Object-Oriented XML Keyword Search

Huayu Wu and Zhifeng Bao

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

Abstract. Existing LCA-based XML keyword search approaches are not aware of the significance of using semantics of object to improve search efficiency and quality during LCA-based computation. In this paper, we propose a novel object-oriented approach for XML keyword search. In each step of our approach, i.e., labeling an XML document, constructing related indexes and searching for relevant LCA nodes, we use the semantics of object. We theoretically and experimentally show the superiority our semantic approach over existing approaches.

1 Introduction

XML keyword search is a user-friendly way to query XML database. The most common way to process XML keyword queries is adopting inverted lists to index data nodes in an XML document and perform *lowest common ancestor* (*LCA*) [4] based computation with the inverted lists. There are many subsequent works to either improve the search efficiency of LCA-based approach (e.g., [6]), or improve the search quality (e.g., [5]). However, all the LCA-based search algorithms only focus on checking the structural relationship between query keywords, but do not consider the semantic relationship between them, i.e., the relationship among object, property and value. Without noticing such information, the LCA-based computation may perform redundant searches and return less meaningful result, as discussed in Section 2.2.

In this paper, we propose an Object-Oriented Keyword Search approach, named *OOKS*, to process XML keyword queries in data-centric XML data. The core idea of our approach is to incorporate semantic information, i.e., object, property and value, into LCA-based keyword search. In particular, we construct indexes, i.e., inverted lists and relational tables, in an object-oriented manner, and process XML keyword queries with those indexes in an OO manner as well.

2 Background and Motivation

2.1 Background

The Lowest Common Ancestor (LCA) of a set of nodes S is the common ancestor of the nodes in S, and does not have a descendant node to also be a common ancestor of these nodes. In an XML tree, normally we assign a Dewey ID [1]

M. Jeusfeld, L. Delcambre, and T.W. Ling (Eds.): ER 2011, LNCS 6998, pp. 402–410, 2011.
© Springer-Verlag Berlin Heidelberg 2011

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 each keyword k, the Dewey IDs of the nodes matching k are stored in an inverted list. 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 1. The inverted lists for the keywords in the query {book, XML, author} are shown in Fig. 3(a). The LCAs of the three lists include 1, 1.1.2, and 1.1.2.19 etc. In particular, 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. □

Intuitively the correct answer to the query in Example 1 is book 1.1.2.19, but LCA returns a lot of false answers. To achieve a good search quality, many improved semantics based on LCA are proposed (e.g., [3][10]). In our approach we propose a new semantics SLCOA (discussed later) by incorporating object into SLCA [10]. 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 1, the SLCA of the inverted lists only returns the correct answer 1.1.2.19.

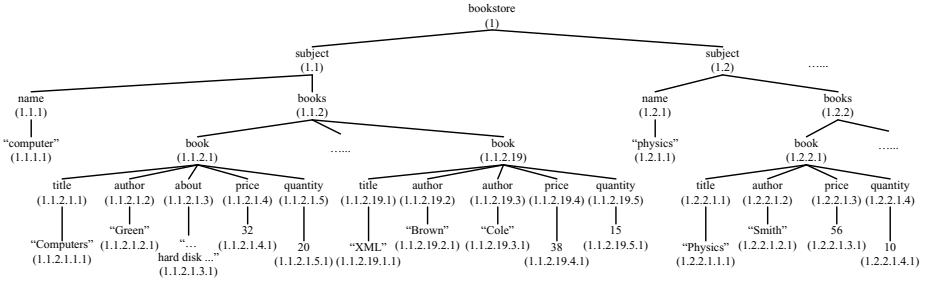


Fig. 1. An example XML document with Dewey IDs for all nodes

2.2 Motivation

We discuss the problems in existing LCA-based XML keyword search approaches which motivate our research.

Efficiency Problems

The LCA-based approach may involve redundant inverted list search. For the example document in Fig. 1, every "book" node corresponds to an object that contains a property "title". To process the query {book, title} which finds all book titles, we actually do not need to scan the inverted list for "title" during LCA-based computation. For another example, consider a query {book, title, XML} to find the books with title of "XML". 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 value of "XML".

