THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TRA7/10

Object-Oriented XML Keyword Search

Huayu Wu, Tok Wang Ling, Zhifeng Bao and Liang Xu

July 2010

T e c h n i c a l R e p o r t

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

Object-Oriented XML Keyword Search

Huayu Wu, Tok Wang Ling, Zhifeng Bao, and Liang Xu

School of Computing, National University of Singapore
{wuhuayu, lingtw, baozhife, xuliang}@comp.nus.edu.sg

***Abstract*— Keyword search is a user-friendly way to query XML data. Existing LCA-based XML keyword search approaches assign a Dewey ID to every document node and find the relevant LCA nodes of the query keywords based on the Dewey IDs. Despite many improved LCA-based semantics proposed, these approaches are still not as effective as expected. We observe that object is an important concept for returning meaningful information in database queries. In this paper, we propose an object-oriented approach for XML keyword search. Our approach only assigns different Dewey IDs to object nodes in the document, and introduces relational tables to organize data values. By considering the semantic relationship among object, property and value during keyword query processing, our approach significantly improves the search efficiency and quality, compared to existing work. Furthermore, after finding any object as a return node, our approach only outputs the useful information about that object, instead of the whole subtree rooted at the object node as in many other approaches. Finally we design an algorithm to rank the possible interpretations of an ambiguous query, and the search results are returned separately based on different query interpretations. The efficiency and effectiveness of our approach are demonstrated with a comprehensive experimental study.**

I. Introduction

XML keyword search provides a very flexible and convenient way for user to query XML databases. Thus, keyword query processing in XML documents attracts lots of research interests. Due to the hierarchical structure of XML data, the research focus of XML keyword search is how to discover the structural relationship between query keywords appearing in a document. The most common way to process XML keyword queries is based on *lowest common ancestor* (*LCA*) [1]. The initial LCA semantics may return many meaningless results. For example, one of the LCA answers of the keywords "book" and "title" in the document in Fig. 1 is the document root 1, because 1 is the lowest common ancestor of the book node 1.1.2.1 and the title node 1.2.2.1.1. To improve the search quality, different LCA-based semantics are proposed. For example, SLCA [2] returns the LCA nodes of a set of keywords, which do not have any descendant to also be an LCA of the given keywords. Thus the SLCA of "book" and "title" in Fig. 1 does not return the root node 1 as it has descendants 1.1, 1.1.2.1 and so on, to be the LCAs of the keywords. There are also many other works, which are reviewed in Section VI.

Although researchers put many efforts to improve the efficiency and quality of XML keyword search, there are still a number of problems that have not been well solved.

Efficiency Problems:

P1: Object-property containment. Different from HTML documents, tags in XML documents are not only for formatting purpose, but also for information description. Thus, tag labels often appear in XML keyword queries. Due to the hierarchical structure of XML data, document nodes may be in containment relationship, especially for object node and property node. For the example document in Fig. 1, every "book" node is an object node that contains a property node "title". In real-life queries, keywords are often matched by the document nodes that are in containment relationship, such as the query {book, title}. Since the LCA of an object node and a property node that are in a containment relationship must be the object node, in some cases (e.g. every book has a title) we actually do not need to scan the inverted list for property during LCA-based computation.

P2: Property-value constraint. The inverted list for a keyword may contain lots of useless Dewey IDs with respect to a certain query. For example, consider a query {book, title, XML} to find the books with title of "XML" over the document in Fig. 1. Suppose there are 100 *title* elements, then we have to consider all 100 Dewey IDs in the inverted list of *title* during LCA searching, though only one of them matches the child value of "XML". Some work ([2]) employs index on inverted list to speed up the search, and some work ([3]) introduces new semantics to avoid redundant computation. However, none of them solves the above example, in which the redundant search is caused by value constraints.

Effectiveness Problems:

P3: Output problem. The return information of many approaches is the whole subtree rooted at each relevant LCA that was found. Actually in many cases the user is only interested in the information of the common object of several nodes, rather than other irrelevant information in the subtree. Take the query {Computers, XML} as an example, which aims to find the common information of the two books. Although some work ([4]) could identify the return information should be "subject", they still cannot decide whether the whole subtree rooted at "subject" or only the property value of "subject" should be returned. A recent work [5] addressed the problem of identifying relevant information in the subtree of LCA nodes, but their method is not precise in many cases, e.g. for a query {book, title, XML} to find the information (including author, quantity, etc) of the book "XML", their method only returns book-title-XML path.

P4: Keyword ambiguity. Many keyword queries have different

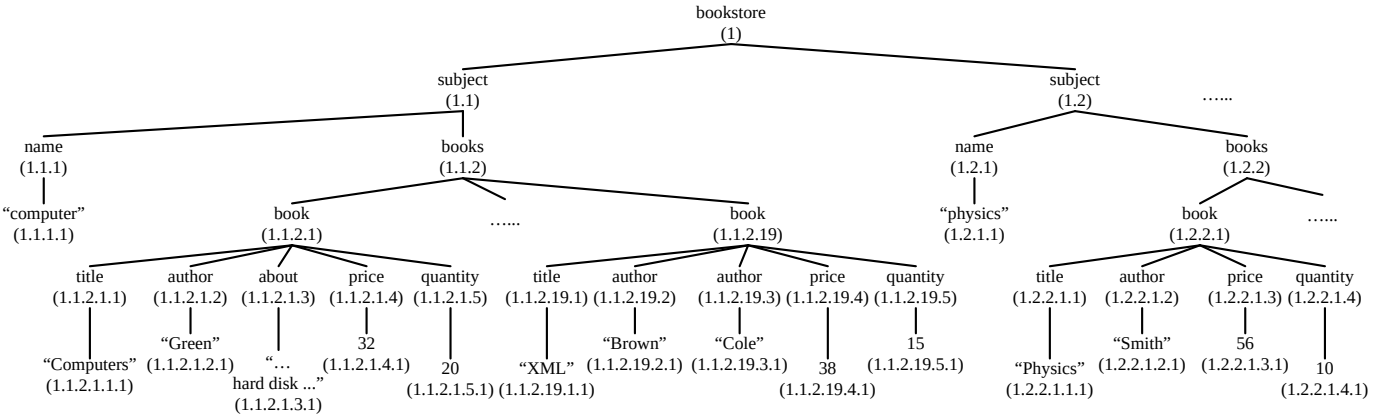


Fig. 1. An example XML document with Dewey IDs for all nodes

interpretations, or say search intentions. A query {physics} may search for a book with title of "Physics", or a subject with name of "physics". Identifying search intention ([6], [7]) and returning diverse query results ([8]) are studied in text-based search, but those algorithms are not applicable to XML keyword search which focuses on structure-based search, e.g. LCA. Most existing XML keyword search approaches mix different interpretations during query processing, and the user may have difficulty filtering the mixed returned results based on his real search intention.

The main reason for the above problems is that the existing approaches do not consider the semantic relationship, e.g. object-property or property-value relationship, between keywords. Thus they either perform redundant search or return less meaningful results. Moreover, existing approaches are not efficient to support advanced search, such as range search and phrase search, though these advanced search features are highly expected by users.

In this paper we propose an object-oriented approach to efficiently and effectively process XML keyword queries. We observe that most XML keyword queries aim to find information about object(s), thus by performing LCA-based computation on object level we not only avoid the inverted list scan for non-object keywords, to improve search efficiency, but also find and return object information, to improve search quality. In more details, we only assign different Dewey IDs to the object nodes in a document. All the property nodes inherit the Dewey ID from their associated object nodes, and the inverted list for each property is organized according to the associated object. For example, in some document we put the Dewey IDs for "department" *name* and "employee" *name* separately in two inverted lists. When a query searches for department names, we do not need to access the inverted list of "employee" *name*, by which we not only improve the efficiency, but also avoid some potential false positives. If a query is vague and we are not sure which *name* is interested by the user, we can process the two interpretations separately using the disjoined inverted lists, rank the two sets of results, and return to the user. We can solve the problems P1, P2 and P4 using this approach.

Another technique used in our object-oriented approach is that we adopt relational tables to store each value with the Dewey ID of its associated object and the information about which property it belongs to. Using relational tables we can efficiently support the advanced search (details discussed in Section IV-E). Furthermore, since the relational table is also object oriented, e.g. for each object class there is a relational table in which the properties and values for each object in this class are stored, we can easily extract useful return information of each object, without outputting irrelevant document nodes as mentioned in P3.

