International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:1, No:10, 2007

A Two-Step Approach for Tree-structured XPath Query Reduction

Minsoo Lee, Yun-mi Kim, and Yoon-kyung Lee

***Abstract*—**XML data consists of a very flexible tree-structure which makes it difficult to support the storing and retrieving of XML data. The node numbering scheme is one of the most popular approaches to store XML in relational databases. Together with the node numbering storage scheme, structural joins can be used to efficiently process the hierarchical relationships in XML. However, in order to process a tree-structured XPath query containing several hierarchical relationships and conditional sentences on XML data, many structural joins need to be carried out, which results in a high query execution cost. This paper introduces mechanisms to reduce the XPath queries including branch nodes into a much more efficient form with less numbers of structural joins. A two step approach is proposed. The first step merges duplicate nodes in the tree-structured query and the second step divides the query into sub-queries, shortens the paths and then merges the sub-queries back together. The proposed approach can highly contribute to the efficient execution of XML queries. Experimental results show that the proposed scheme can reduce the query execution cost by up to an order of magnitude of the original execution cost.

***Keywords*—**XML, Xpath, tree-structured query, query reduction.

I. Introduction

SINCE XML is widely used as a standard language for information exchange on the Web, the technologies to store and retrieve XML are gaining a considerable amount of interest in both the research and commercial sectors. In general, XML data consists of a very flexible tree-structure which makes it difficult to support such storing and retrieving of XML data. While databases store XML in various ways, the node numbering scheme is one of the most popular approaches to store XML in relational databases. Each XML node can be represented with <docid, begin_pos, end_pos, level> information [1,2]. The *docid* is the document identifier which is used when more than one document exists, and *begin_pos* and *end_pos* are the starting and ending offsets of the XML nodes within a document, and *level* is the depth of the node starting from the root node. An XPath query such as "Paper[Title='XML']/Author/Name" would require the evaluation of the parent-child (or ancestor-descendant) relationships among 'Paper' and 'Title', 'Paper' and 'Author', and 'Author' and 'Name', etc. Structural joins [3] can calculate node pairs that satisfy a hierarchical relationship using the node numbering storage scheme. Assuming two XML node sets R and S, the structural join that calculates the node pairs where a node in R is an ancestor of a node in S is defined as:

$StructuralJoin(R, S) = \{ <r,s> \mid (r \in R) \wedge (s \in S) \wedge (r.docid = s.docid) \wedge (r.begin < s.begin) \wedge (r.end > s.end) \}$

(For parent-child relationships, the condition r.level=s.level-1 is added.)

Although the proposed structural join is very efficient, the XML query processor still needs to carry out multiple structural joins and experiences a high query execution cost. This is more severe when tree-structured queries containing many hierarchical relationships need to be processed.

In this paper, we introduce a method to reduce a tree-structured XPath query to a more concise form with less numbers of structural joins. A two step approach is proposed. The first step merges duplicate nodes in the query, and the second step divides the query into sub-queries and recursively reduces the sub-queries and merge them back together. The proposed scheme can contribute to the query optimizer for obtaining a more efficient query execution plan.

The organization of the paper is as follows. Section 2 discusses the related research. Section 3 explains preliminary concepts including the basic data structure called the XIP tree and the linear query reduction algorithms. Section 4 provides the reduction algorithm for tree-structured queries and section 5 shows the experimental results. Section 6 gives the conclusion.

II. Related Research

Several node numbering schemes for storing large amounts of XML documents in relational databases have been proposed. Zhang et al. [2] suggested using the offsets of the beginning and ending word of each node to represent the 'containment' relationship between nodes in an XML document. The node numbering scheme proposed by Li et al. [1] provides flexibility and efficiency when updating XML documents. Srivastava et al. [3] proposed efficient algorithms for structural joins that can be used to retrieve XML data that is organized using a node numbering scheme. Chien et al. [4] proposed structural join

Manuscript received June 15, 2006. This work was supported in part by the Korean Ministry of Commerce, Industry and Energy, and also in part by the second stage of the BK21 program of the Ministry of Education and Human Resources Development.

Minsoo Lee is with the Dept of Computer Science and Engineering, Ewha Womans University, 11-1 Daehyun-Dong, Seodaemoon-Ku, Seoul, Korea 120-750 (e-mail: mlee@ ewha.ac.kr).

