *of u_i's, and of u'_i's respectively, and c' is a* ancestor *of c, then $\forall i\ u_i \notin \mathbb{K}'$.*
- *(P3: Minimal) It is unable to remove any node in an answer such that it still satisfies properties P1 and P2.*

Among nodes in an answer, the center node is the most important one because it connects matching nodes through connecting nodes. It corresponds to both common ancestors and common descendants (Figure 3(a)). Intuitively, common ancestors are similar the LCA semantics while common descendants provide the missing answers. We do not return the subgraph in Figure 3(b) because it may provide meaningless answers. In other words, a center node has only incoming edges (common descendant), or only outgoing edges (common ancestor), but not both.



(a) Answer cases (b) Not answer cases

Figure 3: Illustration for answers

The second Property P2 of Definition 2 is to avoid overlapping information in answers. Each answer needs to contribute new information by having its own set of matching nodes, i.e., matching nodes of an answer cannot also be matching nodes of other answers where the latter is an ancestors/descendants of the former one. This property is similar to the constraint in the ELCA [19] semantics.

3 Our approach

Our approach takes a data-centric XML document as the input, models it as an XML IDREF graph, and returns answers as defined in Definition 2. In the input XML document, objects under IDREF mechanism and objects with duplication can co-exist.

The process of our approach, as shown in Figure 4, comprises of two major components for pre-processing and runtime processing. For pre-processing, there are two main tasks, namely generating XML IDREF graph (discussed in Section 2.1), and indexing (will be discussed in Section 3.2). For runtime processing, there are three main tasks, each of which corresponds to a property of an answer in Definition 2. Particularly, task 1 is to find potential center nodes, task 2 is to find real center nodes, and task 3 is to track back matching nodes and look up

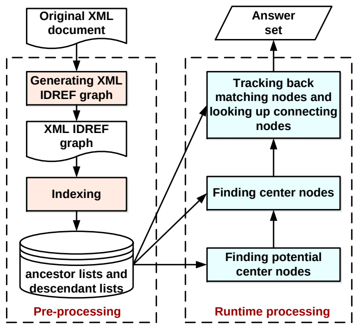

Original XML document
Generating XML IDREF graph
XML IDREF graph
Indexing
ancestor lists and descendant lists
Pre-processing
Answer set
Tracking back matching nodes and looking up connecting nodes
Finding center nodes
Finding potential center nodes
Runtime processing

Figure 4: The process of our approach

connecting nodes. These steps will be discussed in Section 3.3. Before discussing detailed techniques, we provide the overview of the approach at the conceptual level in Section 3.1.

3.1 Overview of the approach

A. Features of our approach. Our approach has three main features: duplication-aware, hierarchy-aware and object-orientation.
Duplication-aware. We recognize that there exists object duplication in XML documents. Thus, we model an XML document as an XML IDREF graph so that all instances of the same object can be linked to a virtual object node. This enables us follows these links to find *missing answers*.
Hierarchy-aware. We are aware of the hierarchical structure of an XML IDREF graph and exploiting it in finding answers. This offers a great opportunity to improve the efficiency of the search by generalizing the LCA-based techniques instead of doing a general graph-based search.

Note that the above two features are the main focus of the paper. However, since object orientation has been demonstrated to be useful in many areas of computation (including database field), our approach also relies upon object orientation.
Object-orientation. All nodes of the same object instance can be grouped. Among these nodes, object node is the most important one and should be chosen as the representative of the group. Instead of working with all these nodes, we only work with the representative of each group, i.e., the object node, and associate all non-object nodes to the corresponding object node. This largely reduces search space and improve the efficiency of the search.

B. Basic ideas of runtime processing. Our runtime processing has three main tasks corresponding to three properties of an answer based on Definition 2. Firstly, we need to find *potential center nodes* which can be either a common ancestor or a common descendant of matching nodes. Secondly, we have to find *center nodes* which can provide informative answers. Finally, we get *full answers* w.r.t. a center node by tracking back matching nodes and looking up connecting nodes. Follows are theories behind the three main tasks. Detailed techniques will be provided in Section. 3.3.
Finding potential center nodes. Consider a keyword query $Q = \{k_1, \ldots, k_n\}$. Let $Anc(Q)$ be the set of common ancestors of Q, i.e., $\forall u \in Anc(Q)$, u is a common ancestor of $\{u_1, \ldots, u_n\}$ where u_i contains k_i. Similarly, let $Dec(Q)$ denote the set of common descendants of Q. Based on *Property P1* of Definition 2, obviously, we have the following property:

Property 1 *Given an answer* $\langle c, \mathbb{K}, \mathbb{I} \rangle$ *for a keyword query* Q*, the* center node $c \in Anc(Q) \cup Des(Q)$.