The contributions of our work are summarized as follows:

- We propose an object-oriented concept in XML keyword search. Processing XML keyword search at object level not only improves the efficiency by reducing the number of inverted lists and the Dewey IDs in each inverted list to compute LCAs, but also improves the search quality by avoiding meaningless output.
- We introduce relational tables to organize property values. This structure enables range search and phrase search for values, which is not easy or not efficient to achieve in existing approaches. Also the relational table for each object helps to output useful object information.
- We design a mechanism to analyze a keyword query to discover the relationships among object, property and value keywords, and design an algorithm to rank different search intentions of an ambiguous query.
- We conduct experiments to compare the query processing efficiency and search quality between our object-oriented approach and the existing approaches. The result shows the superiority of our approach.

The rest of the paper is organized as follows. Background knowledge is presented in Section II. In Section III we discuss the object-oriented indexes used in our approach. The algorithms to process keyword queries in our approach are presented in Section IV. We show the experimental results in Section V and review the related work in Section VI. Finally we conclude this paper in Section VII.

II. BACKGROUND

A. XML keyword search and LCA

XML keyword search aims to find the relevant information of a set of query keywords, which is normally some document nodes with those keywords appearing in their subtrees. Different from IR search, XML keyword search focuses on exploring hierarchical structure of XML data, thus it normally works on data-centric documents. LCA-based approach is a common way to process XML keyword queries. The Lowest Common Ancestor (LCA) of a set of nodes *S* is defined as the common ancestor of the nodes in *S* which does not have a descendant node to also be a common ancestor of these nodes. In an XML document, normally we assign a Dewey ID [9] to each node and the LCA of a set of nodes has the Dewey ID of the longest common prefix of the Dewey IDs for these nodes. For example, in the document in Fig. 1, the LCA of the node 1.1.2.1.3 and the node 1.1.2.19.1 is the node 1.1.2.

B. Inverted list and SLCA

In XML keyword search, normally for each keyword, the Dewey IDs of all the document nodes matching that keyword are stored in an inverted list. A document node *matches* a keyword if its tag label or text value has the same string name as the keyword. XML keyword search focuses on finding the relevant LCAs of the inverted lists of all query keywords. A node is an LCA of a set of inverted lists $\{I_1,..., I_m\}$ if this node is the LCA of $\{u_1,...,u_m\}$ where $u_i \in I_i$ for $1 \leq i \leq m$.

Example 2.1: The inverted lists for the keywords in the query {book, XML, author} are shown in Fig. 2. The LCAs of the three lists include 1, 1.1.2, and 1.1.2.19 etc. Particularly node 1 is the LCA of *book* 1.1.2.1, *XML* 1.1.2.19.1.1 and *author* 1.2.2.1.2.

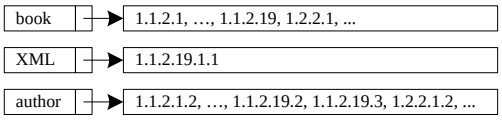


Fig. 2. Example inverted lists

Intuitively the correct answer to the query in Example 2.1 is book 1.1.2.19, but LCA returns quite a lot of false positives, e.g. node 1. To achieve a good search quality, many improved semantics based on LCA are proposed [10], [2], [11]. In our approach we propose a new semantics SLCOA (discussed later) based on SLCA [2].

The SLCA of a set of inverted lists is the LCA node of these inverted lists which has no descendant to also be an LCA of these lists. In Example 2.1, the SLCA of the inverted lists only returns 1.1.2.19, which is the correct answer.

III. OBJECT-ORIENTED INDEXES

A. Object in XML

Definition 3.1: (**Object, Property**) An object (or entity) is a unit to model a person, thing, concept, or event about which a system needs to manage information. Each object has several properties (or attributes), to describe the object from different aspects.

Object is an important information unit in structured data management (e.g. ER model). In XML databases, object also plays an important role because most XML queries aim to find either the information of a certain object based on the conditions on its properties, or the relationships between a set of objects.

Despite the importance of object in data management, most XML keyword search algorithms do not take it into account during query processing. Although some works (e.g. [4]) mention *object* when identifying output information, how to combine the object concept to query processing to enhance the efficiency and search quality is still not well investigated.

Normally object information can be provided by the designer of an XML database. In case such information is not available, object can also be inferred in some other ways. For example from DTD we can consider an element as an object class if it contains ID attribute; [12] designs a semantic rich schema for XML data in which object class are explicitly specified; [13] and [14] discover the semantics such as keys and functional dependencies in XML document, which is helpful to identify objects. Also object can be inferred by interacting with the user, which is widely adopted by many web-based information management systems [15], [16].

All the existing object inference techniques can complement our work. To be self-contained, when no object information is available we consider:

- The parent node of each value node is a **property** node.
- The parent node of each property node is an **object** node.

In Fig. 1, without object information known, we consider that the nodes "title", "author" and so on are properties as they are parents of values, and the nodes "book" and "subject" are objects as they are parents of properties. Although this inference is not always correct, e.g. a composite property may be considered as an object, using such information our approach is not worse than other approaches which do not take object and property into account, in search quality. For example, if *name* is composite property with *firstName* and *lastName*, our approach may inaccurately consider a *name* node as an object with properties *firstName* and *lastName*, and return it for a keyword query {Jim}. However, other LCA-based approaches may simply return a value node *Jim*, as it is the LCA of itself. This is even less meaningful than our approach. Also in our OO approach, the query processing efficiency can be improved, no matter the object inference is accurate or not, as demonstrated in Section V-C. Most of all, when **more precise information on object is known**, our approach can build the object-oriented indexes according to the known object, e.g. integrating composite property with the associated object, so that more non-object inverted lists are skipped and more accurate object information is returned. Then we can process queries **more efficiently and more effectively**.

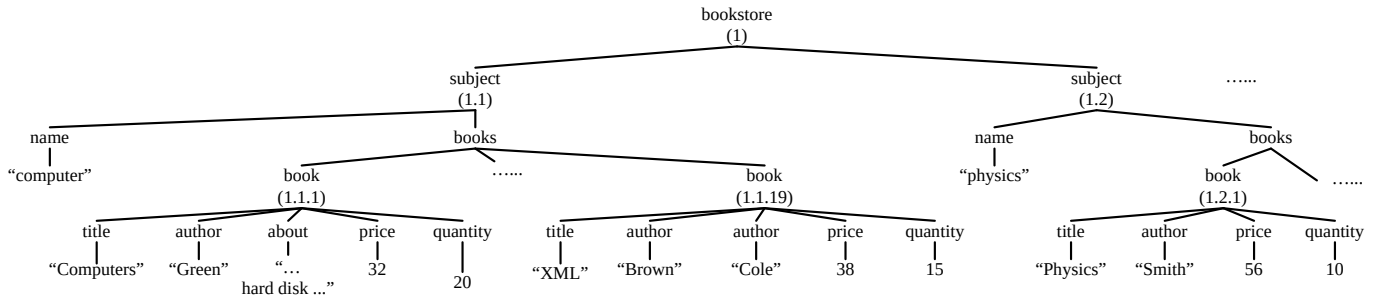


Fig. 3. Document with OO-Dewey ID assignment (only object nodes are labeled)

B. OO-Dewey ID and object tables

Existing LCA-based XML keyword search approaches assign each document node, including tags and values, a different Dewey ID, as shown in Fig. 1. In our object-oriented approach, we assign Dewey ID to object node only, thus we also call it OO-Dewey ID[1]. Fig. 3 is the OO-Dewey ID assignment for the document in Fig. 1. Since we identify two object classes: *subject* and *book*, we only assign Dewey IDs to *subject* and *book* nodes (as well as the root). All non-object internal nodes, e.g. property nodes, will inherit the Dewey ID of the its lowest ancestor object node.

In existing approaches, inverted lists are built for each type of keyword. For document tags, the keyword is the tag label, and for leaf values, the keyword is a number or a word within the value text. The tremendous number of different keywords not only brings the difficulty in inverted list management, but also affects the query efficiency and search quality. In our approach, we put the Dewey IDs for only non-value document nodes into inverted lists, and all the inverted lists are built in object-oriented fashion. In particular, for each object keyword we create an inverted list, while for each property keyword, the inverted list is built by both the property and the associated object. For example, if the bookstore document also contains object class *CD* with a property *title*, then the Dewey IDs for different *title*s are stored in separate inverted lists: one is for book/title and the other one is for CD/title. The object-oriented inverted lists for "book", "author" and "about" are shown in Fig. 4. Note that since we only use object labels, many properties with cardinality of 1 or + to its associated object, e.g. "author", will have exactly the same inverted list as the object. In this case, we only physically store the inverted list once. For optional property, e.g. "about", whose inverted list is a subset of its associated object's inverted list, we keep its inverted list separately. The two cases are both reflected in Fig. 4.

