A Bottom-up Strategy for Query Decomposition

Le Thi Thu Thuy[1], Doan Dai Duong[1] , Virendrakumar C. Bhavsar[1] and Harold Boley[2]

[1] *Faculty of Computer Science, University of New Brunswick*
Fredericton, New Brunswick, Canada
{Thuy_Thi_Thu.Le, Duong_Dai.Doan, bhavsar}@unb.ca

[2] *Institute for Information Technology - e-Business*
National Research Council of Canada, Fredericton, New Brunswick, Canada
harold.boley@nrc-cnrc.gc.ca

***Abstract*—In order to access data from various different data repositories, in Global-As-View approaches an input query is decomposed into several subqueries. Normally, this decomposition is based on a set of mappings, which describe the correspondence of data elements between a global schema and local ones. However, building mappings is a difficult task, especially when the number of participating local schemas is large. In our approach, an input query is automatically decomposed into subqueries without using mappings. An algorithm is proposed to transform a global path expression (e.g., an XPath query) into local path expressions (e.g., XPath queries) executable in local schemas. This algorithm transforms parts of a path expression from right to left. This transformation is applied from the bottom to the top of a tree and depends on structures of local schemas. Compared to top-down approaches as by Lausen and Marron (LM), our bottom-up approach can be more efficient. Even in the worst case, the time complexity of our algorithm can be n times better than that of LM, where n is the number of parts in a global query. In the best case, for a k-ary tree of height h, the time complexity of our algorithm is $T(n,k,h) = \min(n,h)$, whereas that of LM we have found is**

$$T(n,k,h) = n*(k^{h+1}-1)/(k-1)$$

This can reduce to a large extent the time for forming subqueries for local (e.g., XML) schemas.

***Index Terms*—Query Decomposition, Bottom-up Strategy, Database Integration.**

I. INTRODUCTION

ONE of the most important challenges of Web applications is the utilization of available heterogeneous web data sources to automatically share or interoperate data. This could help users, who want to get relevant data from distributed and chaotic sources, to avoid generating these data from scratch. However, data integration (data interoperation and data interchange) is not an easy task. It requires several steps, such as: (i) creating a global schema and a set of mappings for data sharing between different sources, (ii) resolving data conflicts among different sources, (iii) decomposing queries of users, and (iv) optimizing these queries for efficient answering.

In Global-As-View (GAV) integration systems [5, 11, 12], all participating data sources follow their own schemas, which typically differ from the global schema. When users pose queries based on this global schema, these queries cannot be directly employed to query local sources due to the different structures of the global schema and the local ones. In order to access data from these sources for further processing, the input query must be decomposed into subqueries. Each subquery conforms to the structure of a local source's schema; thus, it can be executed to get the relevant data.

Articles about the most recent XML-based integration systems include: [1, 2, 3, 6, 10, 13]. The common feature of these systems is that a global view (i.e., a global schema) is built to reconcile discrepancies among heterogeneous data sources. Based on this global view, a set of mappings [11, 13] is defined to describe the correspondences of elements between local sources and those of the global view. A mediator [5], the main component of such a system, handles query processing using mappings. Thus, mappings play an important role in the success of the systems. However, building mappings is a difficult task, especially when the number of participating local schemas is large. Normally, these mappings are handcrafted with the help of database experts.

An introduction of our proposed approach is given in Section II. Section III gives a query decomposition example. The assumptions of our approach are stated in Section IV. Section V describes our algorithm for query decomposition, including a flowchart and examples. An extension of our algorithm to process other cases of input queries is given in Section VI. Finally, Section VII focuses on our algorithm analysis and comparisons.

II. PROPOSED APPROACH

In our approach, a user's query (e.g., an XPath query [14]) is decomposed into subqueries without using mappings. Compared to the strategy proposed by Lausen and Marron [7] for query decomposition without using mappings, our approach appears to be more efficient, because instead of a top-down strategy, a bottom-up strategy is used for query decomposition. In the top-down strategy, the leftmost part (i.e., p_1) of a global XPath query '$/p_1/.../p_i/.../p_n$' is first evaluated. This evaluation is performed from the top to the bottom of the XML

tree representing the local schema. This step is recursively applied to all parts of the global query from left to right (i.e., from p_1 to p_n). The top-down query decomposition algorithm is not efficient because in an XPath query the rightmost part (i.e., p_n) plays the most important role. It is the actual result, which the user wants to get from the integrated system. We need to determine whether or not a subquery exists for a specific local schema. If p_n does not exist in a local schema, we can quickly conclude that there is no subquery for this schema. Therefore, in the bottom-up strategy, we first evaluate the rightmost part, and then sequentially proceed from the right to the left part of the input query, and from the bottom to the top of the XML tree representing the local schema. This can significantly reduce the time for searching information in XML trees.

III. Query Decomposition Example

Assume that we have two local schemas (Fig. 1.b and 1.c) of two databases, namely `SESP` and `BIGGER`, represented in terms of the XML format. We also assume a global schema (Fig. 1.a), which is the result of an integration of the two local schemas `SESP` and `BIGGER`. This schema integration step is out of scope of this paper (see [5] for details). Our task is to process input queries so that they can get relevant data from the two above local databases using the global schema. The main task is query decomposition. An example of an input XPath query is Q_{global}= `'/department/mobile/products/jammer[price<200]'`, finding the content of all `jammer` elements having `price` less than `200`, which follows the structure of the global schema. Since each local schema has its own structure, the above query must be decomposed into two queries Q_{SESP} `='/products/jammer[price<200]'` for the `SESP` schema and Q_{BIGGER}`='/department/mobile/jammer[price<200]'` for the `BIGGER` schema.

