Efficient XML Keyword Search: From Graph Model to Tree Model

Yong Zeng #, Zhifeng Bao #, Tok Wang Ling #, and Guoliang Li *

#National University of Singapore   * Tsinghua Univeristy

Abstract. Keyword search, as opposed to traditional structured query, has been becoming more and more popular on querying XML data in recent years. XML documents usually contain some ID nodes and IDREF nodes to represent reference relationships among the data. An XML document with ID/IDREF is modeled as a graph by existing works, where the keyword query results are computed by graph traversal. As a comparison, if ID/IDREF is not considered, an XML document can be modeled as a tree. Keyword search on XML tree can be much more efficient using tree-based labeling techniques. A nature question is whether we need to abandon the efficient XML tree search methods and invent new, but less efficient search methods for XML graph. To address this problem, we propose a novel method to transform an XML graph to a tree model such that we can exploit existing XML tree search methods. The experimental results show that our solution can outperform the traditional XML graph search methods by orders of magnitude in efficiency while generating a similar set of results as existing XML graph search methods.

1 Introduction

Keyword search, as opposed to traditional structured query, has been becoming more and more popular on querying XML data in recent years. Users are free from learning the query language and XML schema by simply using some keywords to query the XML data. It attracts a lot of research efforts recently [9, 7, 13, 14, 2]. For example, Figure 1 shows an XML document about a company with *project*, *part* and *supplier*. Each node is assigned a unique Dewey label [12]. To know the price of a part $p1$, users can simply issue a keyword query Q="p1 price" without knowing the schema or any query language (e.g., XPath).

XML documents usually contain some ID nodes and IDREF nodes to represent reference relationships among the data. For example, in Figure 1, every *part* used by each *project* has a reference indicating its *supplier*. An XML document with ID/IDREF is usually modeled as a digraph, where the keyword query results are usually computed by graph traversal [9, 5, 10, 8]. Then the problem is reduced to the problem of finding Minimal Steiner Tree (MST) or its variants in a graph, where an MST is defined as a minimal subtree containing all query keywords in either its leaves or root. Since this problem is NP-complete [6], a lot of works are interested in finding the "best" answers of all possible MSTs, i.e. finding top-K results according to some criteria, like subtree size, etc.

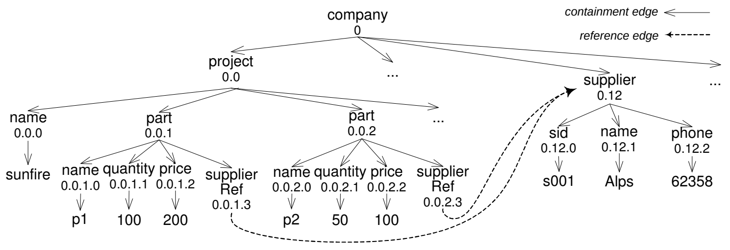


Fig. 1. An Example XML Document (with Dewey Labels)

As a comparison, if we do not consider ID/IDREF, an XML document can be modeled as a tree. Keyword search on an XML tree can be much more efficient based on the tree structure. The results are defined as minimal subtrees containing all query keywords, which is actually a variant of MST adapted to XML tree. Because in a tree, finding an MST for a set of nodes reduces to finding the lowest common ancestor (LCA) of that set of nodes, which can be efficiently addressed by node label computation. For example, if we do not consider the ID/IDREF in Figure 1, given a query Q="p1 price", a node labeled 0.0.1.0 matches keyword "p1" and another node labeled 0.0.1.2 matches keyword "price", then the MST connecting these two nodes is actually the subtree rooted at their lowest common ancestor (LCA), i.e. node 0.0.1. Calculating the LCA simply requires calculating the common label prefix of those two nodes, i.e. 0.0.1 is the prefix of 0.0.1.0 and 0.0.1.2. It is very efficient and does not need any graph traversal.

There are abundant efficient XML tree search methods available, which efficiently calculate the query results based on node labels rather than graph traversal. They build inverted lists of query keywords, in the form of ($keyword$: $dewey_1, dewey_2, dewey_3, ...$), where $dewey_i$ represents the label of a node containing the *keyword*.

XML graph is indeed a tree with a portion of reference edges. This observation offers a great opportunity to accelerate keyword search on XML graph. In this paper, instead of adopting traditional graph search methods, we propose a novel approach to transform an XML graph to a tree model, such that we can exploit XML tree search methods to accelerate query evaluation. The rest of the paper is organized as follows. We present the related work in Sec. 2. Preliminaries are in Sec. 3. We discuss how to transform an XML graph to a tree model for efficient query evaluation in Sec. 4 and how it works on complex reference patterns in Sec. 5. Further extension of our approach is in Sec. 6. The algorithm is presented in Sec. 7. Experiments are in Sec. 8 and we conclude in Sec. 9.