Value nodes are put into relational tables. The relational tables are also object-oriented, which means for each object class we maintain a table with columns of Dewey ID and its single-valued properties. We call it *object table*. For multi-valued property, e.g. "author", we maintain an object/property table. The example tables for the object "book" in the document in Fig. 3 are shown in Fig. 5. If the ordinal information of a multi-valued property is important, e.g. the author order of a book, we can also introduce a new column in the object/property table to indicate the order of the properties under a same object.

[1]For convenience, we still use *Dewey ID* for *OO-Dewey ID* in later explanations.

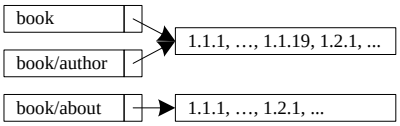


Fig. 4. OO inverted list example

R_{book}

OO-Dewey	Title	About	Price	Quantity
1.1.1	Computers	… hard disk …	32	20
…	…	…	…	…
1.1.19	XML	null	38	15
…	…	…	…	…
1.2.1	Physics	null	56	10
…	…	…	…	…

$R_{book/author}$

OO-Dewey	Value
1.1.1	Green
…	…
1.1.19	Brown
1.1.19	Cole
…	…
1.2.1	Smith
…	…

Fig. 5. Object tables for "book"

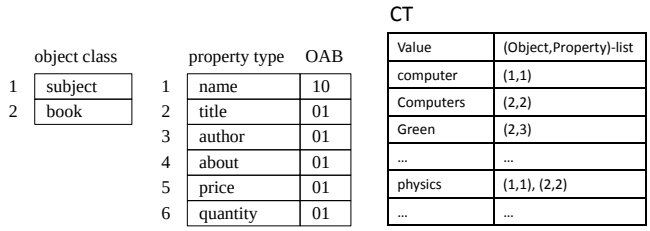
One major difference between inverted list and relational table to manage values is that relational table can extract Dewey IDs based on values and vice versa; but inverted list cannot get values from Dewey IDs, so it cannot help for result extraction. Also relational table offers an opportunity to perform range search and phrase search as discussed in Section IV-E

C. Other object-based indexes

First we assign each type of object and property a numeric ID. The ID of each object class and property type for the document in Fig. 3 is shown in Fig. 6(a) (we do not consider the root as an object). We have a hash table to check whether a keyword refers to an object or a property, and return the numeric ID correspondingly. We maintain an **object attachment bitmap** (***OAB***) for each property type, and a **containment table** (***CT***) for all values.

Object Attachment Bitmap (OAB). The *OAB* for each property type is a bitmap indicating by which object classes it is contained. If an object class with ID i contains a property

type *prop*, the *i*-th position of the *OAB* for *prop* is set to 1, otherwise it is 0. The *OAB*s for different property types in the bookstore document are also shown in Fig. 6(a). Given a property type we can quickly find what object classes contain it through the *OAB*.



	object class
1	subject
2	book

	property type	OAB
1	name	10
2	title	01
3	author	01
4	about	01
5	price	01
6	quantity	01

(a) Numeric IDs and OABs

CT

Value	(Object,Property)-list
computer	(1,1)
Computers	(2,2)
Green	(2,3)
...	...
physics	(1,1), (2,2)
...	...

(b) CT

Fig. 6. OAB and CT examples for the document in Fig. 3

Containment Table (CT). The *CT* stores each value and the information of what objects and properties contain it. The containment information of each value is expressed by a list of (object, property) pairs. The example *CT* for the bookstore document is shown in Fig. 6(b). In this *CT*, the containment information for value "physics" reflects that both (1, 1) and (2, 2) contain this value, which are (subject, name) and (book, title) respectively.

The object tables and the CT are easy to maintain. Any update only results in tuple modification, without affecting other table records.

IV. OO KEYWORD QUERY PROCESSING

The object-oriented keyword query processing in our approach has four steps:

1) Partition the keywords in the query based on different objects. Ambiguous queries with different keyword attaching ways are handled by query splitting, and after that all the interpreted queries are ranked.
2) For each partition, filter the Dewey IDs in the inverted list for that partition, based on the properties and values in it.
3) Find the smallest lowest common object ancestor (SLCOA) of the inverted lists for the objects in different partitions.
4) Identify output information and return results using corresponding object tables.

We describe each step in details as follows:

A. Step 1: Keyword partitioning

Definition 4.1: (**Keyword Partition**) A partition is a group of keywords in a keyword query that contains one object[2], together with a set of properties or property/value pairs that belong to the object. We use the format (*object, [property$_1$[/value$_1$], property$_2$ [/value$_2$]...]*) to represent each partition.

[2]We simply refer to object (or property) keyword as object (or property), for short.

In a keyword query, we create a partition for each object involved. For example, there are two partitions for the query {subject, XML}, as there are two objects, *subject* and *book*, are involved. We put the query keywords into the corresponding partitions, e.g. the value keyword "XML" belongs to the partition for *book* as it is a title value of a book. This process is called keyword partitioning. The partitioned query in this example becomes {(subject), (book, title/XML)}. We design a model-driven mechanism, shown in Fig. 7, to partition query keywords based on objects.

We initialize an undone set with the given keyword query. As long as there are queries in the undone set, we pick a query to process, and put the processed query into the done set. There are two phases when we partition the keywords in a query: attaching the value keywords to properties and objects, and attaching the unattached property keywords to objects. In each phase, the attaching way is possibly not unique. In this case, we split the query by different attaching ways, pick one of them to continue processing, and put others into the undone set.

Now we discuss how to attach a value keyword to the corresponding property and object. Basically, for each value keyword we use the *CT* to find whether there are any property and object keywords in the query containing it. If so, we attach the value to the property and put the pair into the partition for the object. In case there is no explicit property, or object keyword in the query containing the value keyword, we can still find the implicit attachable property and object with *CT* and *OAB*. After attaching all values, we attach the property keywords which are not bound by any values, to objects. Similarly for each such property keyword, we check if any object query keyword, or any object in existing partition contains it. If so, we put it into the partition of that object. Else we get the object of the property from the property's *OAB*. Due to the space limitation, the implementation details are omitted. Note that when there are multiple attaching way for either value keyword or property keyword, we consider the query is ambiguous. We discuss the ambiguous case in details later.

A special case is that multiple value keywords correspond to the same property of the same object. In this case we create partitions based on the occurrences of the property under that object: (1) if the property is multi-valued, we use one partition for the multiple property-value pairs, and (2) if the property is single-valued, we use separate partitions for the multiple property-value pairs.

Example 4.1: Consider the document shown in Fig. 3. The query {book, title, physics} is unambiguous. We attach the value *physics* to the property *title*, and then put the pair into the partition for the object *book* as {(book, title/physics}. Similarly the query {subject, physics} is also unambiguous as the explicit object keyword *subject* contains the value *physics*. After partitioning, the query becomes {(subject, name/physics)}.

Example 4.2: In the query {Brown, Cole}, both value keywords belong to the same object and property, i.e. book/author. Since *author* is a multi-valued property of *book*, the query

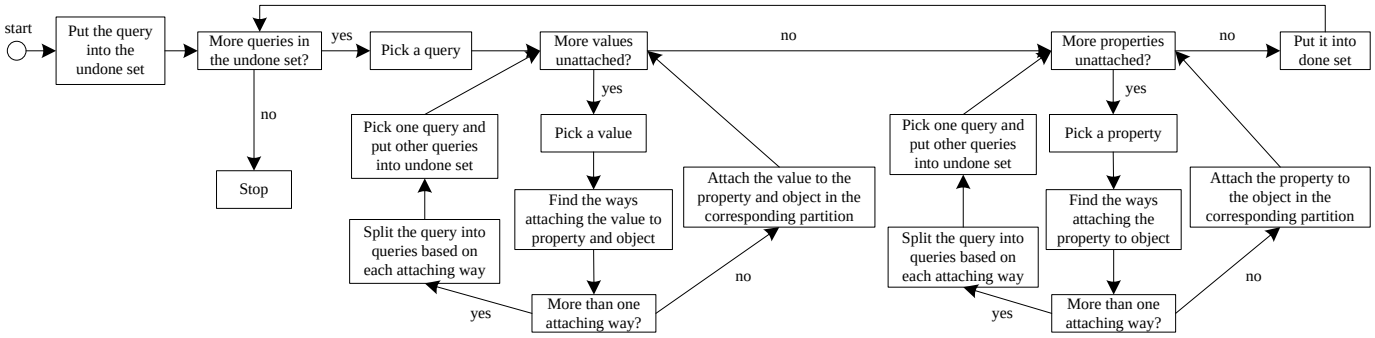


Fig. 7. Keyword partitioning process model

is partitioned as {(book, author/Brown, author/Cole}, which aims to find the common book written by the two authors. The value keywords in the query {Computers, XML} also belong to the same object and property, i.e. book/title, but *title* is a single-valued property of *book*. Then the keywords in the query fall into two separate partitions as {(book, title/Computers), (book, title/XML)}. This query aims to find common information (subject) of the two books.