Search Quality Problems
Many keyword queries have different interpretations, or say search intentions. A query {physics} may search for a book with title of "Physics", or a subject with name of "physics". Most existing XML keyword search approaches mix different interpretations during query processing, and a user may have difficulty in filtering the mixed results based on his/her search intention. Also using inverted lists, the existing XML keyword search algorithms cannot perform advanced search efficiently, such as range search and phrase search.

3 Object-Oriented Indexes

In this section, we present the OO-based indexes used in our approach. The discussion of object semantics can be found in our technical report [8].

3.1 OO-Dewey ID and Object Tables

Different from existing LCA-based approaches that assign Dewey ID to every document node, OOKS assigns Dewey ID to object nodes (as well as the root) only, thus we also call it OO-Dewey ID[1]. All non-object internal nodes inherit the Dewey ID of the its lowest ancestor object node. Fig. 2 is the OO-Dewey ID assignment for the document in Fig. 1, which contains only two object classes, *subject* and *book*. Using OO-Dewey ID labeling, we can significantly reduce the number of labeled nodes, as shown by comparing Fig. 1 and Fig. 2.

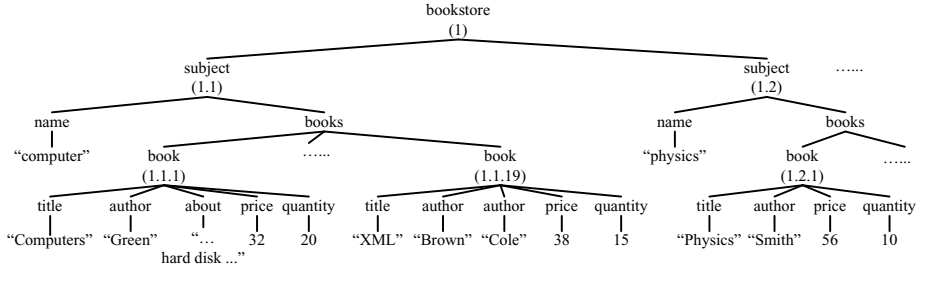
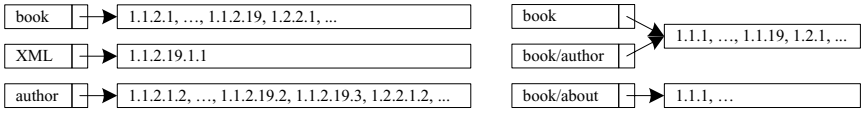


Fig. 2. Document with OO-Dewey ID assignment (only object nodes are labeled)

We only put the Dewey IDs for non-value document nodes into inverted lists, and all the inverted lists are built in object-oriented fashion. The object-oriented inverted lists for "book", "author" and "about" are shown in Fig. 3(b), where the traditional inverted lists are shown in Fig. 3(a). Since we only use object labels, many properties with cardinality of 1 or + to its object, e.g. "author", have the same inverted list as the object.

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



(a) Normal inverted list (b) OO inverted list

Fig. 3. Inverted lists in normal LCA-based approaches and in our OO approach

Values are put into relational tables. The relational tables are also object-oriented. 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 "book" in the document in Fig. 2 are shown in Fig. 4(a). The details of object table construction and problem handling are discussed in [9].

3.2 Other Object-Based Indexes

We assign each type of object and property a numeric ID. We have a hash table to check whether a keyword refers to an object or a property, and return the numeric ID correspondingly. We also maintain Object Attachment Bitmap (OAB) to quickly find by which object classes a given property type is contained, and Containment Index (CI) to match each value to the objects and properties containing it. The details of OAB and CI can be found in [8].

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
…	…

(a) Tables for *book*

$R_{subject}$

OO-Dewey	Name
1.1	computer
…	…

(b) Tables for *subject*

Fig. 4. Object tables for *book* and *subject*

4 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.
2. In each partition, filter the Dewey IDs in the corresponding inverted list using object tables, based on property and value constraints in the partition.

3. Find the smallest lowest common object ancestor (SLCOA) of the inverted lists for different partitions.
4. Identify output information and return result using object tables.

4.1 Step 1: Keyword Partitioning

Definition 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 (*object, [property$_1$[/value$_1$], property$_2$[/value$_2$]...]*) to represent each partition.*

In a keyword query, we create a partition for each object involved. E.g., there are two partitions for the query {subject, XML}, as there are two objects, *subject* and *book*, involved. Then we put the query keywords into the corresponding partitions. 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 our technical report [8]) to partition query keywords.