2 Related Work

XML Graph Keyword Search. An XML graph can be modeled as a digraph [9]. Keyword search on a graph is usually reduced to the Steiner Tree problem or its variants: given a graph $G = (V, E)$, where V is a set of nodes and E is a set of

edges, a keyword query result is defined as a minimal subtree T in G such that the leaves or the root of T contain all keywords in the query. The Steiner tree problem is NP-complete [6], and many works are interested in finding the "best" answers of all possible Steiner trees, i.e. finding top-K results according to some criteria, like subtree size, etc. Backward expanding strategy is used by BANKS [4] to search for Steiner trees in a graph. To improve the efficiency, BANKS-II [10] proposed a bidirectional search strategy to reduce the search space, which searches as small portion of graph as possible. Later [5] designed a dynamic programming approach (DPBF) to identify the optimal Steiner trees containing all query keywords. BLINKS [8] proposes a bi-level index and a partition-based method to prune and accelerate searching for top-k results in graphs. XKeyword [9] presented a method to optimized the query evaluation by making use of the graph's schema in XML. [3] proposed an object-level matching semantics on XML graph based on the assumption that the object information is given.
XML Tree Keyword Search. In XML tree, LCA (lowest common ancestor) semantics is first proposed and studied in [7] to find XML nodes, each of which contains all query keywords within its subtree. For a given query $Q = \{k_1,...,k_n\}$ and an XML document D, L_i denotes the inverted list of k_i. Then the LCAs of Q on D are defined as $LCA(Q)$=$\{v \mid v = lca(m_1, ..., m_n), v_i \in L_i(1 \leq i \leq n)\}$. Subsequently, SLCA (smallest LCA [13]) is proposed, which is indeed a subset of $LCA(Q)$, of which no LCA in the subset is the ancestor of any other LCA. ELCA [7, 14], which is also a widely adopted subset of $LCA(Q)$, is defined as: a node v is an ELCA node of Q if the subtree T_v rooted at v contains at least one occurrence of all query keywords, after excluding the occurrences of keywords in each subtree $T_{v'}$ rooted at v's descendant node v' and already contains all query keywords. [2] proposed a statistical way to identify the search target candidates. Recently, more efficient algorithms have been proposed for SLCA and ELCA computation based on hash index [16] and set intersection operation[15].

3 Preliminaries

3.1 Data Model

We model an XML graph as a digraph, where each node of the graph corresponds to an element of the XML data, with a tag name and (optionally) some value. Each containment relationship between a parent node a and a child node b in the XML data corresponds to a containment edge in the digraph, represented as $a \rightarrow b$. Each ID/IDREF reference in the XML data corresponds to a reference edge in the digraph, represented as $a \dashrightarrow b$, where a is the IDREF node and b is the ID node. Thus an XML graph is denoted as $G = (V, E, R)$, where V is a set of nodes, E is a set of containment edges and R is a set of reference edges.

We use T_n to denote the query result rooted at node n. A node n is usually represented by its label or *tag:label*, where tag is the tag name of n. To accelerate the keyword query processing on XML tree model, existing works adopt the dewey labeling scheme [12], as shown in Figure 1.

3.2 Reference Types

If the reference edges are not considered, an XML graph will reduce to an XML tree. There are three types of reference edges in an XML graph: *basic references* (as mentioned in our data model), *sequential references* and *cyclic references*. When an object a references an object b, while b also references a third object c, sequential references occur. Cyclic references happen when containment edges and reference edges form a cycle in an XML graph.

4 Transforming Query Processing over XML Graph to XML Tree

In order to fully exploit the power of tree search methods over the XML digraph, we realize two challenges to tackle: (1) how to transform an XML graph to a proper tree model, which can work with different reference patterns; (2) how to apply existing tree search techniques onto such a tree model. We start addressing these challenges by focusing on the case of *basic references* first, then discuss how the proposed solution can handle sequential and cyclic cases in Sec. 5.

4.1 Real Replication

As shown in Figure 1, every IDREF node in an XML graph points to a particular object with a unique ID value. An object is in the form of a subtree. Therefore, a straightforward yet naive transformation is to just to make a real replication of all such subtrees being referenced. For every reference edge $a \dashrightarrow b$ in the XML graph, we make a replication of the subtree T_b rooted at b and put it under a. Figure 2 shows a transformed XML tree based on the XML graph in Figure 1, where the subtrees in dotted circles are the replication of the subtree $T_{0.12}$.