1) Intermediate Tag: In XML document, some tags correspond to neither an object nor a property. We call them intermediate tags. Generally, a meaningful intermediate tag is either a composite property or a grouping node to group a set of objects. We discuss how to extend our algorithm to search intermediate tags.

For composite property, e.g. *name* which is composed of first_name and last_name, we can combine it in object tables. Then a *person* table contains fields *name/first_name* and *name/last_name*. Also *OAB* is extended to link *name* to *person*. A query searching for person names returns all property fields beginning with *name* in the *person* table.

When the intermediate tag is a grouping node, e.g. *books* in the bookstore document, we just maintain an inverted list for this tag type and associate it a list of object classes that are contained by this tag type. A query searching for a grouping node will visit all the tables of the objects contained by it, to return tuples whose Dewey ID prefix is the Dewey ID of this grouping node. For example, if a query searches for *books* with subject of computer, after finding *books* 1.1, all tuples in the *book* table (as *book* is the only object class contained by *books*) whose Dewey ID begin with 1.1 are returned.

In the rest of the paper, we will not explicitly mention the intermediate tag. Any query node corresponding to an intermediate tag is considered as a property node.

2) Ambiguous Case: All the ambiguous cases appearing in each pass mentioned above are because of the multiple attaching ways, or say multiple interpretations of query purpose. For these ambiguous cases, we split the current query based on different interpretations, and execute the new queries separately. Interpretations are ranked and then search results are returned to the user.

Actually the number of interpretations of an ambiguous XML keyword query is normally quite limited in real life. We investigate 11 real-life XML data sets and 1 XMark benchmark data set, and discover that the average and maximum ambiguity degree (i.e. the number of attaching ways of a given keyword on average) of property keyword are 1.18 and 2.37, and the average and maximum ambiguity degree of value keyword are 1.05 and 1.22[3]. We also gather the top 500 popular real-life keyword queries[4], and count the average and maximum keywords in a query is 1.6 and 4. Then in our collection, the average and worst-cased number of interpretations of a random query are 1.18*1.05*1.6=1.98 and 2.37*1.22*4=11.56 respectively, which are computed by multiplying the number of value keywords and the attaching ways from each value to property and from each property to object.

Although normally the interpretations of an ambiguous real-life query is limited, we still handle the case that some queries are really "bad" with too many different interpretations. In this case, we do not distinguish different interpretations, but return one set of mixed result as other approaches do. The different attaching way of each ambiguous keyword will result a union of relevant inverted lists, and LCA-based computation is performed on the inverted lists of all query keywords. Before splitting a query, we estimate the total number of splitting ways by multiplying the number of attaching ways for each value and property keyword, and set a threshold (depending on needs) to decide whether to split it or not.

Example 4.3: In the bookstore document, a query {subject, book, physics} is ambiguous, as the value *physics* can be attached to both *subject* and *book*. Then, the query is split into two queries with different partitions as {(subject, name/physics), (book)} and {(book, title/physics), (subject)}. The first query is to find the books under subject of physics, while the second query has the purpose of finding the subject of the book physics. The two interpretations are both reasonable. Other approaches cannot return two sets of results based on the two interpretations.

[3]The experimental data sets and detailed results are available from http://www.comp.nus.edu.sg/∼wuhuayu /amb_test.htm.

[4]From http://www.mikes-marketing-tools.com/keywords, which gathers information from a database of over 330 million search terms extracted from popular metacrawlers.

3) Interpretation Ranking: After splitting an ambiguous query into several possible queries based on different interpretations, we rank them and return separate result for each interpretation, so that the query issuer could retrieve more accurate result based on his real search intention.

We design the interpretation ranking method based on two assumptions:

A1: The query with fewer objects is considered more likely to be asked.

For example, the query {physics, Smith} can be interpreted in two ways based on our partitioning: {(subject, name/physics), (book, author/Smith)} and {(book, title/physics, author/Smith)}. Then we consider the second interpretation is more likely to be the search intention.

A2: The query with objects that appear closer in the document is considered more likely to be asked.

Consider a portion of an XML document in Fig. 8. The query {head, Smith} can be interpreted as {(department, head), (project, leader/Smith)} and {(club, head), (project, leader/Smith)}. Both the interpretations contain two objects, but the relationship between *department* and *project* in the first one is clearly closer than the relationship between *club* and *project* in the second one. We consider the first interpretation is more likely to be the search intention.

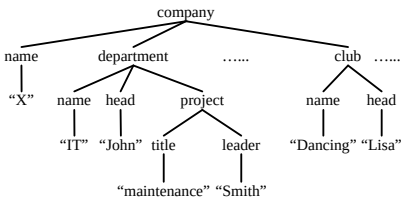
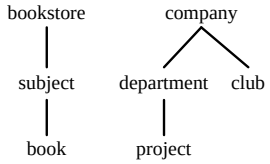


Fig. 8. A portion of an example XML document

It is trivial to check the number of objects involved in a query. Now we model the distance of the objects in an XML document. In our approach, we construct an *object tree* either from DTD or during document parsing. Object tree is similar to structural summary [17], to summarize the structural information of an XML document, without considering data values; but the difference is object tree involves only object nodes, without other types of nodes, e.g. property. Two example object trees are shown in Fig. 9(a). If all objects in a query are different, we measure the distance between them by finding the number of edges in the minimal spanning tree of all these objects in the object tree. However, if two query objects are identical, e.g. the two *book* objects in the partitioned query {(book, title/Computers), (book, title/XML)}, we consider the distance between them is 2 instead of 0, though they correspond to the same node in the object tree. The reason is that actually we have to at least walk through a common ancestor of the two objects to connect them in the document.

The detailed ranking algorithm is shown in Algorithm 1. There are two steps to find the size of the minimal spanning tree of a set of query objects in an object tree *OT*. The first



(a) Object trees for the documents in Fig. 3 and 8

$R_{subject}$

OO-Dewey	Name
1.1	computer
…	…

(b) Object table for *subject*

Fig. 9. Example object trees and an object table

Algorithm 1: Ranking possible queries

```
1  OT := the object tree of the XML document;
2  foreach possible query Q do
3      S := the set of different objects in Q;
4      foreach v ∈ S do
5          find the length of the shortest path from v to other object nodes in S by
           computing Dijkstra(OT, v);
6      construct the complete graph G_c with S and computed weights between
       every two object nodes in S;
7      x := any object node in G_c as a starting node;
8      T := {x};
9      w := 0;          /* w is the total weight of the minimum
       spanning tree of G_c */
10     while |T| < |G_c| do
11         find edge (u,v) from G_c with minimal weight such that u is in T and v
           is not;
12         T := T ∪ {v};
13         w := w + weight(u, v);
14     n := number of query objects;
15     m := number of duplicated query objects;
16     d := w + 2m;
17     if n == 1 then
18         score_Q := 1;
19     else
20         score_Q := 1/(n*d);
21 sort all possible queries by their scores;
```

step is to find the length of the shortest path between any two query objects in *OT* (line 4-5). Then in the second step we construct a weighted complete graph G_c, in which vertices are all the query objects and the weight of the edge between each pair of vertices is the length found in the first step, and find the minimum spanning tree T of G_c (line 6-13). The weight of T is the minimal cost of the subtree spanning all the query objects in *OT*, which is considered as the distance between the query objects.

We denote the number of query objects as n, and the distance between query objects as d. By the two assumptions A1 and A2, we set the score of each possible query as $\frac{1}{n*d}$, and rank all the possible queries by their scores. Note that if a query has only one object, its score is set to 1 (maximum). The complexity of Algorithm 1 is $O(k*(|S|*|V|^2+|S|^2))$, where k is the number of possible queries, S is the object set in the query that contains the maximum number of objects, and V is the vertex set of the object tree.

Actually ranking search intentions of ambiguous query is a most challenging problem in most search engines. The proposed algorithm is preliminarily based on underlying structure of XML data. However, to get a more effective ranking result, we also need to consider the factor of human's sense by, e.g. user interaction or query history analysis, which is a long-term effort with large amount of practical user data.

After partitioning, we only use the inverted list of the object

in each partition for later processing.