Yun-mi Kim is with the Dept of Computer Science and Engineering, Ewha Womans University, 11-1 Daehyun-Dong, Seodaemoon-Ku, Seoul, Korea 120-750 (e-mail: cherish11@ewhain.net).

Yoon-kyung Lee is with the Dept of Computer Science and Engineering, Ewha Womans University, 11-1 Daehyun-Dong, Seodaemoon-Ku, Seoul, Korea 120-750 (e-mail: polyandry@hanmail.net).

algorithms for indexed XML documents. XML queries containing paths or branches require several consecutive structural joins. Join order selection methods [5] can be used to reduce the cost caused by several joins. Holistic twig joins[6][7] was proposed to process paths or branches at a time instead of stitching several structural joins, but the cost of algorithms are also expected to get higher as the number of nodes involved in XML queries increases.

As for query rewriting techniques, Fernandez et al. [8] proposed query pruning and rewriting techniques for regular path expressions using graph schemas which represent partial knowledge about the structures of the semi-structured documents. Their query rewriting methods are based on state extents over the graph schema, while our query reduction focuses on the reduction of the query itself.

III. Preliminaries: Linear XPath Query Reduction

We first explain the concept and the effects of XML query reduction. Assume that we have stored into the database an XML document that has the structure shown in Fig. 1.

```
<ProgramTable>
<ProgramInformation>
  <programId>P0001</programId>
  <BasicDescription>
    <Title>Sunrise News</Title>
    <Synopsis>Morning News</Synopsis>
    <Keywords>
      <Keyword>politics</Keyword>
      <Keyword>economy</Keyword>
    </Keywords>
    <Genre> <Name>News</Name> </Genre>
    <CastList>
      <CastMember>
        <Role>Reporter</Role>
          <Name>Richard Perry</Name>
        <Role>Producer</Role>
          <Name>Richard Perry</Name>
      </CastMember>
    </CastList>
  </BasicDescription>
</ProgramInformation>
<ProgramInformation>
…
</ProgramTable>
```

Fig. 1 An example XML document

A user can give the following two different forms of the same query for "Retrieve all names of the cast members of the program of which the id is 'P1234'.

(1) //ProgramInformation[@programId='P1234']/ BasicDescription /CastListCast Member/Name

(2) //ProgramInformation[@programId='P1234']// CastMember/Name

However, when the query processor receives these two different inputs from the users, the query execution plans created from these inputs could be totally different. Query (1) needs to perform 5 structural joins, while query (2) only needs to perform 3 structural joins. As shown in this example, the XML query that is specified by the user could be in any arbitrary form while the query processor prefers a reduced number of nodes in the query for efficient execution. In this sense, a query reduction stage is necessary when optimizing user given XML queries.

A. Equivalance Classes among XPaths and XIP trees

In order to perform query reduction on XML queries, we introduce the equivalence class concept between regular path expression XML queries. A group of regular path expressions that are interchangeable among each other (i.e., expressions that yield the same result) is defined to form an *equivalence class*. For example, the two regular expression queries (1), (2) discussed above belong to the same equivalence class. To identify regular path expressions that belong to the same equivalence class, a structure called the *XML instance path (XIP) tree* is used. The XIP tree is dynamically created from the input XML documents and merges structurally similar paths together by holding only a single node in the XIP tree for those child nodes that appear multiple times under the same parent node in the original XML document. Even if two different input XML documents are based on the same DTD or XML Schema, the generated XIP tree can be different. In other words, the XIP tree is constructed for each XML document instance. Fig. 2 shows the XIP tree generated from the XML document in Fig. 1. Note that nodes like *Keyword* appear multiple times in Fig. 1 but appear only once in Fig. 2.

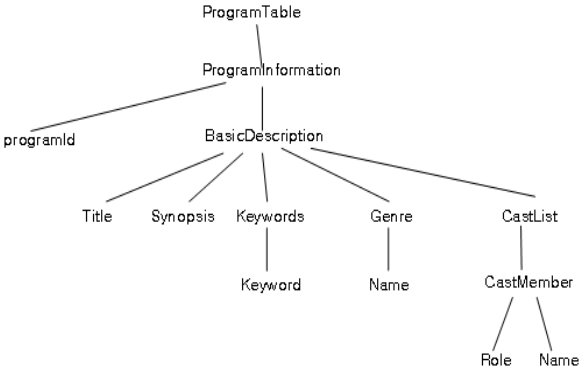

Fig. 2 An example XML Instance Path (XIP) Tree

The size as well as the building cost of the XIP tree in many cases is expected to be small enough to be kept in main memory like the DTD or XML Schema.