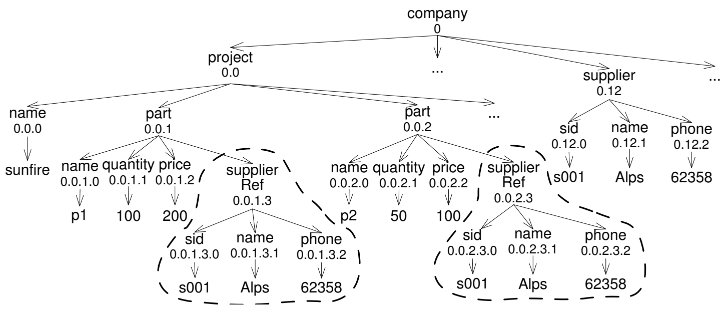


Fig. 2. Naive Method: Real Replication

The transformed XML tree is identical to the original XML graph in the sense that they can infer each other. As a result, any existing keyword search method designed for XML tree can be applied on it without any change.

However, even though the real replication approach can work well for the case of basic references, it is **not usable in practice** because:

- The number of nodes will increase due to the replication of subtrees. We will show in Sec. 5 that, in the worse case, the number of nodes will increase exponentially for the case of sequential references and cyclic references. The space cost is unacceptable in practice.
- Some duplicate results may be generated (as shown in Example 1).

Example 1. If we issue a query Q="Alps phone" to find the phone number of supplier *Alps* in Figure 1, the real replication method will get the transformed XML tree in Figure 2 and do the keyword search on it. By ELCA search method, we get three results: $T_{supplier:0.12}$, $T_{supplierRef:0.0.1.3}$ and $T_{supplierRef:0.0.2.3}$ respectively. The last two results, which are the same as the first one, are actually redundant. Because they are found within the replicated subtrees, while the same results should have already been found in the original subtree. □

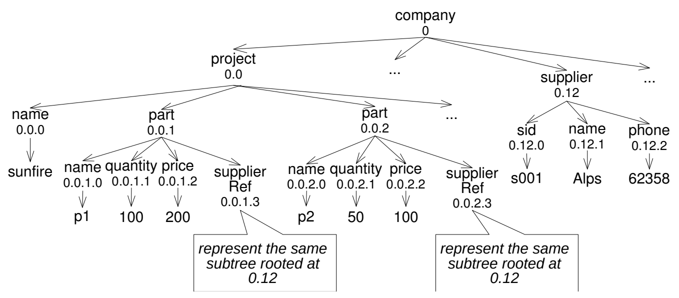
4.2 Virtual Replication

As discussed in the previous section, real replication is not usable in practice. From Example 1 we observe that, a result is **redundant** if it is found within the replicated subtrees, because it must have been found in the original subtree as well. Thus, a result is **non-redundant** only if the root of the result is not within any replicated subtree. Based on this observation, we find that the cost of replicating the subtrees is not necessary because we do not need to search within any replicated subtree.

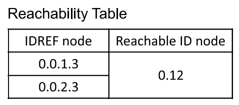
Instead, we propose to use a special node, i.e. the IDREF node, to virtually represent the whole replicated subtree (without inducing any new node), which is able to find the same set of non-redundant results as the real replication method. This is what we call **virtual replication**. For instance, Figure 3(a) shows the idea of using one node to represent the whole replicated subtree. As compared to Figure 2 of real replication, here we use node supplierRef:0.0.1.3 in Figure 3(a) to represent the whole replicated subtree under it.

Example 2. For a query Q="Alps part" in Figure 2, the real replication method will get the following results: $T_{part:0.0.1}$ and $T_{part:0.0.2}$. These two results are non-redundant because their roots, part:0.0.1 and part:0.0.2, are not within any replicated subtree.

Now by virtual replication, keyword "Alps" will no longer match the node 0.0.1.3.1 and 0.0.2.3.1 in Figure 2. Instead, it will match two IDREF nodes 0.0.1.3 and node 0.0.2.3 in Figure 3(a), because we use these two IDREF nodes to represent the whole replicated subtrees. But the final results are still the same: (1) $T_{part:0.0.1}$, which is computed from node 0.0.1.3 (matching keyword "Alps") and node 0.0.1 (matching keyword "part") in Figure 3(a); (2) $T_{part:0.0.2}$, which is computed from node 0.0.2.3 (matching keyword "Alps") and 0.0.2 (matching keyword "part"). □



(a) Part 1: XML tree


Reachability Table

IDREF node	Reachable ID node
0.0.1.3	0.12
0.0.2.3	

(b) Part 2: Reachability Table

Fig. 3. Advanced Method: Virtual Replication (Two Parts)

In this manner, we do not induce any new node while it is able to get the same set of non-redundant results as the real replication method. A proof can be seen at Appendix (section 10).

In order to know which IDREF node represents which subtree, we need a data structure to keep track of the information that which subtree will be replicated under which IDREF node. For such a purpose, we maintain a table called **reachability table**, as shown in Figure 3(b). The table is based on a concept called **reachable**.