B. Step 2: Inverted list filtering

For a query with keywords partitioned based on objects, we need to filter the Dewey IDs in the inverted list for each partition (or say each object). There are three cases regarding the occurrences of properties and values in each partition: Case (1) there is no property or value, Case (2) there are only properties, and Case (3) there are properties and some properties have values attached. We discuss how to filter the Dewey IDs in the inverted list for a partition in each case as below.

- Case (1): The partition contains only the object, but no property or value. We directly use the inverted list of that object.
- Case (2): The partition contains both the object and properties, but no value. We take the intersection of the inverted lists for the properties as the inverted list for this partition.
- Case (3): Both properties and values appear with the object in the partition. We access the object table R_{obj} for the object *obj* in this partition, and get the Dewey IDs based on the constraints on the properties and values.

Inverted list filtering can reduce the size of relevant inverted lists, which makes later computation more efficient. When there are value keywords in a query, which usually happens in real life, the inverted list size reduction for relevant objects is rather significant, due to the high selectivity based on values.

Example 4.4: The query {book, XML, subject, name} contains two partitions after keyword partitioning: {(book, title/XML), (subject, name)}. The first partition has both property *title* and value *XML* with the object *book*, so by Case (3) we select the Dewey IDs from R_{book} based on *title="XML"*. For the second partition, we just use the inverted list for subject/name. Now we only use two inverted lists to process this query, compared to other approaches which use five inverted lists for the five keywords. Furthermore, the inverted list for *book* contains only a few Dewey IDs due to the high selectivity on the property *title*, compared to other approaches in which the inverted lists for both *book* and *title* contain a lot more Dewey IDs. The filtered inverted lists for *book* and *subject* are shown in Fig. 10.

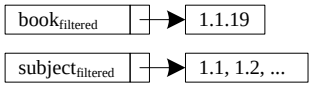


Fig. 10. Filtered inverted lists for *book* and *subject*

C. Step 3: SLCOA processing

It is quite normal that a keyword query involves two or more objects. In this case, after simplifying the query with only objects left, we still need to process an LCA-based computation to find the useful information related to the query objects in the document. We propose a Smallest Lowest Common Object Ancestor (SLCOA) semantics based on our object-oriented document labeling. The rationale of finding SLCOA is that we ensure all the relevant LCA nodes found are object nodes, which are more meaningful as return nodes.

Definition 4.2: (**LCOA of nodes**) Given m nodes u_1, u_2,..., u_m, node *v* is called a Lowest Common Object Ancestor (LCOA) of these m nodes, iff (1) *v* is a common ancestor of all these nodes, (2) *v* is an object node, and (3) *v* does not have any descendant object node *w* which is also a common ancestor of all these nodes. We denote *v* as LCOA(u_1, u_2,..., u_m).

Proposition 4.1: The LCA of a set of nodes has the same Dewey ID as the LCOA of these nodes.

Based on our labeling scheme, each non-object node inherits the Dewey ID of its lowest ancestor object node. Thus the Proposition 4.1 holds (detailed proof is omitted). In the document shown in Fig, 3, the LCA of two *book* nodes 1.1.1 and 1.1.19 is the node *books* 1.1, while the LCOA of the two *book* nodes is *subject* 1.1. Obviously the LCOA node is more meaningful than the LCA node as a result node.

Definition 4.3: (**LCOA of sets**) Given m keywords k_1, k_2,..., k_m, and m sets of nodes I_1, I_2,..., I_m such that $\forall 1 \leq i \leq$m, I_i stores a list of nodes matching k_i. Node *v* belongs to the LCOA of the m sets iff $\exists u_1 \in I_1$, $u_2 \in I_2$,..., $u_m \in I_m$, such that *v*=LCOA(u_1, u_2,..., u_m). We denote $v \in$LCOA(I_1, I_2,..., I_m).

Definition 4.4: (**SLCOA of sets**) A Smallest Lowest Common Object Ancestor (SLCOA) *v* of m sets of nodes I_1, I_2,..., I_m is defined as (1) $v \in$LCOA(I_1, I_2,..., I_m), and (2) *v* does not have any descendant *w* such that $w \in$LCOA(I_1, I_2,..., I_m).

In this step, we find the SLCOA of the reduced inverted lists for the partitions in a query. The SLCOA can be computed in the same way as SLCA computation, because of Proposition 4.1. Thus, all the efficient SLCA computation algorithms can be adopted. The SLCOA of the reduced inverted lists for objects *book* and *subject* in Example 4.4 contains *subject* node 1.1.

D. Step 4: Result return

Now we discuss the last step of query processing, which is returning results to users. There are two issues we concern in this step: how to identify useful return information, and how to extract return values.

1) Output information identification: For a query, we need to identify the meaningful output information. Normally a query aims to find the information of a certain object(s), so we infer the meaningful output information based on object. We first find the LCOA, which is defined in Section IV-C, of the query objects in the object tree of the document. We classify the LCOA into two cases, and infer the output information based on each case:

- Case (1): The LCOA belongs to a new object class from the objects involved in the query. In this case, the details of the LCOA object is considered as the output information.
- Case (2): The LCOA is one of the objects involved in the query. In this case the output information is: for a certain query partition with object *obj*, Case (2a) if there is no

other property or value in this partition, we consider the details of *obj* as output information, Case (2b) if there is a property *prop* without value in this partition, we consider the value of *prop* as output information. Finally, Case (2c) if there is no output information identified in the above two cases, we consider details of the object nodes, which appear as leaves in the subtree of the object tree that spans all the query objects, as the output information.

Example 4.5: Consider the document shown in Fig. 8 and the corresponding object tree in Fig. 9(a). The output information for the query {IT, dancing} is the object *company*, because the two partitions in this query are with respect to the objects *department* and *club*, and the LCOA of *faculty* and *club* in the object tree is *company*, which is in Case (1). Another query {department, maintenance} returns the information about the *department*, as it is in Case (2a). The query {department, name} belongs to Case (2b) and only returns the *name* value of *department*. A last example query {IT, maintenance} returns the details of the maintenance project based on Case (2c).

2) Value extraction: After identifying the output information, the last task is to extract values. In our approach, when the output is an object, we access the object table for that object, and select all the properties and corresponding values based on the Dewey IDs. If the output is a property, we access the corresponding object table and get the property value based on both the object Dewey ID and the property name. Other existing approaches have to access the original XML document to extract the values, which is less efficient. Also without considering object, many existing approaches output the subtree rooted at the identified return node, which may contain lots of irrelevant information.

Example 4.6: For the query {book, XML, subject} to find the subject of book "XML" in the document in Fig. 3, our approach infers the object *subject* as the output node. After finding *subject* node 1.1 is the only answer, we access the object table $R_{subject}$ shown in Fig. 9(b), to get the information, including the property "name" and the value "computer", based on Dewey ID 1.1. Although some other works can also infer *subject* as the return node, when they output the information about *subject*, the whole subtree rooted at the resulting *subject* node will be returned, i.e. the subject name "computer" and all computer books. Obviously this is not effective because all the books under the *subject* node are actually not desired.

E. Advanced search

Inefficient support to range search is a shortcoming for most inverted list based algorithms, in both XML keyword search and IR search. For example, to process a query to find the book with price less than 50, one possible way for existing works is to find all the numeric keywords with values less than 50 and combine the inverted lists of all these keywords for LCA computation. Obviously this is not efficient. In our approach, the values are stored in relational tables. Then the range query on a certain property can be easily performed by a table selection, similar to the selection based on equations.

Example 4.7: Consider a query to find the books with price greater than 50 to the document in Fig. 3. It can be expressed as {book, price>50}. There is only one partition on the object *book* for this query, so we access table R_{book} (Fig. 5) to select Dewey IDs based on price>50. With the selected Dewey ID 1.2.1, we can extract other book information in R_{book}.

Phrase search has been studied in IR area [18]. Since it is nature that a user searches for a phrase in an XML document, how to incorporate such IR search techniques with XML keyword search is also an important issue. In IR, inverted lists index different documents by keywords. In our approach, by putting text values into relational tables, such IR techniques for phrase search can be adopted over the text in each tuple, just like working over the text in each document. Actually nowadays most RDBMS have already supported such full-text search on the value column. If the full-text module is not available or to avoid building additional index for such full-text search, we can also use *LIKE* keyword in SQL to perform phrase search, though it is slower.

Example 4.8: Consider the query to find the books with phrase "hard disk" in the *about* to the document in Fig. 3. The query is expressed as {book, "hard disk"}. As usual we identify *book* as the only object involved, and filter the *book* Dewey IDs in R_{book} (Fig. 5) based on the condition that *about* contains phrase 'hard disk', e.g. *about LIKE %hard disk%*. Using the selected Dewey ID (1.1.1) we can get the full information of the book.