B. Linear XPath Reduction Algorithm: A Top-down Approach

The basic idea of path reduction is that a chain of *parent-child* ('/') axes can be replaced with an *ancestor -descendant* axis ('//'), on condition that the resulting regular path expression belongs to the same equivalence class as the original regular path expression. Given a regular path expression, possible expressions within the same equivalence class are too many to

evaluate each expression one by one in order to find the shortest one (i.e., ${}_{2k-1}C_k$ paths for *k*-length paths). To simplify the path reduction process, we make use of a greedy algorithm. The algorithm sequentially probes each node of the given XPath expression from left-to-right (i.e., top-down order in XIP tree) and determines whether it can be removed or not. When a node is removed, its preceding axis is replaced with '//' accordingly. A node can be removed only if the resulting XPath expression where the node is removed still belongs to the original equivalence class. If an arbitrary node is removed from the XPath expression, the resulting one could represent a path that is not a member of the equivalence class of the original regular path expression. The Expand() function is used at this time to check the membership in the equivalence class. Details of the top-down path reduction algorithm are specified in [9].

C. Linear XPath Reduction Algorithm: A Bottom-up Approach

The greedy algorithm used by the top-down approach has a few problems. One of the problems is that the execution time of the algorithm could be significantly large when the path tree is complex, due to the fact that the Expand() function needs to be called every time a node in the original path expression is traversed in the algorithm. Another problem is that an efficient shortest path may not be found because the algorithm considers the nodes in the original path expression in a left-to-right manner. Eliminating nodes closer to the leaves of the XIP tree (i.e., nodes on the right side within a path expression) is more effective than eliminating the ones near the root (i.e., nodes in the left side), because usually if a node is closer to a leaf in the XIP tree, there are more instances of the node occurring in the XML document.

Therefore, an improved algorithm using a bottom-up approach was devised. This algorithm uses not only the XIP tree but also a hash table which maps the node names to the ID's of the nodes that have the name in the path tree. While inspecting the nodes in the original path expression from right-to-left, we also traverse the corresponding nodes in the XIP tree in a bottom-up manner to check if any nodes in the original path expression could be transformed into '//', resulting in a shorter path expression. This requires the identification of anchor nodes, which are nodes that cannot be deleted and should be maintained in the shortened path expression. Details can be found in [9].

IV. Tree-structured XPath Query Reduction

A. Equivalance Classes for Tree-Structured XPath Query

To handle the tree-structured XPath queries, we use the following definitions.

Definition 1. *Complete tree*

A path expression C is a complete path expression if it is both an absolute path expression (i.e., starts with the '/' axis) and does not include any '//' axis within the path expression.

Definition 2. *Matching complete tree*

If a complete tree T results in a path down the XIP tree where the arbitrary tree X (which may contain a '//' axis and contains branch nodes) also represents the same paths in the XIP tree, then T is a 'matching complete tree' of X.

Definition 3. *Expand*

An *Expand* function takes an arbitrary tree as input and returns the set of all matching complete trees of the input path expression. Given an arbitrary tree X, Expand(X) = $\{T_1, T_2, \ldots, T_k\}$ where T_i ($1 <= i <= k$) is a matching complete tree of X. Every matching complete tree T of X is always an element of Expand(X).

Definition 4. *Equivalence class*

A set of path expressions, $X_{set} = \{X_1, X_2, \ldots, X_m\}$ (where $m >= 1$), form an Equivalence class iff there exists a set of complete trees, $T_{set} = \{T_1, T_2, \ldots, T_n\}$ (where $n >= 1$), such that for all X_i ($1 <= i <= m$) in X_{set}, Expand(X_i) = T_{set}. In other words, for all X_i ($1 <= i <= m$) in X_{set}, Expand(X_1) = Expand(X_2) = … = Expand(X_i) = … = Expand(X_m) = T_{set}.