Definition 1. ***Reachable.*** *Given an IDREF node n, if there is a directed path from n to a node m in the XML graph, where the last edge of the path is an reference edge, then we say m is a reachable ID node of n.*

Example 3. Given the XML graph in Figure 1, we can find that from the IDREF node 0.0.1.3, there is a directed path from it to node 0.12, where the path ends with a reference edge. So node 0.12 is a reachable ID node for node 0.0.1.3. Similarly, node 0.12 is a reachable ID node for node 0.0.2.3. □

For every pair of *(IDREF node, reachable ID node)*, we store it as a tuple into a table called reachability table, indexed by the attribute "reachable ID node". Every pair of *(IDREF node, reachable ID node)* means the subtree rooted at the *reachable ID node* will be replicated under the *IDREF node*. E.g., the reachability table inferred from the XML graph in Figure 1 is shown in Figure 3(b). The reachability table can be computed offline by a breadth-first search based on each node and the algorithm is presented in Sec. 7.

4.3 Query Evaluation

So far we have completed the transformation from an XML graph to an advanced tree model. Given an XML graph $G = (V, E, R)$, we transform G to a novel tree model consisting of two parts:

1. An XML tree $GT = (V, E, \emptyset)$, which is exactly the same as G with all the reference edges dropped.
2. A reachability table *table*, which maintains the information of which subtree will be virtually replicated under which IDREF node.

Now, we will present how to make an efficient keyword query evaluation based on our transformed tree model.

As discussed in Sec. 1, existing keyword search methods on XML tree do not traverse the tree to search query results. Instead, they compute results based on nodes' labels, e.g., the dewey label. Such labels are stored in an inverted list index in form of $keyword : dewey_1, dewey_2, dewey_3, ...$, where $dewey_i$ represents a node containing the *keyword*. Any LCA-based keyword search method for XML tree will build such an index. Given a keyword query $Q = \{k_1, k_2, ..., k_n\}$, they will retrieve an inverted list for each keyword k_i, and then compute the results based on the inverted lists.

Similarly, after we transform an XML graph to tree model in *virtual replication*, we will also build such an inverted list index. Our tree model consists of an XML tree and a reachability table. The inverted list index will be built on the XML tree, while later the reachability table will help to expand the inverted lists to handle ID/IDREF.

With the index ready, we exploit the existing XML tree keyword search methods and evaluate a keyword query on our tree model in three steps:

1. Retrieve the inverted lists for each keyword in a query.
2. Expand the inverted lists retrieved in step 1.
3. Apply an existing XML tree keyword search method to the expanded inverted lists.

Step 1. Given a query Q = "$k_1 k_2 ... k_n$", one inverted list will be retrieved from the index for each keyword. E.g., given a query Q="Alps part", based on our tree model in Figure 3, we will first retrieve the inverted lists as follows:
"Alps" : 0.12.1
"part" : 0.0.1, 0.0.2, ...

Take note that the keyword "Alps" only matches one node, i.e. 0.12.1, because the inverted list is built on the XML tree in Figure 3(a). So in this step, we only find out the nodes in the XML tree matching the keywords before replication.

Step 2. With the help of the reachability table, we will try to find out whether there is any node in the replicated subtree matching the keywords as well. We can do it in the following way: for each dewey label retrieved in step 1, we check each of its ancestors to see whether the ancestor appears in the *reachable ID node* column of the reachability table. If yes, we add the corresponding IDREF nodes to the inverted list.

E.g., for the dewey label 0.12.1 retrieved in step 1 in the above example, 0.12.1 has two ancestor (prefix): 0.12 and 0. We can find that its ancestor 0.12 appears in the *Reachable ID node* column of the reachability table in Figure 3(b). This means the subtree $T_{0.12}$ is reachable by some IDREF nodes and it should be replicated under those IDREF nodes. So the keyword should match

those IDREF nodes as well. Then we add the corresponding IDREF nodes to the inverted list. But its ancestor 0 does not appear in the *Reachable ID node* column. After that, the expanded inverted list will be:
"Alps" : 0.0.1.3, 0.0.2.3, 0.12.1

After we do the same thing to the "part" inverted list, it will become:
"part" : 0.0.1, 0.0.2, ...

The reachability table is organized in a B+ tree and indexed by the column *Reachable ID node*. So given the dewey label of a reachable ID node, the corresponding IDREF nodes can be retrieved efficiently.

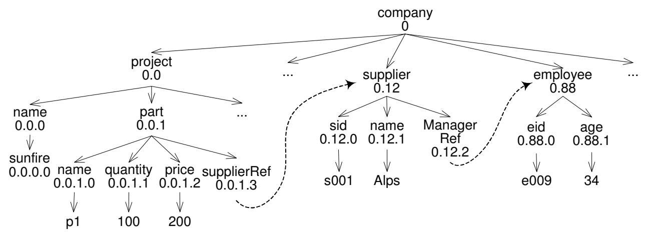
Step 3. After step 2, the final inverted lists are ready. Now we can apply any existing keyword search methods designed for XML tree, like SLCA, ELCA, etc., as all of them operate on the inverted lists for result computation.