V. Experiments

We conduct experiments to compare our approach with several existing approaches, to evaluate the efficiency and effectiveness. For convenience we name our object-oriented keyword search approach as OOKS. To be fair, we consider the worst case that no explicit information on object is known. Then for each document, OOKS by default considers the parent node of each value as the corresponding property, and the parent node of each property as an object. If more information about objects is known, search efficiency and quality can be even better. As range search is not supported in other approaches, we do not test it.

A. Experimental settings

All the algorithms in our experiments were implemented with Java, and performed on a dual-core 2.33GHz processor with 3.5G RAM. We use three data sets: a real-life DBLP data set (91MB) with 10 million data nodes, a well known benchmark data set XMark (6MB) [19] with 0.5 million data nodes, and a real-life course data set (2MB)[5] with 0.2 million data nodes. We first test eight unambiguous meaningful queries to each data set to compare the efficiency and search quality between OOKS and other approaches. The queries are shown in Fig. 11. These queries include the cases that only data values are involved (e.g. DQ1), only tag labels are involved (e.g. XQ2) and both tag labels and values are involved (e.g. CQ3).

[5]http://www.cs.washington.edu/research/xmldatasets/data/courses/uwm.xml

The selectivity of the queries varies from low (e.g. DQ8) to high (e.g. CQ1). After that we conduct ambiguous query test to show the advantage of OOKS. In OOKS, we used Sybase SQL Anywhere [20] to manage relational tables, and inherit default database parameters.

DBLP	DQ1	Stefan Berchtold	**Course**	CQ1	Urban Internship
	DQ2	phdthesis title		CQ2	course 660-227
	DQ3	IBM research report note		CQ3	course start 3:00pm
	DQ4	ICDT year 1999 inproceedings		CQ4	course_listing level G title
	DQ5	inproceedings title Tong Zhou		CQ5	Study Abroad section_listing
	DQ6	SIGMOD title		CQ6	course B60
	DQ7	Electronic Imaging year 1998		CQ7	BOL B95 days hours
	DQ8	article 1989		CQ8	title credits 4

Xmark	XQ1	Kien Kriegman
	XQ2	closed_auction price
	XQ3	item condemn location
	XQ4	payment Cash shipping
	XQ5	person Sushant Bass profile
	XQ6	open_auction initial current
	XQ7	closed_auction seller buyer item price 03/14/1998
	XQ8	open_auction115 bidder

Fig. 11. Unambiguous experimental queries

B. Index analysis

We analyze the index built in SLCA [2] (LCA-based approach), XReal [21] (IR-style approach) and our OOKS for the largest experimental document, DBLP. The result on four metrics is shown in Fig. 12. SLCA and XReal build inverted list for every different keyword in both tags and values, thus they have a large number of inverted lists to manage; whereas OOKS only builds inverted list for different tag names, whose number is quite limited. The index size of SLCA is the smallest among the three approaches, because SLCA only builds inverted lists. OOKS also needs some table indexes, besides inverted lists. The total index size of OOKS is less than 3 times of the original document. XReal needs to store lots of additional statistical data for ranking, so its index size is very large. We also test the index building time, though it is normally a one-time cost. Both SLCA and XReal require a long time to build inverted lists, especially for tremendous number of value keywords. OOKS performs table insertion for each different value, which is much cheaper than building an inverted list, thus it has lower cost. Note that we adopt a naive way to store each inverted list as a file on disk. Different storing method and implementation may reduce the total size and building time. Last we randomly perform a set of updating operations to test the index updates. For each update, SLCA changes the inverted lists of the relevant value and property, probably also object; while OOKS only changes non-value inverted list and update a tuple in the relevant object table for value. The updating time for both approaches are acceptable. XReal indexes can hardly support updates, as every update will affect many statistical data at many document nodes. Normally updates do not change the document schema, i.e. introducing new element. It is similar to updates in relational database, which only change tuples instead of altering table schemas. Thus in OOKS, updates do not result in new object table creation.

metric \ approach	SLCA	XReal	OOKS
No. of inverted lists	513,758	513,758	29
Total index size (MB)	94	448	254
Building time (min)	512	629	221
Avg. updating cost (ms)	219	N.A.	234

Fig. 12. Index analysis between three approaches on DBLP data

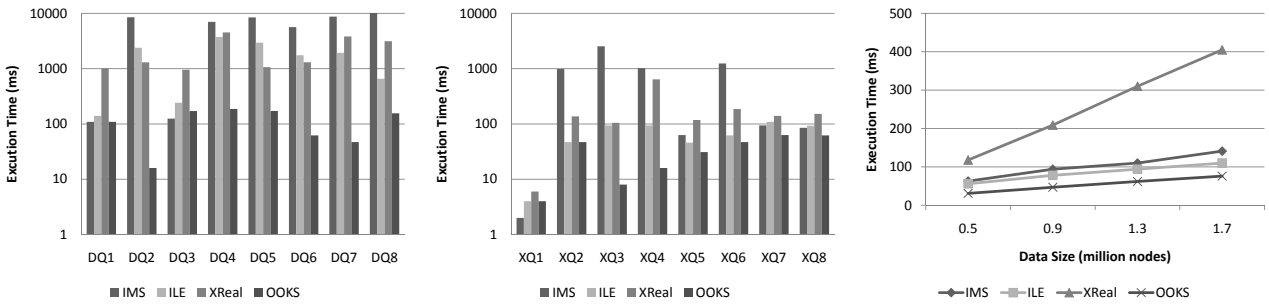
C. Efficiency

In this section we evaluate the query processing efficiency of our OOKS and other approaches. We first compare to LCA-based keyword search algorithm, e.g. SLCA. We take two efficient implementations for SLCA computation: Incremental Multiway-SLCA (IMS) [3] and Indexed Lookup Eager (ILE) [2]. IMS introduces anchor node semantics to skip redundant node search, while ILE introduces index to accelerate inverted list scans. These are two representative approaches to improve SLCA computation. We also compare OOKS with an IR-style XML keyword search algorithm, XReal. We choose a larger data set (DBLP) and a smaller data set (XMark) for the efficiency evaluation[6]. The result is shown in Fig. 13(a) and 13(b). All the indexes, i.e. inverted lists and relational tables are stored on the disk and loaded into memory during query processing. In OOKS, we use ILE to compute SLCOA. The time on return information extraction is not included for all approaches.

From the evaluation results in Fig. 13, we can see that for most queries OOKS outperforms IMS, ILE and XReal. IMS is quite efficient for particular kinds of data sets and queries, but for the data without many repeated nodes under an SLCA candidate, IMS has few advantage over index based ILE and OOKS. ILE uses index to accelerate inverted list scans. However, when the inverted lists are large, it still has to scan many Dewey IDs in each inverted list. For example, to process DQ2, both IMS and ILE have to scan two inverted lists for the two keywords *phdthesis* and *title*. Because of the large size of the *title* list, no matter how they try to skip redundant search, there are still lots of Dewey ID to be read. Using OOKS, we attach *title* to *phdthesis*, and only search for *phdthesis*. Since *phdthesis* has much fewer Dewey IDs than *title* (because *article* and *inproceedings*, also have *title*), by only scanning the inverted list for *phdthesis* and ignoring *title*, naturally OOKS runs much faster. It is similar for many other queries containing frequently appearing, but attachable keywords. XReal introduces overhead on *ID*IDREF* computation and ranking, thus the efficiency is affected in most queries.

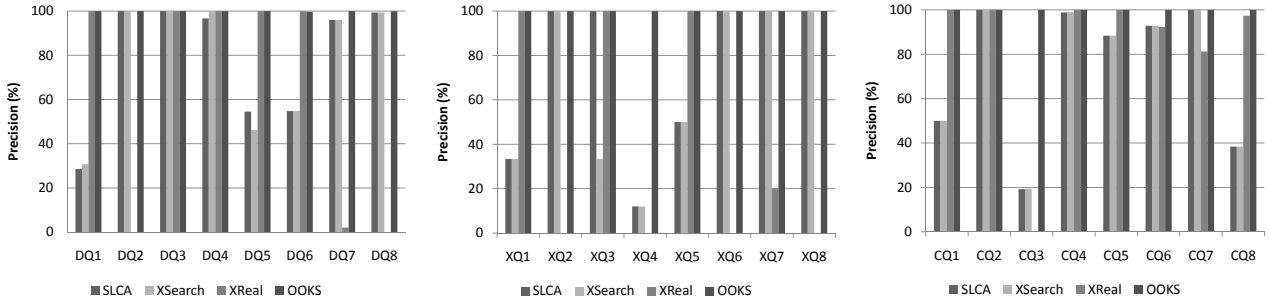
Although OOKS needs to access relational tables, the experiments show that this overhead does not affect the benefit of other reduction. For example, when we process XQ5, it

[6]Since the course data is too small, we do not evaluate the efficiency using the course data.