For a set of nodes, common descendants of these nodes can only be object nodes (real or virtual) which is referred by some other node(s) by IDREF links. We call them *referred object node*. Let $Ref(k)$ is the set of referred object nodes w.r.t. k, each of which is a descendant of some node containing k. For example, in Figure 2,

$Ref(Cloud) = \{8\}$. Let $Ref(Q)$ be the set of referred object nodes w.r.t. Q. We have the following property about common descendants.

Property 2 *Given a keyword query Q, $Des(Q) = Ref(Q)$.*

By Property 1 and Property 2, $Anc(Q)$ and $Ref(Q)$ can provide potential center nodes. Since $Anc(Q) = \bigcap_1^n Anc(k_i)$, and $Ref(Q) = \bigcap_1^n Ref(k_i)$, in order to find $Anc(Q)$ and $Ref(Q)$, we use computation of *set intersection*. The computation of set intersection has been used to find SLCA and ELCA in [18] and has been shown to be more efficient than the traditional computation based on common prefix of labels when dealing with XML tree.
Finding center nodes. Among potential center nodes, we identify *real center nodes* by checking *Property P2* of Definition 2, which infers that an answer should have its own matching nodes from its ancestor/descendant answers to be informative. Let $Des(c, k)$ denote the set of descendants of node c which contains keyword k. $Center\_Des(c)$ denotes center nodes which are descendants of c. $Content\_Des(c, k)$ denotes the set of matching nodes w.r.t. all nodes in $Center\_Des(c)$. We have the following property.

Property 3 *Given a keyword query $Q = \{k_1, \ldots, k_n\}$ and $c \in Anc(Q)$, if $Des(c, k) - Content\_Des(c, k) \neq \emptyset \ \forall i = 1..n$, then c is a real center node.*

We use bottom up for checking common ancestors. For a *common ancestor* c, after removing matching nodes of descendant answers out of $Des(c, k)$'s, if c still has its own matching nodes, then it is a real center node. This is similar for checking whether a *common descendant* is a center node. However, the process is top down.
Tracking back matching nodes and looking up connecting nodes. To return a full answer $\langle c, \mathbb{K}, \mathbb{I}\rangle$ to users, after having a center node c, we need to get the corresponding set $\mathbb{K}$ of matching nodes and set $\mathbb{I}$ of connecting nodes such that they satisfy *Property P3* of Definition 2. We follow the below property.

Property 4 *Given an answer $\langle c, \mathbb{K}, \mathbb{I}\rangle$, if $\mathbb{I}$ is the set of nodes on the* paths *from the center node c to matching nodes in $\mathbb{K}$, then that answer is minimal.*

3.2 Labling and indexing

A. Labeling. Different from conventional labeling schemes, where each node has a distinct label, we only label *object nodes*. All *non-object nodes* are assigned the same label with their corresponding object nodes. This is the feature of object-orientation of our approach. By this labeling scheme, a keyword matching a non-object node is considered as matching the corresponding object node and the number of labels is largely reduced. This brings huge benefits for the efficiency because the search space is reduced. We use *number* instead of Dewey for labeling because in XML IDREF graph, a node can have multiple parents. The other reason is that computation on number is faster than on Dewey since as each component of the Dewey label needs to be accessed and computed. Labels are compatible with the *document order* of the XML document. Besides labeling real nodes in XML document, we also label *virtual nodes*. Each virtual node is also assigned a label which succeeds labels of real nodes. For example, in Figure 2, real object nodes are labeled from 1 to 7, and the virtual node is labeled 8.