5 Sequential References and Cyclic References

Section 4 presents our solution on transformation and query evaluation to basic references case, and in this section we would like to discuss how they are capable of handling the cases of sequential references and cyclic references as well.

5.1 Sequential References

In this case, e.g. in Figure 4(a), to make a complete replication, the subtree rooted at employee:0.88 should be replicated not only under managerRef:0.12.2, but also supplierRef:0.0.1.3. Therefore, if we adopt the real replication approach in Sec. 4.1, the number of nodes may increase exponentially because each object in the sequential references could have multiple references to some other objects.



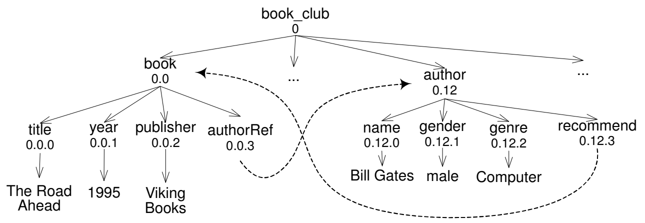
(a) XML Graph with Sequential References

IDREF node	Reachable ID node
0.0.1.3	0.12
0.0.1.3	0.88
0.12.2	

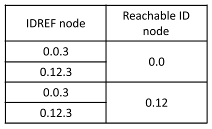
(b) Its Reachability Table

Fig. 4. Constructing Reachability Table for Sequential References

For the virtual replication in Sec. 4.2, we do not need to induce any new nodes into the XML graph. For the XML graph in Figure 4(a), according to Definition 1, we just construct a reachability table as shown in Figure 4(b). E.g., there is a directed path from supplier:0.0.1.3 to employee:0.88, where the path ends with a reference edge. So node 0.88 is a reachable ID node for node 0.0.1.3.



(a) XML Graph with Cyclic References



IDREF node	Reachable ID node
0.0.3	0.0
0.12.3	
0.0.3	0.12
0.12.3	

(b) Its Reachability Table

Fig. 5. Constructing Reachability Table for Cyclic References

5.2 Cyclic References

In the case of cyclic references, our reachability concept in Definition 1 is still able to handle it. E.g., in Figure 5(a), there is a directed path from node authorRef:0.0.3 to node author:0.12, where the path ends with a reference edge. So node 0.12 is a reachable ID node for node 0.0.3. There is also a path from node authorRef:0.0.3 to node book:0.0, where the path ends with a reference edge. So node 0.0 is also a reachable ID node for node 0.0.3. So we will have a reachability table shown in Figure 5(b), and we can find that every ID node in a cycle is reachable by all the IDREF nodes in that cycle.

5.3 Reachability Table Space Complexity

Let the number of IDREF nodes in an XML graph be L, where each IDREF node corresponds to one reference edge, then there could be at most L different ID nodes which are referenced by a reference edge. In the worst case, if every IDREF node can reach all these L ID nodes, then the space complexity is $O(L^2)$ in the worst case. The worst case only happens when all the ID/IDREF nodes forms a big cycle. Furthermore, the L IDREF nodes only occupy a small portion of all nodes in an XML graph in practice (around 5% in real-life data set in our experiments in Sec. 8). This is because every IDREF node must belong to a particular object in the XML, and the attribute information of an object, like ID, name, etc., can only be described by non-IDREF nodes.

6 Further Extension and Optimization

6.1 Removing unnecessary checking of the reachability table

For query evaluation on our transformed XML tree model, we need to expand the inverted lists by checking the reachability table. However, we find that many of the checking is unnecessary. E.g., given the reachability table in Figure 3(b) and the following inverted lists to be expanded: “Alps” : 0.12.1 and “part” : 0.0.1, 0.0.2, ...

As discussed in Sec. 4.3, in step 2 we need to check the ancestor of each dewey label to see whether their ancestors appear in the *Reachable ID node* column of the reachability table. But the ancestors of 0.0.1, 0.0.2, ... do not appear in that column, thus the checking is in vain. To avoid unnecessary checking, we can add a check bit to each dewey label in the inverted list index, indicating whether we need to check such a dewey in the reachability table. E.g., the above inverted lists will become "Alps" : 0.12.1(true) and "part" : 0.0.1(false), 0.0.2(false), ...

Now we only need to check those dewey labels with the check bit being true. Only the ancestors of 0.12.1 will be checked in reachability table. Such a check bit can be computed during offline by checking whether the ancestors of each dewey label appear in the *Reachable ID node* column of the reachability table.

6.2 Adding Distance to Reachability Table

Some of the XML tree keyword search methods need to rank the query results by some criteria. For example, one of the common criteria is the size of the results. It is usually measured by the sum of path length from the result root to each match node. To meet such a need, we can extend our virtual replication method (in Sec. 4.2) by adding a column called *distance* to the reachability table. The *distance* value records the distance from the IDREF node to the reachable ID node. If an IDREF node can reach a reachable ID node by more than one paths, we record the distance of the shortest one. Because substructure with minimal size is in favor in both XML tree search and XML graph search.