B. Tree-structured XPath Query Reduction Algorithm

If an XPath query contains conditional sentences, the query takes the form of a tree-structure. In this case, the previously discussed linear path reduction algorithm cannot be applied. Therefore, we propose a new algorithm that can handle tree-structured queries in order to reduce such queries into a more efficiently executable form.

The tree-structured XPath query reduction algorithm uses the previously discussed path reduction algorithms to reduce linear paths. The idea behind the tree-structured XPath query reduction is that the branching node (i.e., ProgramInformation in the query (1)) will divide the branch query into several sub-paths and each sub-path could be recursively processed via the branch query reduction algorithm until the sub-path becomes a linear path. Once the linear path is identified, the previously discussed path reduction algorithms, either top-down or bottom-up, can be applied. The merging of these individual results from the paths can be done by comparing the participating node ids. We call this process base reduction.

In some cases, the branch query contains duplicate nodes. In this case, we first attempt to merge duplicate nodes in the branch query as a preprocessing step. When merging duplicate nodes together, we consider three cases based on the level and relationships among the duplicate nodes. Examples of the following cases are shown in Fig. 3. The detail reduction algorithm is described in Fig. 4.

Case 1. *Duplicate nodes are located at the same level and have the same parent.*

Duplicate nodes are merged together and the merged node is linked to the common parent as a child node. The path that is then reduced using base reduction.

Case 2. *Duplicate nodes are located at the same level and have the same parent, but are not leaf nodes.*

Duplicate nodes are merged and the children that the duplicate nodes have are linked to the merged node as

non-overlapping children. Then the tree-structure is reduced using the base reduction process.

Case 3. *Duplicate nodes have different parents and may be at different levels.*

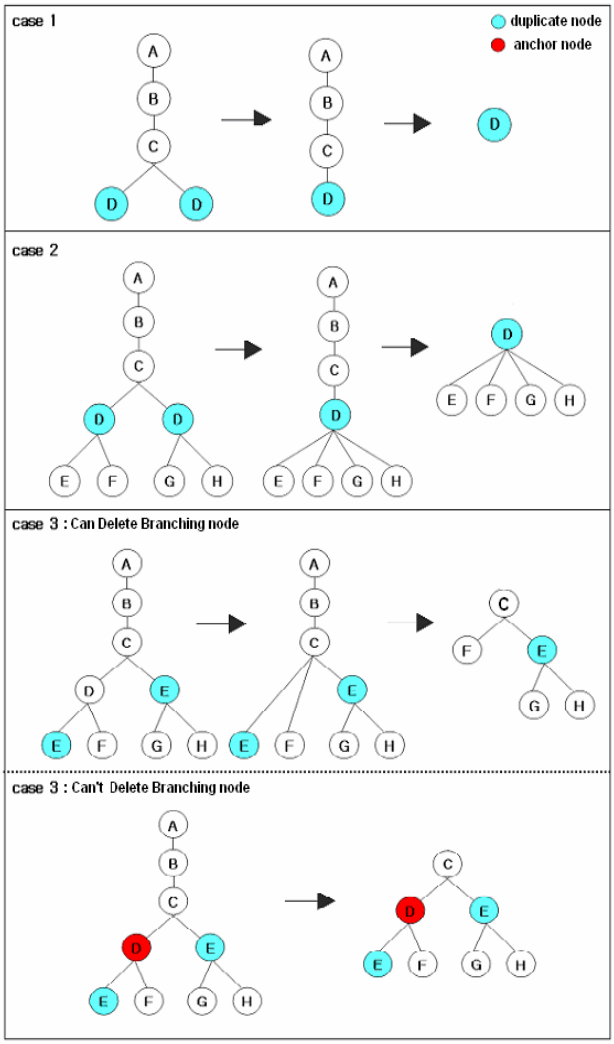

Fig. 3 Query tree transformation cases

If the parent node of the duplicate node is located at a lower level or located at the same level, the branching node can be eliminated. We delete the higher level duplicate node and link their children with the lower level duplicate node. Afterwards the transformed tree is reduced using base reduction. But if we can't eliminate the branching node because the node is an anchor node, we don't transform the tree but just reduce it via the base reduction. At this time, we need to perform the process of extending the reduced tree using the *Expand()* function in order to examine whether the branching node can be deleted or not. However this process has a problem. If the depth of the query tree is deep, checking for the deletion of the branching node can incur a significant amount of overhead. To solve the problem, we decide the threshold for the depth of the query tree for performing the *Expand()* function and select the expand node by a heuristic method. However, in general, most of the query trees do not have the depth exceeding the threshold. Therefore we defer this problem for future work.