Table 1: The ancestor lists for keywords `Cloud` and `XML`

	L^a_{Cloud} (k = Cloud)				L^a_{XML} (k = XML)					
Matching node and its ancestors	*u*	1	2	3	*u*	1	2	3	5	7
Parent of u	*Par(u)*	0	1	2	*Par(u)*	0	1	2	1	5
Descendants of u directly containing k	*Des(u,k)*	2,3	2,3	3	*Des(u,k)*	3,5,7	3	3	3,7	7
Referer of u containing k if u is IDREF node	*IDREF(u,k)*	∅	∅	∅	*IDREF(u,k)*	∅	∅	∅	∅	∅

B. Indexing. Apart from traditional inverted list where each keyword corresponds to a set of matching nodes, to capture both ancestor-descendant relationships and ID/IDREFs in XML IDREF graph, and to facilitate the search, it is necessary to have complex techniques on indexing. A keyword k corresponds to an ancestor list L_k^a and a descendant list L_k^d to facilitate the computation of finding *common ancestors* and finding *common descendants* respectively.

Ancestor list. Each entry in an ancestor list L_k^a is a quadruple which includes:

– u: an object node that matches k or its ancestors. This will be used in finding common ancestors.
– $Par(u)$: the parent of object node u. If u is the root, then $Par(u) = -1$. If the parent of u is the root, then $Par(u) = 0$.
– $Des(u, k)$: the set of descendant object nodes (and itself) of u, which directly contains k. This is used for Property P2.
– $IDREF(u, k)$: the referrer of u w.r.t. k if u is an referred object node. Otherwise, $IDREF(u, k) = \emptyset$. For example, in Figure 2, $IDREF(node\,8, Cloud)$ is $node\,4$. This is used in presenting output when the common ancestor is a referred object node.

For example, Table 1 shows the ancestor lists for keywords `Cloud` and `XML` in the XML IDREF graph in Figure 2, where each entry corresponds to a column in the tables. For instance, the first entry of L^a_{Cloud} corresponds to node 1; the parent of node 1 is node 0 (the root); node 2, 3 are descendants of node 1 containing `Cloud`; and node 1 is not an referred object node.

Descendant referred object node list. Similar to an ancestor list, each entry in a descendant referred object node list L_k^d is a quadruple which includes:

– u: an *referred object node* which is a descendant of an matching object node. This will be used in finding common descendants. Note that a common descendant can only be an referred object node.
– $Child(u)$: set of referred object nodes which are children of u.
– $Match(u, k)$: the set of ancestor object nodes of u, which directly contains k. This is used for Property P2.
– $Path(u, k)$: paths from each node in $Match(u, k)$ to u. This will be used in presenting output.

For example, Table 2 shows the descendant referred object node lists for keywords `Cloud` and `XML` in the XML IDREF graph in Figure 2, where each entry corresponds to a column in the tables. Among object nodes (nodes 1,2,3) matching keyword `Cloud`, only node 2 has referred object node 8 as descendant. Nodes on the path from node 2 to node 8 are 2-4-8.

Table 2: The descendant referred object node lists for keywords `Cloud` and `XML`

	L^d_{Cloud} (k = Cloud)		L^d_{XML} (k = XML)	
IDREF node	*u*	8	*u*	8
Children of **u**	*Child(u)*	∅	*Child(u)*	∅
Matching node	*Match(u,k)*	2	*Match(u,k)*	5
Path from **Match(u,k)** to **u**	*Path(u,k)*	2,4,8	*Path(u,k)*	5,6,8

3.3 Runtime processing

Given a keyword query $Q = \{k_1, \ldots, k_n\}$ to an XML IDREF graph, there are three steps for finding answers to Q, which are (1) finding potential center nodes (common ancestors and common descendant referred object nodes), (2) finding center nodes and (3) generating full answers. This section presents detailed techniques on these steps.

Step 1: finding potential center nodes. Based on Property 1 and Property 2, $Anc(Q)$ and $Ref(Q)$ are potential center nodes, where $Anc(Q)$ and $Ref(Q)$ are the set of common ancestors and the set of common descendant referred object nodes respectively. For each keyword k, we retrieve $Anc(k)$ and $Ref(k)$ from the first field, i.e., the field containing u of the ancestor list L^a_k and the descendant referred object list L^d_k respectively. Let $Anc(Q)$ and $Ref(Q)$ be the set intersection of $Anc(k)$'s and $Ref(k)$'s for all keywords k respectively. The computation of set intersection can leverage any efficient existing algorithms for set intersection. In this work, we use a simple yet efficient set intersection for m *ordered* sets L_1, …, L_m by scanning both lists in parallel, which requires $\sum_i(|L_i|)$ operations in worst case.