Generally, there are two phases when we partition the keywords in a query: attaching each value keyword to properties and objects, and attaching each unattached property keyword to objects. The keyword attaching is performed with the *CI* and *OAB*. The details of keyword partitioning can be found in our technical report [8]. It is possible that a query is ambiguous with different attaching ways for a value keyword or a property keyword. We discuss such ambiguous cases in our technical report as well.

Example 2. The query {subject, physics} to the document in Fig. 2 is unambiguous. We attach the value keyword *physics* to the object *subject* with the implicit property *name*. The final partition is {(subject, name/physics)}. □

A special case is that multiple value keywords correspond to the same property type of the same object class. Then we consider both cases of using one partition and using multiple partitions for multiple property/value pairs.

Example 3. In the query {Brown, Cole}, both value keywords belong to book/author. The query is partitioned as {(book, author/Brown, author/Cole)} and {(book, author/Brown), (book, author/Cole)}. The first interpretation finds the book co-authored by the two people, while the second interpretation finds the common information (i.e., subject) of the books written by the two authors. □

For ambiguous queries, we design an algorithm to rank different interpretations in [8].

4.2 Step 2: Inverted List Filtering

For a query with keywords partitioned based on objects, we filter the Dewey IDs in the inverted list for each partition. We consider the three cases regarding the occurrences of properties and values in each partition separately.

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

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

Inverted list filtering can reduce the size of relevant inverted lists, which makes later operations on inverted lists more efficient. When there are value keywords in a query, which frequently happens in practice, the inverted list reduction for relevant objects is more significant, due to the high selectivity of value keywords on specific properties.

Example 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, while other approaches 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*. □

4.3 Step 3: SLCOA Processing

When a keyword query involves two or more objects, after simplifying the query with only objects left, we still need to perform an LCA-based computation among the query objects in the document. We propose a Smallest Lowest Common Object Ancestor (SLCOA) semantics based on our OO 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 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 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 1 holds (detailed proof is omitted). In the document in Fig. 2, 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 is more meaningful than the LCA as a result node.

Definition 3. *(**LCOA of sets of nodes**) 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. *(**SLCOA of sets of nodes**) A Smallest Lowest Common Object Ancestor (SLCOA)* v *of m sets of nodes* I_1,..., I_m *is defined as (1)* $v \in LCOA(I_1, ..., I_m)$, *and (2)* v *does not have any descendant* $w \in LCOA(I_1, ..., I_m)$.

In this step, we find the SLCOA of the reduced inverted lists for the partitions in a query. We can use all the existing efficient SLCA computation algorithms to compute SLCOA, because of Proposition 1.

4.4 Step 4: Result Return

Output Information Identification. Normally a query aims to find the information of a certain object(s), so we infer the meaningful output information based on object. We propose rules in [8] to determine what information should be returned for a keyword query. Generally, we reuse the concept of SLCOA for output information inference. We check the SLCOA of query objects in the structural summary of the XML document. If the SLCOA belongs to a new object class from the objects involved in the query, we will return that SLCOA object. Otherwise, we will return the object in the query partition without property or value, or return the object property containing no value in its partition.

Value Extraction. 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 Dewey ID and the property name.

4.5 Advanced Search

Inefficient support to *range search* is a shortcoming for most inverted list based algorithms. 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 their inverted lists. Obviously it is inefficient. To perform *Phrase search* in XML data, existing works have to adopt a similar technique as in IR [7] to index all phrases, which is very space costly.

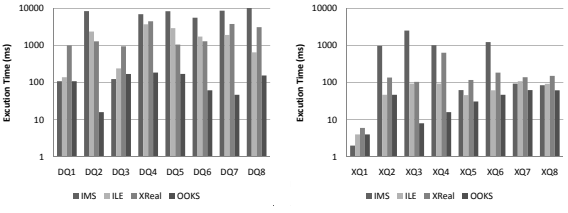
OOKS stores values in relational tables. Then the range queries and the phrase queries can be easily performed by SQL selection.