Although the details of the algorithm are provided in Fig. 4, a simple example is given to illustrate how the algorithm works.

As an example, consider the path expression /ProgramTable / ProgramInformation / BasicDescription [keywords="AA"] / keywords / keyword. This path has duplicate nodes, 'keywords'. So, we first merge the nodes and transform the query tree's figure and then we reduce the branch path using base reduction. In order to check whether there exist duplicate nodes, we scan the regular expression. Then the results are stored such as same_node[0] = 'keywords', same_node[1] = 'keywords'. As the path expression has duplicate nodes, it falls into case2. And we the following information; merge_data = 'keywords', merge_node[0] = 'Null', merge_node[1] = 'keyword'. We delete same_node[1], and link merge_node[1] to merge_data as the child of merge_data by changing axis of the first node of merge_node[1] to '//'. After we perform this process, the duplicate nodes are merged as one node and the path becomes a linear path. Using the top-down or bottom-up path reduction algorithm introduced in the previous section, the resulting path is //BasicDescription [keywords = "AA"]//keyword.

```
TreeStructuredQueryReduction()
input : P = A1N1...AiNi[Pi]....AnNn /* Ni is a branch node, Pi is a
branch paths.*/
      XIPtree, HashTable

/* declarations for base reduction */
exist_branch_node ← NULL;   preceding_path ← NULL;
branching_node ← NULL;
branch_path[MAX] ← NULL;  branch_reduced_path[MAX] ←
NULL;
all_branch_path ← NULL;      result_path ← NULL;
/* declarations for transforming the tree */
count ← 0; same_node[MAX] ← NULL;
merge_data ← NULL; merge_node ← NULL;
/* check whether the path include the same nodes or not */
for each i from 1 to n-1 do
  for each j from i+1 to n do
    if Ni = Nj then
      same_node[count++] ← Ni ;
    end
  end
end
if same_node != NULL then
  for each i from 0 to count -1 do
    merge_data = getData(same_node[i]);
    for each j from 0 to same_node[i].length-1 do
      merge_node[j] = same_node[i].item(j);
    end
/* case 1 : same level, same parent, all nodes are leaf nodes */
    if merge_nodes' level are same & nodes' parents are same &
nodes are leaf nodes do
      delete(all nodes except merge_node[0]);
    end
/* case 2 : same level, same parent, nodes are not leaf nodes */
        else if merge_nodes' level are same & nodes' parents are same
                & nodes aren't leaf nodes do
      delete(all nodes except merge_node[0]);
      /* move the delete node's children to merge_node[0] */
      moveChildren();
    end
```

```
/* case 3 : different level, different parent */
   else if merge_nodes' level are different & nodes' parents are
different do
     /* check the possibility that nodes' parent nodes can become the
same node.  The reduction tree is in the equivalence class with the
original tree. So, it can delete parent nodes of lower level */
     if CheckEquivalClass() do
       delete(all nodes except the lowest level node);
       /* move the delete node's children to the lowest level node */
       moveChildren();
     end
   end
  end
end
/* Base Reduction */
/* check whether the path is linear path or not */
for each i from 1 to n do
/* CheckBranchNode() is a function to check if the node has two or
more children */
  if CheckBranchNode() = TRUE then
    exist_branch_node ← TRUE;
/* The path has one or more branch nodes */  break;
  end
end
/* The path is a linear path */
if exist_branch_node = FALSE then
  result_path = LinearPathReduction(P, XIPtree, HashTable);
end
/* The path has a branch query */
else
   for each i from 1 to n do
   /* If the node isn't a branching node */
    if CheckBranchNode() = FALSE then
      preceding_path ← preceding_path + AiNi;
    end
    else   /* If that node is a branching node */
     branching_node ← AiNi;   SetAnchorNode();
       /* The function sets the anchor node */
    peceding_path←preceding_path+branching_node;
     for each j from 1 to number_children do
       if Pi's A1 = '//' then
         branch_path[count] ← preceding_path + Pi ;   count ++;
       end
       else
         branch_path[count]← preceding_path + '/' + Pi ;   count ++;
       end
     end
     number_count ← count ;     count ← 0;
      /* each path does reduction recursively */
     for each k from 0 to number_count -1 do
     branch_reduced_path[k]=BranchQueryReduction
                    (branch_path[k],XIPtree,HashTable);
     end
    /* For Merge */
    for each a from 1 to branch_reduced_path[0].length do
      if branch_reduced_path[0]'s AaNa != branching_node then
        result_path←result_path+branch_reduced_path[0]'s AaNa;
      end
      else   /* if the node is branching node */
        result_path ← result_path + branch_node + '[' ;
             /*The number of the children is 2*/
      if number_children = 2 then
           if Aa+1 = '/' then
             result_path←result_path+
               The rest of branch_reduced_path[0] except Aa+1 + ']' ;
             result_path←result_path+branch_reduced_path[1] ;
           end
           else  /* Aa+1 = '//' */
             result_path←result_path+
               The rest of branch_reduced_path[0] + ']' ;
           result_path←result_path+branch_reduced_path[1] ;
         end
      break;
      end
       else
           /*The number of the children is more than 2*/
        if Aa+1 = '/' then
        result_path←result_path+The rest of
              branch_reduced_path[0] except Aa+1 + '|' ;
       end
       for each b from 1 to number_children - 2 do
        if branch_reduced_path[b] 's A1 = '/' then
          result_path←result_path+branch_reduced_path[b]
            except A1 + '|' ;
        end
        else  /* The path starts '//' */
          result_path←result_path+branch_reduced_path[b] + '|' ;
        end
       end
       /* store the last child path */
     if branch_reduced_path[b] 's A1 = '/' then
          result_path←result_path+branch_reduced_path[b]
           except A1 + ']' ;
       end
       else  /* The path starts '//' */
         result_path←result_path+branch_reduced_path[b] + ']' ;
       end
       result_path←result_path+branch_reduced_path[b+1] ;
       break;
      end
   end
    end
  break;
  end
 end
return result_path ;
end
```