Take the reachability table in Figure 4(b) as an example. We can extend the table with a *distance* column and the distance values for the the three tuples are 1, 3 and 1 respectively. For the second tuple, the distance value is 3 because the IDREF node supplierRef:0.0.1.3 need to go through a path 0.0.1.3⇢0.12→0.12.2⇢0.88 to the reachable ID node employee:0.88. The length of such a path is 3.

When we measure the path length from a result root to a match node of a keyword, it consists of three parts: (1) distance from the result root to the IDREF node; (2) distance from the IDREF node to the reachable ID node; (3) distance from the reachable ID node to the match node. The first and the third part can be found in the XML tree, the second part can be found in the reachability table *distance* column. E.g., given a result root part:0.0.1 and a match node eid:0.88.0, the distance from the result root to the match node is the sum of three parts: (1) distance from part:0.0.1 to supplierRef:0.0.1.3 is 1; (2) distance from supplierRef:0.0.1.3 to employee:0.88 is 3, which can be found in the reachability table *distance* column; (3) distance from employee:0.88 to eid:0.88.0 is 1. Therefore the total path length is 5.

7 Algorithms

In this section, we present Algorithm 1 to transform an XML graph to our tree model, which consists of an XML tree and a reachability table.

Given an XML graph, the XML tree part can be easily generated by removing all the reference edges (line 17). The main task here is to generate the reachability table. We will first assign a dewey label to each node in the XML graph (line 3). Then based on each IDREF node n in the XML graph (line 4), we do a breadth-first search to explore the reachable ID nodes until no more new ID node can be further explored (line 5-14). The first node to be explored is the ID node being referenced by n and it will be pushed to a queue (line 5-6). The ID nodes which have been visited will be stored in a set (line 7). Each time we will take a node from the queue to explore until there is no more node in the queue (line 8-9). If the node taken from the queue is not visited before (line 10), we will visit it and mark it as explored (line 11). Then we will further explore within the node. For each IDREF node within it (line 12), we will add the corresponding ID node to the queue (line 13-14), which stores the nodes waiting to be explored. This process will terminate until no more node to explore (line 8). Finally, it will add all reachable ID nodes to the reachability table (line 15-16).

For the algorithm of doing query evaluation based on our tree model, it is similar to the 3 steps discussed in Sec. 4.3 and existing XML tree search algorithms can be easily found in the literature [13, 14]. So the pseudo code will be omitted here.

Algorithm 1: transformXMLGraphToTree(XT)

```
   input  : XML Graph $XG$
   output: Transformed XML Tree $XT$ and reachability table $RT$
1  // Construct reachability table
2  Table $RT$;
3  assignDeweyLabel($XG$); //regardless of reference edges
4  foreach IDREF node $n \in XG$ do
5      $np$ = the ID node which $n$ references to;
6      Queue $nodesToExplore$={$np$};
7      Set $exploredNodes = \{\emptyset\}$;
8      while $nodesToExplore \neq \emptyset$ do
9          $v = nodesToExplore$.removeFirst();
10         if $exploredNodes.notContains(v)$ then
11             $exploredNodes = exploredNodes \cup v$;
12             foreach IDREF node $m \in$ the subtree rooted at $v$ do
13                 $mp$ = the node which $m$ references to;
14                 $nodesToExplore$.add($mp$);
15     foreach node $r \in exploredNodes$ do
16         $RT$.addTuple($n$.dewey, $r$.dewey);
17 $XT$ = removeAllReferenceEdges($XG$); // Generate the XML tree
18 return $XT$ and $RT$;
```

8 Experiments

In this section, we present the experimental results comparing our approach with two graph-search-based methods. One is XKeyword [9], which is dedicated for

XML graph by making use of the XML schema. Another one is BLINKS [8], which is one of the most efficient pure graph search method so far by building a bi-level index.

Experimental Settings. All algorithms are implemented in Java. The experiments were performed on a 2.83GHz Core 2 Quad machine with 3GB RAM running 32-bit windows 7. Berkeley DB Java Edition [1] is used to organize our reachability table in a B+ tree and store the inverted lists. MySQL [11] is used to support XKeyword. BLINKS does not need any database support since it is an in-memory approach.

Data Set. To test the real impact of the keyword search methods, we use a 200MB subset of real-life XML data set with ID/IDREF, ACMDL [1] , in our experiments. It contains publications from 1990 to 2001 indexed by the ACM Digital Library. There are 38K publications and 253K citation (as IDREF) among the publications. Totally 4.5M XML nodes and 4.8M XML edges are included. We can see that IDREF nodes (253K) are 5% of all XML nodes.