(a) Execution time on DBLP data (b) Execution time on XMark data (c) Scalability of XMark on XQ5

Fig. 13. Query processing efficiency comparison



(a) DBLP data (b) XMark data (c) Course data

Fig. 14. Search precision comparison

takes 16 ms on keyword partitioning and table accessing, and 15 ms to perform SLCOA computation. The total time 31 ms is still shorter than the execution time of the other approaches.

Note that, in XQ5 we consider *profile* as an object because it is the parent node of some properties like *age*, *gender*, etc. With more semantic information, we should know that *profile* is a composite property of *person*. In this case we can include this composite property into the object table for *person*. Then the SLCOA computation for *person* and *profile* can be saved and the query processing efficiency can be further improved by around 50% (not shown in figure).

We also evaluated the scalability over four XMark data sets with increasing sizes. The evaluation result for the query XQ5 is shown in Fig. 13(c). We can see for all the approaches, the execution time scales well.

D. Search quality

1) Precision for unambiguous queries: In this section, we evaluate the search quality of OOKS in comparison with other approaches, including: SLCA semantics, XSEarch [22], and XReal. Especially to be mentioned is XReal, which is a ranking based keyword search algorithm. We take top k returned results from XReal, where k is the number of expected answers, for search quality comparison. If there is only one expected answer and the first returned result from XReal is not that answer, we use top 5 results from XReal to compute precision, instead of returning 0. Since most approaches do not consider returning useful information, in this test we are only interested in whether the different approaches can find the occurrences of the query keywords in the document correctly. The advantage of OOKS over other approaches to return meaningful results is illustrated in Example 4.6.

All the tested queries are unambiguous. We conducted a mini survey to 19 people to verify the unique search intention of each query and got 100% positive feedbacks. Based on the search intention, for each query we generate an XPath expression, and take the XPath processing result as the standard answer for search quality test.

The two commonly used measures to evaluate the search quality are *precision* and *recall*. Intuitively the *precision* measures the correctness of the search result, while the *recall* measures the completeness. In our experiments, the recall value of each query is very high for all the approaches, or say all the approaches are able to find the correct answers, but they may introduce false positives as noise. Thus we only compare the precision of the different approaches. The result is shown in Fig. 14.[7]

Let us first take a look at the queries involving phrase. DBLP data is flat and regular, and cannot well reflect the difference between different LCA-based semantics. Then we use many DBLP queries to test phrase search, including DQ1, DQ3, DQ5 and DQ7 (rest of DBLP queries do not contain phrase). Phrase also appears in some queries of the other two documents: XQ1, XQ5 and CQ1. For most such queries, SLCA and XSEarch perform badly. Because in those pure LCA-based techniques, indexing on each keyword cannot

[7]The missing column means the relevant query was not tested.

guarantee the existence of phrase. On the other hand, XReal and OOKS can search for phrase more effectively.

For other queries that do not contain phrase search, SLCA and XSEarch may also bring a lot of false positives. Consider the query CQ3 {course, start, 3:00pm} to find the course starting at 3:00pm. Since there are many *course* elements containing both *start* and *end*, and the *end* value is 3:00pm, based on SLCA and XSEarch semantics these courses are also returned. For CQ8 to find the title of the course with credits of 4, SLCA and XSEarch return many course nodes with keyword "4" in the subelement *comments*, but not in the *credits*. Similar queries include DQ6 and XQ4 too. XReal and OOKS perform well for such queries. Although XReal also returns the false positive answers as in SLCA, those false positives have lower scores than correct answers. OOKS attaches value and property to object before processing each keyword query, this attempt significantly reduces the false positive chances and thus gets a good search quality.

Comparing OOKS with XReal, XReal is very good for queries without tag keywords. If the query contains tag keywords, the performance of XReal will be affected. To be fair, in the experiments we did not process queries containing only tag keywords for XReal. XQ7 has only one correct answer. XReal does not return the correct answer with highest score (but with the second highest score), so we consider the top 5 return answers to compute the precision. DQ7 is another case that XReal performed badly. The reason is that both keywords "Electronic" and "Imaging" have a high document frequency. In their IR-style ranking, the *IDF* value is small for this query, so scores of the correct answers are affected. Similarly for CQ6 and CQ7, XReal may return answers that do not contain all the keywords but have higher *TF*IDF* scores on top of the correct answers.

2) Ambiguous query test: In this section we test ambiguous queries. We first show the capability of our approach to return search results based on different search intentions. Most existing algorithms do not differentiate search intentions when returning result, and we only take SLCA as an example for comparison. The experimental results on how SLCA and OOKS deal with three ambiguous queries issued to the XMark data are shown in Fig. 15. Take the first query as an example. Suppose a user would like to find the person with age of 29 and issues the query {person, 29}. However, the number 29 may also appear as a *zipcode* or a part of *street* address under a *person* element in the document. We can see that SLCA returns a large set of *person* nodes with mixture of the age, zipcode or unit number equals 29, which may not be quite meaningful to the user. OOKS can identify different search intentions and return results based on each intention. Obviously the result returned by OOKS is more useful to the user.

All the different interpretations of each ambiguous query in Fig. 15 involve only one object, thus have the same score in our ranking method. We consider them to have the same chance to be asked. Now we take another real-life document,

Query	SLCA result size	OOKS result	
		Identified intentions	size
person 29	82	person/age=29	13
		person/address/zipcode=29	57
		person/address/street.contains(29)	13
United States	2567	address/country='United States'	953
		item/location='United States'	1614
quantity type	2175	open_auction/quantity, open_auction/type	1200
		closed_auction/quantity, closed_auction/type	975

Fig. 15. Return result comparison between SLCA and OOKS on XMark data

Mondial[8], to test our ranking method. The Mondial is a geographic data set integrated from the CIA World Factbook and some other sources. The reason why we choose the Mondial document is that (1) the Mondial data contains more ambiguous terms for our test, and (2) the common sense on geography can help users to give more accurate feedback in our experiment. We only consider the tree structure, but ignore the ID reference in the Mondial data. We choose four ambiguous queries to the Mondial data. For each query, OOKS identifies different interpretations and rank them based on the number of objects involved and the distance between objects. We also conduct a survey to 19 people with basic knowledge on XML data and query, and let them vote each interpretation (assign a score between 0-least likely and 4-most likely) based on their feelings on how likely the interpretation could explain the ambiguous query. The keyword queries and test result are shown in Fig. 16. We can see for most cases, our ranking method can effectively reflect the users' feedback, except the last two interpretations of the second query. The reason is most people participating our survey are more familiar with the city of "Panama", instead of the province of "Panama ", though the latter one is closer to the *country* object in document. This also shows that another important factor for interpretation ranking is human's sense, as discussed in Section IV-D. However, to incorporate human's sense into ranking requires large amount of user data for analysis. We will add more user-interaction to our system to improve our search intentions ranking.

VI. Related work

The first area of research relevant to this work is the computation of the Lowest Common Ancestor (LCA) and/or Smallest LCA (SLCA) of a set of nodes on XML tree data model ([1], [2], [3], [10], [23], [21], [4], [5]). XRANK [1] proposes a stack based algorithm to efficiently compute LCAs, and also presents a ranking method to rank among subtrees rooted at LCAs, where the rank is a combination of keyword proximity (distance among query keywords) inside the subtree and refined PageRank [24] of the subtree.XSEarch [22] is a variation of LCA, which claims two nodes n_1 and n_2 are related if no two distinct nodes with same tag name on the paths from their LCA to n_1 and n_2 (excluding n_1 and

[8]http://www.cs.washington.edu/research/xmldatasets/data/mondial/mondial-3.0.xml

Query	Interpretations	No. of objects	Distance between objects	OOKS rank	Avg. vote (0 – 4)
Indian country	country/ethnicgroups='Indian'	1	-	1	3.5
	ocean/name='Indian Ocean', country	2	2	2	1.2
Panama government	country[name='Panama']/government	1	-	1	4.0
	province/name='Panama', country/government	2	1	2	1.9
	city/name='Panama', country/government	2	2	3	2.3
California Florida Michigan	province/name='California', province/name='Florida , province/name='Michigan'	3	3	1	3.9
	province/name='California', province/name='Florida , lake/name='Lake Michigan'	3	4	2	1.3
Victoria capital	province[name='Victoria']/capital	1	-	1	3.8
	province[city/name='Victoria']capital	2	1	2	2.4
	island/name='Victoria Island', country/capital	2	2	3	1.1
	lake/name='Lake Victoria', country/capital	2	2	3	1.0
	island/name='Victoria Island', province/capital	2	3	5	0.9
	lake/name='Lake Victoria', province/capital	2	3	5	0.8