Fig. 4 Algorithm of Branch Query Reduction

V. Experimental Results

We evaluated the performance of the XML query reduction algorithm in terms of the overhead of evaluating the XIP trees for query reduction and the benefit in query execution time. About 1G bytes of an XML document generated from the XMark benchmark[10] were populated into the Berkeley DB [11] using the node numbering scheme proposed by Zhang et al. [2]. For an exhaustive performance evaluation of the proposed algorithms, we used as many as 200 XML branch queries that are available from the generated XIP trees. The length of the queries varied from 4 to 10. We implemented the tree-structured query reduction and the structural join algorithm by Srivastava et.al. [3]. Fig. 5, Fig. 6, Fig. 7 summarize the experimental results. For each query set which has the same length varying between 4 and 10, we compared the number of resulting structural joins between the original and the reduced query. Fig. 5 shows that at least 33% of the structural joins could be eliminated by the proposed query reduction algorithm. The response time for the original query and the reduced query were also compared and the results are shown in Fig. 6. The response time of the reduced query could be in some cases 58% less of the response time of the original query. Fig. 7 shows the overhead incurred by the query reduction. As the number of queries increase, the time taken for the structural joins will increase at a very fast pace. However, the time taken for the query reduction does not increase as

much. This illustrates that the time taken for the query reduction incurs a minimal amount of overhead, while in return it can reduce the time taken for structural joins dramatically. Though the effectiveness of the query reduction algorithm could vary depending on the structure of the XIP tree and the queries in the domain area, in most cases, it is expected to improve the query execution time with very little overhead in query reduction time.