8.1 Comparing The Results

There are abundant search methods available designed for XML tree. Here we adopt one of them, i.e. ELCA [14], to work on our transformed tree model. Most of the XML tree methods focus on finding a meaningful subset of all possible results with regard to users' search intention. However, studying whether these subset of results are meaningful regarding users' search intention is not the main focus of this paper. So here we study the similarity between the subset found by tree methods and the subset found by graph methods in terms of result overlap rather than users' search intention.

We generate 100 random queries with two keywords, three keywords, four keywords and five keywords respectively. For each group of queries, we compare the top-20 results found by XKeyword and BLINKS on the original XML graph, to the top-20 results found by ELCA on our transformed tree model. For a fair comparison, all results are ranked by the size of the corresponding Minimal Steiner Tree, i.e. the sum of the path length.

Table 1 shows the average result overlap between our approach and the graph methods, which is calculated as *(# of same results in top-20)/20*. Two results are the same only if the root and each match node are the same.

As we can tell from Table 1, averagely 16 out of top-20 results are the same between our approach and XKeyword, while averagely 18 out of top-20 results are the same between our approach and BLINKS. Because XKeyword sets a maximum result size to constraint the search space, sometimes it finds less than 20 results. Therefore the result overlap is smaller.

8.2 Performance

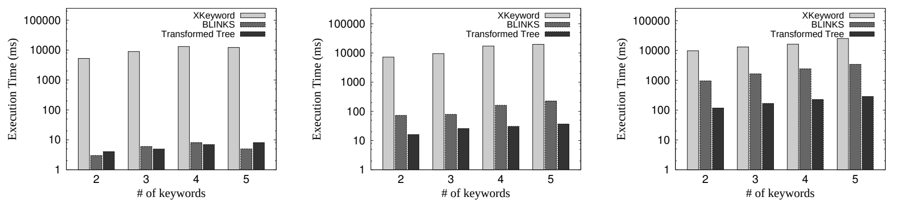
Next we will study the performance of our approach with the transformed tree model. XML tree search methods can be very efficient. E.g., ELCA can compute

[1] Thanks to Craig Rodkin at ACM Headquarters for providing the ACMDL dataset.

Table 1. Result Overlap Between Our Approach and Graph Methods

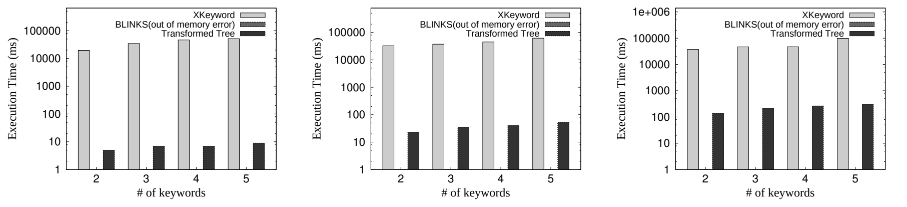
Graph Methods	XKeyword				BLINKS			
# keywords	2	3	4	5	2	3	4	5
Average Result Overlap	77.9%	83.0%	85.4%	82.9%	92.1%	89.0%	90.5%	91.2%

the results by linearly scanning the inverted lists. Here we will compare our approach with two graph search methods, XKeyword and BLINKS. However, BLINKS is an in-memory approach, which throws out-of-memory errors when handling the ACMDL date set. In order to be able to compare the performance of these three approaches, we have to downgrade the data set size to 45MB, which is the maximum data size BLINKS can handle on our machine. Later we will compare on the full data set with only our approach and XKeyword.



(a) keyword frequency 10 (b) keyword frequency 100 (c) keyword frequency 1000

Fig. 6. Query Execution Time (45MB data Size)



(a) keyword frequency 10 (b) keyword frequency 100 (c) keyword frequency 1000

Fig. 7. Query Execution Time (200MB Data Size)

Figure 6 shows the execution time of the three approaches. Our approach performs a full ELCA computation while XKeyword and BLINKS perform a top-20 results computation. We generate 100 random queries for each combination of *keyword frequency* and *# of keywords*. We can see that our approach outperforms XKeyword by orders of magnitude. This is because XKeyword stores the node information in relational tables to accommodate very large graphs, then the results computation is based on table join. Although schema information can help prune some search space, it is still not efficient.

For BLINKS, our approach is faster than it by an order of magnitude when keyword frequency is 100 or 1000. But our approach runs neck and neck with BLINKS when the frequency is around 10. We find that this is because BLINKS is an in-memory approach, which loads the whole graph into memory and does not need to access disk during the whole query evaluation. Yet it is not scalable to large data set. With 1.5 GB heap size assigned to JVM on our machine, 45MB is the maximum data size it can handle without out of memory. For our approach, we store the inverted lists and reachability table in database, so the disk access dominates the query evaluation time when keyword frequency is low.