Algorithm 1: Finding potential center nodes

```
Input: Ancestor lists $L^a_i$ of keyword $k_i, \forall i = 1..n$
Output: $L^a_c$: list of common ancestors
1  $L^a_c \leftarrow \emptyset$
2  //SetIntersection($L_1, \ldots, L_n$)
3  for each cursor $C_i$ do
4      $C_i \leftarrow 1$
5  $index \leftarrow 1$
6  for each element e in $L_1$ do
7      $cur \leftarrow L_1[e]$
8      $next \leftarrow L_1[e+1]$
9      //Search e in the other lists
10     for each inverted list $L_i$ from $L_2$ to $L_n$ do
11         while $L_i[C_i] < next$ do
12             if $cur = L_i[C_i]$ then
13                 break;
14             $C_i$++;
15         break;
16     //update if e is a common one
17     if $L_i[C_i] < next$ then
18         $L^a_c[index].u \leftarrow e$
19         $L^a_c[index].Par(u) \leftarrow Par(e)$
20         for each keyword k do
21             Add($Des(u,k)$) to $L^a_c[index].Des(u,Q)$
22             Add($IDREF(u,k)$) to $L^a_c[index].IDREF(u,Q)$
```

Algorithm 1 is to find common ancestors. For each common ancestor, we get the corresponding information (line 17-22). The result is illustrated in Table 3, where $Des(u,Q)$ and $IDREF(u,Q)$ contains n components of $Des(u,k)$ and $IDREF(u,k)$ respectively for all keywords k. This figure shows the common ancestor list for query Q = {`Cloud, XML`}. From L^a_{Cloud} and L^a_{XML}, we get $Anc(Cloud)$ = $\{1,2,3\}$ and $Anc(XML)$ = $\{1,2,3,5,7\}$. So, $Anc(Q) = \{1,2,3\}$. For common ancestor 1, $Des(1,Q) = \{2\},\{5\}$ means $Des(1,Cloud) = \{2\}$ and $Des(1,XML) = \{5\}$.

Finding common descendant is similar. However, the differences are the information we get for each potential center node. Particularly, from line 17 to line 22, for each

Table 3: Common ancestors of query {Cloud, XML}

L^a_c

Common ancestor	*u*	1	2	3
Parent of **u**	*Par(u)*	0	1	2
Descendants of **u** directly containing query keywords	*Des(u,Q)*	{2}, {5}	{2}, ∅	{3}, {3}
Referer of **u** if **u** is IDREF node	*IDREF(u,Q)*	∅	∅	∅

common descendant referred object node u, we will update $u, Child(u)$, $Match(u, k)$ and $Path(u, k)$.

Step 2: finding real center nodes. Among potential center nodes, we need to find real center nodes by checking whether they have their own matching nodes. Based on Property 3, $Des(c, k_i) - Content\_Des(c, k), \forall i = 1..n$ is checked bottom up. Initially, each common ancestor c, we can get n sets $Des(c, k_i)$, $i = 1..n$ from the ancestor lists. Among common ancestors $Anc(Q)$, we start from those having no descendant in $Anc(Q)$ (bottom up). They are center nodes. For the parent c' of each new center node c, we update $Des(c', k_i)$, $\forall i = 1..n$ by removing $Des(c, k_i)$ out of $Des(c', k_i)$, $\forall i = 1..n$. Finally, if $Des(c', k_i) \neq \emptyset$, $i = 1..n$, then c' is center node. This is similar for finding common descendant refered object nodes. The list of common ancestors is sorted so that an ancestor occurs before its children. Therefore, to find center nodes from the list of common ancestors, we start from the end of the list of common ancestors because the descendant is then considered first. A considered common ancestor will be filtered out of the list if it is not an center node.