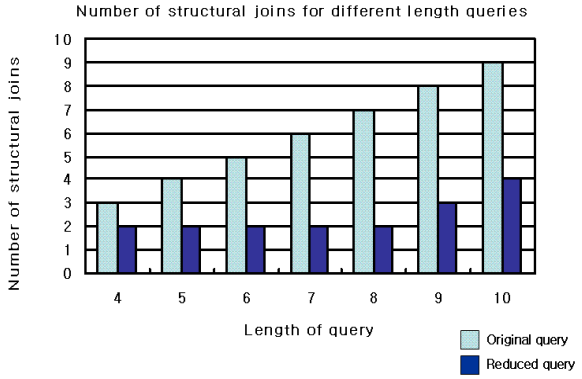

Fig. 5 Query length comparison

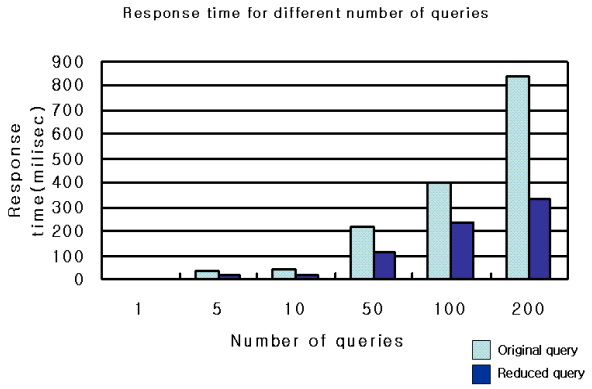

Fig. 6 Response time comparison

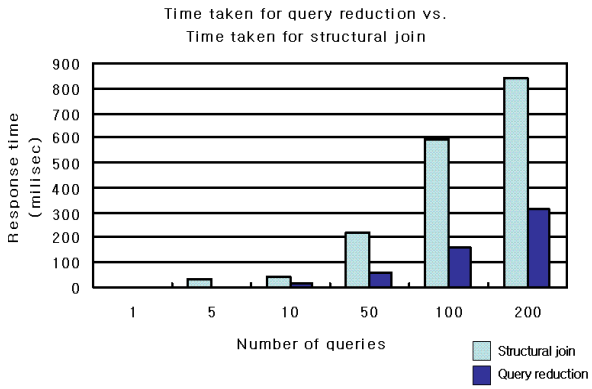

Fig. 7 Reduction time and structural join time comparison

VI. Conclusion

This paper proposed an XML path reduction algorithm for tree-structured queries. This work reduces the number of query nodes in a complex XML query so that an XML query processor exploiting node numbering schemes and structural joins can more efficiently execute the query. The schemes use XIP trees, which reflect the summarized structure of input XML document instances. The equivalence class concept among regular path expressions is very useful for reducing path expressions. Experimental results show that the presented tree-structured XPath query reduction algorithm could eliminate up to 58 % of the original query execution time with only a little extra cost for query reduction. As a result, the performance of the XML query execution was enhanced by up to an order of magnitude.

References

[1] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In Proc. of the 27th VLDB conference, Rome, Italy, Sep. 2001.
[2] Chun Zhang, Jeffrey F. Naughton, Qiong Luo, and David J. DeWitt, and Guy M. Lohman. On supporting containment queries in relational database management systems. SIGMOD conference, Santa Barbara, CA, USA, May 2001.
[3] Divesh Srivastava, Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jinesh M. Patel, and Yuqing Wu. Structural joins: A primitive for efficient XML query pattern matching. IEEE conference on Data Engineering, San Jose, USA, Feb. 2002.
[4] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed XML documents. 28th VLDB conference, p.263-274, Hong Kong, China, Aug. 2002.
[5] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish, Structural Join Order Selection for XML Query Optimization. IEEE conference on Data Engineering, p. 443-454, Bangalore, India, March 2003.
[6] Nicolas Bruno, Nick Koudas, and Divesh Srivastava, Holistic twig joins: optimal XML pattern matching, SIGMOD conference, p. 311-321, Madison, Wisconsin, USA, Jun. 2002.
[7] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu, Holistic Twig Joins on Indexed XML Documents, 29th VLDB conference, p. 273-284, Berlin, Germany, Sep. 2003.
[8] Mary Fernandez and Dan Suciu. Optimizing regular path expressions using graph schemas. IEEE Conference on Data Engineering, p. 4-13, Orlando, Florida, Feb. 1998.
[9] Hyoseop Shin, Minsoo Lee, An Efficient Branch Query Rewriting Algorithm for XML Query Optimization, ODBASE 2005, LNCS 3761, Springer-Verlag, p. 1629-1639, Agia Napa, Cyprus, October 31 - November 4, 2005.
[10] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, Ralph Busse. XMark: A Benchmark for XML Data Management. In Proc. of the 28th VLDB conference, p. 974-985, Hong Kong, China, Aug. 2002.
[11] Sleepycat Software Inc., http://www.sleepycat.com.