Now we will compare XKeyword and our approach on the full data set. Figure 7 shows the experiment results. As we can see, our approach is still orders of maganitude faster than XKeyword on full data set. Comparing Figure 7 to Figure 6, we find that XKeyword consumes more time even the keyword frequency in a query is the same. This is because XKeyword is based on table join. Larger data set will lead to larger tables. Therefore XKeyword requires more time to join the tables for results regardless of keyword frequency.

9 Conclusion

In this paper, we observed that an XML graph is mainly a tree structure with a portion of reference edges. It motivated us to proposed a novel method to transform an XML graph with ID/IDREF to a tree model, such that we can exploit abundant efficient XML tree search methods. We transform an XML graph to a tree model by virtually replicating the subtrees being referenced. Our tree model consists of an XML tree and a reachability table, which is capable of handling different kinds of reference patterns in an XML graph. We also designed a query evaluation framework based on our tree model. It can work with the existing XML tree search methods. The experimental results show that our approach is orders of magnitude faster than the traditional search methods on XML graph.

10 Appendix

We declare: *Virtual Replication* will find the same set of non-redundant results as *Real Replication*.

Proof. ($a \prec b$ denotes that node a is an ancestor of b. $a \preceq b$ denotes that $a \prec b$ or $a = b$.) Step 1: to prove that every non-redundant result found by *real replication* can also be found by the *virtual replication*. Let any non-redundant result found by the *real replication* be T_r, which is a subtree rooted at node r. It should be an LCA of a set of nodes $M_{real} = \{n_1, n_2, ..., n_k, \hat{n_1}, , \hat{n_2}, ..., \hat{n_l}\}$ matching the query keywords, where $\hat{n_j}$ is a match node appearing in a replicated subtree and n_i is a match node not in any replicated subtree. Each match node corresponds to a keyword in the query. The LCA relationship can be represented as two properties: ① $r \preceq n_i (1 \leq i \leq k)$, $r \preceq \hat{n_j} (1 \leq j \leq l)$; ② $\nexists r' \prec r$ s.t. $r' \preceq n_i (1 \leq i \leq k)$

and $r' \preceq \hat{n_j}(1 \leq j \leq l)$. In the *virtual replication* method, suppose we use an IDREF node $\hat{N_j}$ to represent the replicated subtree which $\hat{n_j}$ is in, we have ③ $\hat{N_j} \preceq \hat{n_j}$. Then we can prove that the same result T_r can also be calculated based on the following set of match nodes $M_{virtual} = \{n_1, n_2, ..., n_k, \hat{N}_1, , \hat{N}_2, ..., \hat{N}_l\}$. Formally, we need to prove r is the LCA of $M_{virtual}$. Since T_r is a non-redundant result, we have ④ $r \prec \hat{N_j}(1 \leq j \leq l)$. So from ① and ④, we have ⑤ $r \preceq n_i(1 \leq i \leq k)$, $r \preceq \hat{N_j}(1 \leq j \leq l)$. Next we need to prove ⑥ $\nexists r' \prec r$ s.t. $r' \preceq n_i(1 \leq i \leq k)$ and $r' \preceq \hat{N_j}(1 \leq j \leq l)$ by contradiction. If ⑥ is not true, with ③ we can infer that $\exists r' \prec r$ s.t. $r' \preceq n_i(1 \leq i \leq k)$ and $r' \preceq \hat{N_j} \preceq \hat{n_j}(1 \leq j \leq l)$, which contradicts with ②. So with ⑤ and ⑥ being true, r is the LCA of $M_{virtual}$ as well. Step 1 is finished. Step 2: to prove that every non-redundant result found by *virtual replication* can also be found by *real replication.* The proof is similar to step 1, which is omitted here due to space limitation. □

References

1. *Berkeley DB*. http://www.sleepycat.com.
2. Z. Bao, T. W. Ling, B. Chen, and J. Lu. Effective xml keyword search with relevance oriented ranking. In *ICDE*, 2009.
3. Z. Bao, J. Lu, T. W. Ling, L. Xu, and H. Wu. An effective object-level xml keyword search. In *DASFAA*, 2010.
4. G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.
5. B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, 2007.
6. S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. In *Networks*, 1971.
7. L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over xml documents. In *SIGMOD*, 2003.
8. H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, 2007.
9. V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on xml graphs. In *ICDE'03*, 2003.
10. V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.
11. MySQL. http://www.mysql.com.
12. V. Vesper. http://www.mtsu.edu/vvesper/dewey.html.
13. Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD*, 2005.
14. Y. Xu and Y. Papakonstantinou. Efficient lca based keyword search in xml data. In *EDBT*, 2008.
15. J. Zhou, Z. Bao, W. Wang, T. W. Ling, Z. Chen, X. Lin, and J. Guo. Fast slca and elca computation for xml keyword queries based on set intersection. In *ICDE*, 2012.
16. R. Zhou, C. Liu, and J. Li. Fast elca computation for keyword queries on xml data. In *EDBT*, 2010.