5 Experiments

5.1 Experimental Settings

All algorithms were performed on a dual-core 2.33GHz processor with 3.5G RAM. We use three data sets: DBLP (91MB), XMark (6MB), and a real-life

course data set (2MB)[3]. We compare OOKS to several existing algorithms, as mentioned later. We use eight unambiguous meaningful queries for each data set to evaluate efficiency and search quality. We also test the ability of OOKS to return results based on different search intentions for ambiguous queries (including our ranking method). In this paper, we only show the experimental result for unambiguous queries in this paper. The query details and the result for ambiguous query test and index analysis can be found in our technical report [8].

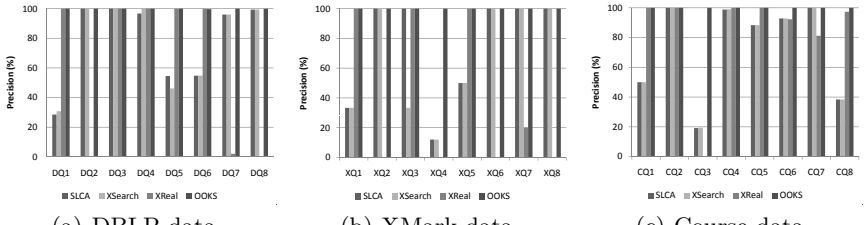


(a) DBLP data (b) XMark data

Fig. 5. Query processing efficiency comparison

5.2 Efficiency

We compare OOKS with two LCA-based algorithms: Incremental Multiway-SLCA (IMS) [6] and Indexed Lookup Eager (ILE) [10] for efficiency test. IMS introduces anchor node semantics to skip redundant node search, while ILE introduces index to accelerate inverted list scans. They are two representative approaches to improve SLCA search. We also compare to an IR-style algorithm XReal [2]. We choose a larger data set (DBLP) and a smaller data set (XMark) for evaluation. The result is shown in Fig. 5(a) and 5(b). In OOKS, we use ILE to compute SLCOA.



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

Fig. 6. Search precision comparison

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

5.3 Search Quality

We evaluate the search quality of OOKS in comparison with other approaches: SLCA, XSEarch [3] and XReal. Especial to be mentioned is XReal, which is a ranking based keyword search algorithm. We take top k results from XReal, where k is the number of expected answers. If there is only 1 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. The recall value is very high for all approaches, or say all approaches can find the correct answers, but may introduce false positives as noises. Then we only compare the precision, as shown in Fig. 6. More explanations of the result are presented in [8].

6 Conclusion

We propose OOKS, a novel object-oriented approach for XML keyword search. In our approach, we label an XML document, construct related indexes and process XML keyword queries in an object-oriented way. Compared to some existing approaches, both query processing efficiency and search quality are improved in OOKS, as shown in our experiments. Furthermore, by introducing relational table for values, our approach can perform advanced search more efficiently.

Acknowledgement. The work of Zhifeng Bao was in part supported by the Singapore Ministry of Education Grant No. R252-000-394-112 under the project name of UTab.

References

1. `http://www.mtsu.edu/~vvesper/dewey2.htm`
2. Bao, Z., Ling, T.W., Chen, B., Lu, J.: Efficient XML keyword search with relevance oriented ranking. In: ICDE, pp. 517–528 (2009)
3. Cohen, S., Mamou, J., Kanza, Y., Sagiv, Y.: XSEarch: A semantic search engine for XML. In: VLDB, pp. 45–56 (2003)
4. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: Ranked keyword search over XML documents. In: SIGMOD Conference, pp. 16–27 (2003)
5. Li, Y., Yu, C., Jagadish, H.V.: Schema-free XQuery. In: VLDB, pp. 72–83 (2004)
6. Sun, C., et al.: Multiway SLCA-based keyword search in XML data. In: WWW, pp. 1043–1052 (2007)
7. Williams, H.E., et al.: Fast phrase querying with combined indexes. ACM Trans. Inf. Syst. 22(4), 573–594 (2004)
8. Wu, H., Ling, T.W., Bao, Z., Xu, L.: Object-oriented XML keyword search. TRA7/10, Technical Report, School of Computing, National University of Singapore (July 2010)
9. Wu, H., Ling, T.-W., Chen, B.: VERT: A semantic approach for content search and content extraction in XML query processing. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 534–549. Springer, Heidelberg (2007)
10. Xu, Y., Papakonstantinou, Y.: Efficient keyword search for smallest LCAs in XML databases. In: SIGMOD Conference, pp. 537–538 (2005)