The progress is given in Algorithm 2. If $Des(c, k_i) - Content\_Des(c, k) \neq \emptyset$, $\forall i = 1..n$, then it is a center node. Otherwise, it is filtered out (line 2, 3). For a returned center nodes u, we update the set of exclusive matching descendants of all common ancestors which are ancestors of u (line 5, 6, 7), for each of which, if the set exclusive matching descendants w.r.t. any keyword is empty, we remove it from the set of common ancestors (line 8, 9). Thereby, after finding a center nodes, we proactively filter out a lot of its ancestors if they are not center nodes.

Algorithm 2: Finding real center nodes

```
Input: $L^a_c$: the list of common ancestors
1  for each element e from the end of $L^a_c$ do
2      for each keyword k do
3          if $Des(u, k)$ is $\emptyset$ then
4              $L^a_c$.Remove(e)
5      //Update the sets of exclusive
       matching descendants of COAs
6      for each parent node $e'.u$ of $e.u$ in $L^a_c$ do
7          $v \leftarrow e'.u$ for each keyword k do
8              $Des(v, k)$.Remove($Des(u, k)$)
9              if $Des(v, k)$ is $\emptyset$ then
10                 $L^a_c$.Remove(v)
```

The way to get the list of common descendants is similar to that of common ancestors but we start from the beginning of the list of descendant referred object nodes because an descendant occurs before its parents.

For example, Figure 5 illustrates the way our approach checks whether a potential center node is a real center node or not. First of all, our approach checks nodes 3, 4 and 5 in the most left figure because they have no descendant. After this checking step, descendants containing keywords of node 2 and node 1 is updated. Specifically, we remove n_1 and n_2 for node 2 because they are contained in node 3 and node 4

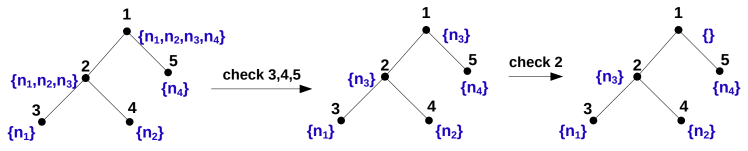


Figure 5: Illustration of checking center nodes

(descendants of node 1), and similarly remove n_1, n_2, and n_4 for node 1. We continue checking until the root is reached.

Step 3: tracking back matching nodes and looking up connecting nodes. Once we have found a center node c, we generate the answer $\langle c, \mathbb{K}, \mathbb{I}\rangle$ accordingly by identifying $\mathbb{K}$ and $\mathbb{I}$. The $Des(u,Q)$ and $Match(u,Q)$ fields allow us to produce the matching nodes $\mathbb{K}$ for answers with simple extension and without affecting space and time complexity. If c is a common descendant, $\mathbb{I}$ can be retrieved by the field $Path(u,k)$. If c is a common ancestor, to find the path from c to a matching node u, we need to find backward from u to c by using Par field in the ancestor lists. Finally, to return the full answers, we need another index from a label to the content of whole matching node.

4 Experiment

This section studies how the features of our approach, including hierarchy-aware, duplication-aware and object-orientation, impact on the performance. We will show the impacts of each feature as well as the impacts of all features on the effectiveness and efficiency. We implemented a system prototype called `XBroad` for evaluation. The experiments were performed on an Intel(R) Core(TM)i7 CPU 3.4GHz with 8GB of RAM.

4.1 Experimental Settings

Datasets. We used three real datasets including **NBA**[4], **IMDb**[5], and **Basketball**[6]. In IMDb, an actor or actress can play in many movies, and a company can produce several movies. In NBA and Basketball, a player can play for several teams in different years. We pre-processed them and used the subsets with the sizes 2.4MB, 90MB and 56MB for NBA, IMDb and Basketball respectively.

Discovering object classes and OIDs of datasets. We first apply [12] to automatically discover object classes and OIDs of datasets. We then manually adjust the results to get 100% of accuracy for the results of discovery. This is to make sure that the discovery step does not affect our results.

Modeling datasets. Each dataset corresponds to four models:

– An X-tree: an XML tree with object duplication and without ID/IDREF.
– An X-graph: an XML IDREF graph obtained from an X-tree.