Fig. 16. Interpretation ranking test on Mondial data

n_2). XKSearch [2] further defines Smallest LCAs to be the LCAs that do not contain other LCAs, and proposes efficient algorithms to compute SLCAs, such as indexed lookup and scan eager. Its main idea is to probe the shortest keyword inverted list S_1 for a given query, and for each node n in S_1, it finds the closest node that contain each of the remaining keyword in the query, and compute the LCA of them as a candidate SLCA. Sun et al. proposes Multiway-SLCA [3] to further optimize the performance of finding SLCAs by maximizing the skipping of redundant LCA computations to the same SLCA result, and it generalizes the algorithm to support keyword search involving combinations of AND and OR boolean operators. Li et al. [10] study Meaningful LCAs which is similar to the concept of SLCA and they incorporate Meaningful LCA search in XQuery. Hristidis et al. [23] introduced *MCTs (minimum connecting trees)* to exclude the subtrees rooted at the LCAs that do not cover query keywords. When a single LCA is too large, MCTs is similar to search by keyword disjunction. [21] proposed an IR-style algorithm with ranking. XSeek [4] and MaxMatch [5] focus on identifying return information.

W3C proposes XQuery Full-Text to enable XQuery support keyword search. However, XQuery Full-Text, as well as some variant, e.g. [25], still requires the user to specify some XQuery-like structural constraints, so it is not pure keyword search. Also to process such a query, the system still replies on XQuery search engine which is different from LCA-based search for pure XML keyword queries. Thus we do not compare with the XQuery Full-Text searching techniques.

There are also many work on keyword search in digraph modeled databases [26], [27], [28], [29], [30], [31], [32], and IR search in XML data without focusing on hierarchical structure [33]. We do not review them in details. Recently, [34] propose to process keyword query in object level. However, it focuses on returning more meaningful result by noticing object and relationship between objects. To the best of our knowledge, our paper is the first work on physically performing XML keyword search in object level.

VII. Conclusion

We propose an object-oriented approach for XML keyword search. In our approach, we assign different Dewey IDs, named OO-Dewey ID, to only object nodes in the document, and let property nodes inherit the OO-Dewey IDs of their associated objects. Values of object properties in our approach are stored in the relational tables for each object class. During keyword query processing, we first partition the keywords based on different object classes, then filter the inverted list of the object in each partition according to the constraints on properties and values using object tables. Now we can perform our proposed smallest lowest common object ancestor (SLCOA) computation on object level with reduced inverted lists. For ambiguous queries, our approach could identify different search intentions, rank them using an object-based algorithm, and return results separately for each search intention. This attempt is more convenient for the user to find useful information based on his real search intention. The relational tables, used to store values and associated property based on different objects, make our system more powerful to support range search and phrase search. Furthermore, these tables can help to extract more meaningful information, i.e. the property values, about each return object node, whereas other existing approaches return the whole subtree of the object node, which contains lots of irrelevant information. Compared to the existing approaches, both query processing efficiency and search quality are can be improved in our object-oriented approach, shown by experimental results.

References

[1] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRANK: Ranked keyword search over XML documents," in *SIGMOD Conference*, 2003, pp. 16–27.

[2] Y. Xu and Y. Papakonstantinou, "Efficient keyword search for smallest LCAs in XML databases," in *SIGMOD Conference*, 2005, pp. 537–538.

[3] C. Sun, C. Y. Chan, and A. K. Goenka, "Multiway SLCA-based keyword search in XML data," in *WWW*, 2007, pp. 1043–1052.

[4] Z. Liu and Y. Chen, "Identifying meaningful return information for XML keyword search," in *SIGMOD Conference*, 2007, pp. 329–340.

[5] ——, "Reasoning and identifying relevant matches for XML keyword search," *PVLDB*, vol. 1, no. 1, pp. 921–932, 2008.

[6] E. Kandogan, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu, "Avatar semantic search: a database approach to information retrieval," in *SIGMOD*, 2006, pp. 790–792.

[7] S. Tata and G. M. Lohman, "SQAK: doing more with keywords," in *SIGMOD*, 2008, pp. 889–901.

[8] E. Vee, U. Srivastava, J. Shanmugasundaram, P. Bhat, and S. Amer-Yahia, "Efficient computation of diverse query results," in *ICDE*, 2008, pp. 228–236.

[9] V. Vesper., "Let's do dewey," http://www.mtsu.edu/∼vvesper/dewey2.htm.

[10] Y. Li, C. Yu, and H. V. Jagadish, "Schema-free XQuery," in *VLDB*, 2004, pp. 72–83.

[11] G. Li, J. Feng, J. Wang, and L. Zhou, "Effective keyword search for valuable LCAs over XML documents," in *CIKM*, 2007, pp. 31–40.

[12] T. W. Ling, M. L. Lee, and G. Dobbie, *Semistructured Database Design (Web Information Systems Engineering and Internet Technologies Series)*. Springer-Verlag, 2004.

[13] Y. Chen, S. B. Davidson, C. S. Hara, and Y. Zheng, "Rrxs: Redundancy reducing XML storage in relations," in *VLDB*, 2003, pp. 189–200.

[14] C. Yu and H. V. Jagadish, "Efficient discovery of XML data redundancies," in *VLDB*, 2006, pp. 103–114.

[15] A. Doan, R. Ramakrishnan, F. Chen, P. DeRose, Y. Lee, R. McCann, M. Sayyadian, and W. Shen, "Community information management," *IEEE Data Eng. Bull.*, vol. 29, no. 1, pp. 64–72, 2006.
[16] A. Spink, "A user-centered approach to evaluating human interaction with web search engines: an exploratory study," *Inf. Process. Manage.*, vol. 38, no. 3, pp. 401–426, 2002.
[17] R. Goldman and J. Widom, "Dataguides: Enabling query formulation and optimization in semistructured databases," in *Proc. of VLDB*, 1997, pp. 436–445.
[18] H. E. Williams, J. Zobel, and D. Bahle, "Fast phrase querying with combined indexes," *ACM Trans. Inf. Syst.*, vol. 22, no. 4, pp. 573–594, 2004.
[19] "http://www.xml-benchmark.org/."
[20] "http://www.sybase.com/products/databasemanagement- /sqlanywhere."
[21] Z. Bao, T. W. Ling, B. Chen, and J. Lu, "Efficient XML keyword search with relevance oriented ranking," in *ICDE*, 2009, pp. 517–528.
[22] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, "XSEarch: A semantic search engine for XML," in *VLDB*, 2003, pp. 45–56.
[23] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava, "Keyword proximity search in XML trees," *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 4, pp. 525–539, 2006.
[24] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine." *Computer Networks*, vol. 30, no. 1-7, pp. 107–117, 1998.
[25] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram, "TeXQuery: a full-text search extension to XQuery," in *WWW*, 2004, pp. 583–594.
[26] S. Agrawal, S. Chaudhuri, and G. Das, "DBXPlorer: A system for keyword-based search over relation databases," in *Proc. of ICDE Conference*, 2002, pp. 5–16.
[27] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword search in relational databases," in *Proc. of VLDB Conference*, 2002, pp. 670–681.
[28] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword searching and browsing in databases using banks," in *Proc. of ICDE Conference*, 2002, pp. 431–440.
[29] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional expansion for keyword search on graph databases," in *VLDB*, 2005, pp. 505–516.
[30] H. He, H. Wang, J. Yang, and P. S. Yu, "Blinks: ranked keyword searches on graphs," in *SIGMOD Conference*, 2007, pp. 305–316.
[31] V. Hristidis, Y. Papakonstantinou, and A. Balmin, "Keyword proximity search on XML graphs," in *ICDE*, 2003, pp. 367–378.
[32] 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, pp. 903–914.
[33] M. Lalmas and A. Tombros, "Evaluating XML retrieval effectiveness at INEX," *SIGIR Forum*, vol. 41, no. 1, pp. 40–57, 2007.
[34] Z. Bao, J. Lu, T. W. Ling, L. Xu, and H. Wu, "An effective object-level XML keyword search," in *DASFAA*, 2010, pp. 93–109.