[4] http://www.nba.com
[5] http://www.imdb.com/interfaces
[6] http://www.databasebasketball.com/stats_download.htm

Table 4: Size and node of X-tree, X-graph, O-tree, O-graph

Dataset	Size (MB)	Number of nodes (thousand)				
		X-tree	X-graph	O-tree	O-graph	Virtual nodes
NBA	2.4	180	215	50	85	35
IMDb	90	48267	49835	18254	19822	1568
Basketball	56	30482	31285	5042	5845	803

- An O-tree (XML object tree): obtained from an X-tree by labelling only object nodes and assigning all non-object nodes the same label with the corresponding object nodes.
- An O-graph (XML object graph): obtained from an X-graph in the same manner with obtaining an O-tree from an X-tree.

Size and node of datasets. The size the three datasets and the number of nodes of their X-tree, X-graph, O-tree and O-graph are given in Table 4. The numbers of nodes of an X-graph, O-graph is the sum of those of the corresponding X-tree, O-tree respectively and the number of virtual nodes.

Queries. We randomly generated 183 queries from value keywords of the three real datasets. To avoid meaningless queries, we retained 110 queries and filtered out 73 generated queries which are not meaningful at all (e.g., queries only containing articles and preposition). The remaining queries include 15, 57 and 38 queries for NBA, IMDb and Basketball datasets, respectively.

Compared approaches. Since `XBroad` can supports both XML documents with and without IDREFs, we compare the performance of `XBroad` with both a tree-based approach (Set-intersection [18]), and XRich [8] and a graph-based approach (BLINKS [5]).

Running compared approaches. Since Set-intersection and XRich work on XML tree, we can run them with X-tree and O-tree. Since BLINK works with XML graph, we ran it on X-graph and O-graph. `XBroad` is also a graph-based approach, thus we can run it on X-graph and O-graph. Since X-tree, X-graph, O-tree and O-graph can be derived from one another with a necessary minor cost and they represent the same information, it is fair to use them together for comparison.

Methodology

Hierarchy-aware, duplication-aware and object orientation are three features which impact the effectiveness and the efficiency of `XBroad`. Among compared approaches, BLINKS does not have concepts of object and hierarchy, Set-intersection does not have the concept of object and duplication, and XRich has all the three similar features, but works on XML tree only. Thus, we have the following methodology to show the impact of each feature and of all features.

Impact of hierarchy-aware. To show the impact of hierarchical structure on graph search, we compared `XBroad` with BLINKS, a non-hierarchy graph-based approach. To separate with the impact of object orientation, we also operated BLINKS at object level, i.e., ran BLINKS on O-graph.

Impact of duplication-aware. For duplication-aware, we compared `XBroad` with Set-intersection, an unaware duplication approach. To separate with the impact of object orientation, we also operated BLINKS at object level, i.e., ran BLINKS on O-tree.

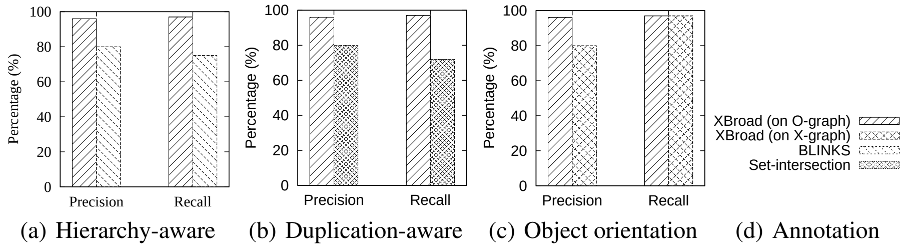


Figure 6: Impact of each feature on the effectiveness [Basketball dataset]

Impact of object orientation. To show the impact of object orientation, we ran `XBroad` at object level (i.e., O-graph) as well as at node level (i.e., on X-graph).
Impact of all three features. To show the impact of all features, we ran compared algorithms on the data they initially designed for. Particularly, we ran BLINKS on X-graph, Set-intersection on X-tree, and `XBroad` on O-graph. Besides, we also compare with XRich because XRich has three similar features.

4.2 Effectiveness Evaluation

Metrics. To evaluate the effectiveness, we used standard *Precision* ($\mathcal{P}$) and *Recall* ($\mathcal{R}$) metrics. We randomly selected a subset (20 queries) of 110 generated queries for effectiveness evaluation. To compute precision and recall, we conducted surveys on the above 20 queries and the test datasets. We asked 25 researchers of our database labs to interpret 20 queries. Interpretations from at least 18 out of 25 researchers are manually reformulated into schema-aware XQuery queries and the results of these XQuery queries are used as the ground truth.
Impact of each feature. Figure 6 shows the impacts of each feature on the effectiveness. As can be seen, each feature can help to improve effectiveness. Follows are the reasons of improvement w.r.t. each feature. The hierarchical structure enables us to avoid meaningless answers caused by unrelated matching nodes. BLINKS uses the distinct root semantics which is similar to the LCA semantics, thus its recall is affected by the problems of not returning common descendants. Duplication-aware help return

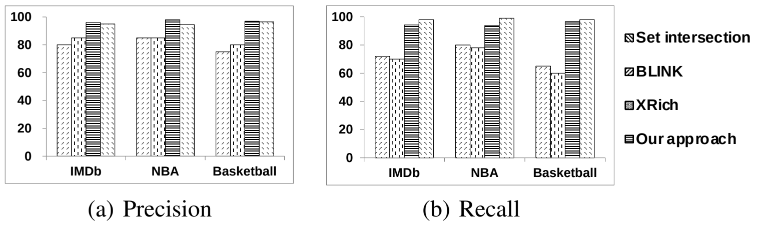
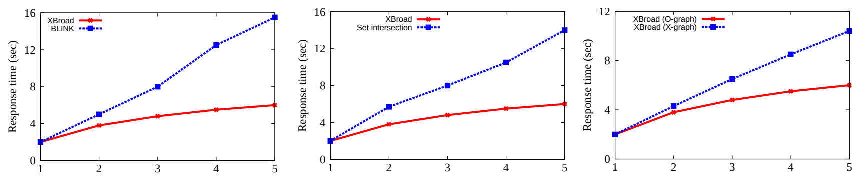


Figure 7: Impact of all features on the effectiveness



(a) Hierarchy-aware (b) Duplication-aware (c) Object-orientation

Figure 8: Impact of each feature on the efficiency [Basketball dataset]

missed answers. Object orientation improves precision because it enables us to avoid meaningless answer caused by returning only non-object nodes.

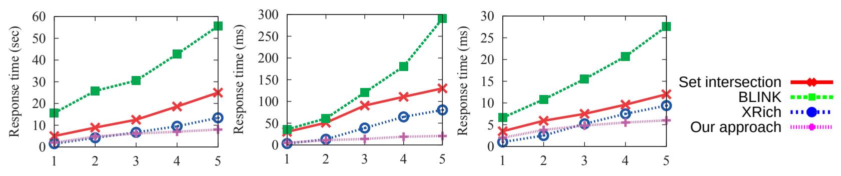
Impact of all features. Figure 7 shows the impacts of all features on the effectiveness. As discussed above, all these features have impacts on the effectiveness. Among them, duplication-aware has the highest impact. Thus, the more features an approach possesses, the higher precision and recall are. Particularly, `XBroad` has highest precision and recall because `XBroad` has all three features. BLINKS has higher performance than Set-intersection because BLINKS works with graph, a duplication-aware data. Compare to XRich, our approach has higher recall while the precision is similar.

4.3 Efficiency Evaluation

Metrics. To measure the efficiency, we compared the running time of finding returned nodes. For each kind of queries, e.g., 2-keyword query, we selected five queries among 110 retained queries sharing the same properties. For each query, we ran ten times to get the average response time. We finally reported the average response time of five queries for each kind of query.

Impact of each feature Figure 8 shows the impact of each features on the efficiency. As can be seen, all features improve efficiency, among which, hierarchy has the most impact. The reasons for the improvement are follows. Hierarchy enables us to avoid NP-Hard problem of the general graph search and it just extends visited nodes to two directions: ancestor and descendant rather than to all directions. Duplication-aware and object orientation both help reduce the search space.

Impact of all features The response time of algorithms is shown in Figure 9, in which we varied the number of query keywords. As discussed above, all features impact the ef-



(a) IMDb (b) NBA (c) Basketball

Figure 9: Impact of all features on the efficiency (varying number of query keywords)

ficiency. Thus, the more features an algorithm has, the more efficient it is. `XBroad` obtains the highest efficiency because it has all three features. Among the others, BLINKS is the least efficient because it is affected by structure which does not have hierarchy. Compared to XRich which based on tree-structure, our approach can get almost the similar response time, even better when the number of keywords is increased.

5 Related work

LCA-based XML keyword search. XRANK [4] proposes a stack based algorithm to efficiently compute LCAs. XKSearch [17] 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 [19] introduces the concept of valuable/exclusive LCA to improve the effectiveness of SLCA. MaxMatch [14] investigates an axiomatic framework that includes the properties of monotonicity and consistency. Although extensive works have been done on improving the effectiveness of LCA-based approaches, these works commonly still suffers from the problem of missing answers because of undetected object duplication in XML document. Moreover, these works only can work with XML documents with no IDREF. Recently, XRich [8] takes common descendants into account of answers. However, its input XML document must not contain IDREF or n-ary relationships ($n \geq 3$).

We generalize the technique of set intersection [18] in processing queries. However, there are several differences. First, that work considers XML tree where a node has only one parent whereas ours can deal with the case where a node has multiple parents by employing more complex indexes and searching techniques for XML IDREF graph. Secondly, that work operates at node level whereas we operate at object level, which enables us to improve both efficiency (by reducing search space) and effectiveness (by avoiding meaningless answers) of the search. Most importantly, that work only search up to find common ancestors but miss common descendants because it cannot detect object duplication. In contrast, we have techniques to find missing answers.

Graph-based XML keyword search. BANKS [1] uses backward search to find Steiner tree in labeled, directed graph. Later, Bidirectional [6] improves BANKS by using bidirectional (backward and forward) search. EASE [11] introduces a unified graph index to handle keyword search on heterogeneous data. Without the exploiting hierarchical structure of XML graph, the graph search in general suffers from NP-Hard Problem. Moreover, since XML graph (with IDREF) can contain object duplication, but these works cannot detect object duplication. Thus they may also miss answers. To the best of our knowledge, only [9] can provide the such missing answers. Nevertheless, this work transfers XML 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.

6 Conclusion

We introduced an approach to handle the problem of missing answers due to object duplication for keyword search in a data-centric XML document. We model the input XML document as an XML IDREF graph where all instances of the same object are connected via a virtual object node (duplication-aware). We only work with object nodes and associate non-object nodes to the corresponding object nodes (object-orientation). More importantly, we discover the hierarchical structure of XML IDREF graph to inherit LCA-based techniques for an efficient search (hierarchy-aware). The experiments showed the impact of each and all feature(s) (duplication-aware, hierarchy-aware, object-orientation) to the efficiency and effectiveness, which made our approach outperforms the compared approaches in term of both efficiency and effectiveness.

References

1. G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.
2. S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. *Networks*, 1971.
3. J. Fong, H. K. Wong, and Z. Cheng. Converting relational database into XML documents with DOM. *Information & Software Technology*, 2003.
4. L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.
5. H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD*, 2007.
6. V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, and R. D. Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.
7. M. Kargar and A. An. Keyword search in graphs: finding r-cliques. *PVLDB*, 2011.
8. T. N. Le, T. W. Ling, H. V. Jagadish, and J. Lu. Object semantics for XML keyword search. In *DASFAA*, 2014.
9. T. N. Le, H. Wu, T. W. Ling, L. Li, and J. Lu. From structure-based to semantics-based: Effective XML keyword search. In *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. In *DEXA*, 2013.
13. Y. Li, C. Yu, and H. V. Jagadish. Schema-free XQuery. In *VLDB*, 2004.
14. Z. Liu and Y. Chen. Reasoning and identifying relevant matches for XML keyword search. In *PVLDB*, 2008.
15. Y. Tao, S. Papadopoulos, C. Sheng, and K. Stefanidis. Nearest keyword search in XML documents. In *SIGMOD*, 2011.
16. A. Termehchy and M. Winslett. EXTRUCT: using deep structural information in XML keyword search. *PVLDB*, 2010.
17. Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *SIGMOD*, 2005.
18. 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.
19. R. Zhou, C. Liu, and J. Li. Fast ELCA computation for keyword queries on XML data. In *EDBT*, 2010.