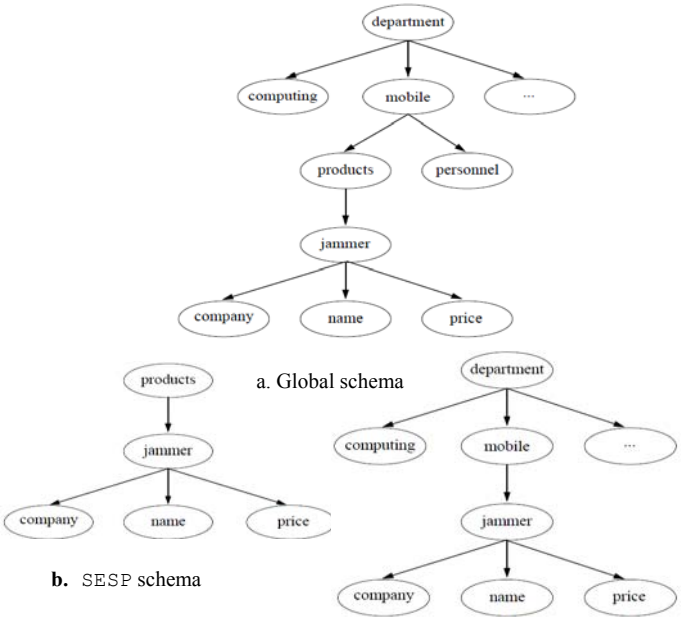


Fig. 1. Example of a global schema and two local schemas (from [7])

IV. Assumptions

In order to apply our algorithm, we make the following assumptions similar to those by Lausen and Marron [7]. There are several data repositories participating in the system. Each of them, represented in XML, has its own local schema. Since these data repositories are created by different designers, there often exist some conflicts between their respective structures. Moreover, these schemas share a pre-defined global schema. For example, in Fig. 1, while in the `SESP` schema, `products` is the father element of `jammer`, in the `BIGGER` schema `mobile` is the father element of `jammer`; also, in the global schema, `mobile` is the father element of `products`. It is clear that there are conflicts between the structures of schemas `SESP` and `BIGGER`. However, when data between them need to be interoperated, they both conform to the global schema as their integration view. We also assume that naming conflicts among local schemas do not exist. This means that our algorithm cannot be applied directly in the presence of naming conflicts such as synonyms or homonyms among local schemas. Finally, we assume that there are two built-in functions for finding the occurrence and the position of a node in an XML tree.

V. Query Decomposition Algorithm

In our algorithm, local schemas are processed sequentially. For each local schema, the global query is transformed into a local query following the structure of this local schema. Thus, by applying this algorithm for all local schemas, local subqueries are obtained for these local sources. The algorithm given in pseudo-code below transforms an XPath query Q_{global}`='/p1/p2/…/pi/…`p_{n-1}`/`p_n`'` following a global schema into a subquery following a local schema. The main idea of the algorithm is as follows. We first take the rightmost part P_n of the user query to evaluate. If P_n is not found on the local schema, we can immediately conclude that there is no subquery for the local schema and stop the algorithm. Otherwise, if P_n is found at a node in the tree (the local schema), we mark that node so that the next search will only be performed on its ancestor nodes (the nodes on the branch from the marked node to the root of the tree). Sequentially, we take P_i `(i=(n-1)..1)` of the query to evaluate. We check whether P_i exists in the local schema or not. Note that, instead of searching the whole tree, we only need to search the ancestor nodes of the previously found node, which we have marked. This can significantly reduce the time for searching a node on a tree. If P_i is found, the local query will be a concatenation of P_i and P_n. In the algorithm (Fig. 2), $\oplus$ denotes a concatenation.

Since the algorithm searches for P_n in the XML tree of a local schema from the bottom up to the root, `Anchor` is used to mark a node in the tree from where the algorithm can start to search. At the initial state of our algorithm, `Anchor = LeftmostLeafNode` means that we begin to search from the leftmost leaf node of the tree. The function `Check(`P_i`, Anchor)` in line 14 of the pseudo-code in Fig. 2 checks whether P_i exists from the `Anchor` up to the root node. The main task of `Check(P, Anchor)` is to find a node `P` in the local schema (i.e., a tree) from a current node `Anchor` up to the root node such that the number of visited nodes in the

worst case is equal the number of nodes in the tree. This ensures that the time complexity of our algorithm (in Fig. 2) for k-ary tree in the worst case is

$$T(n,k,h) = \frac{k^{h+1}-1}{k-1}$$

In order to achieve this goal, we construct a queue Q. If it is the first time search is performed on the tree, we begin our search with leaf nodes. Each time we visit a node, if that node is not P (i.e., we still have not found P in the tree) we check if its father node is already in Q. If the father node is not in Q, we insert the father node in Q. After processing all the leaf nodes, we only need to process the nodes in Q (i.e., all father nodes of leaf nodes). Similarly, if this strategy is iteratively applied for every node in Q, we can visit all nodes from the bottom to the top of the tree. Note that in this case each node is visited only once. (If the previous search has found a node, we simply search on the ancestors of that previously marked node, i.e., Anchor). This function is detailed in Fig. 3.

```
Input
        A local schema S
        A user query Q_global based on a global schema
Output
        A decomposed query Subquery for S

Algorithm

Function BottomUpDecomposition(S, Q_global)
Anchor:= LeftmostLeafNode;
Subquery:='';
i:=|Q_global|;
repeat
  if Check(P_i, Anchor)
  {
    if Subquery=''
        Subquery:=P_i
    else
    {
       if P_i=Anchor
          Subquery:=P_i⊕'/'⊕Subquery
       else
          Subquery:=P_i⊕'//'⊕Subquery;
    }

    if IsRoot(P_i)
        Subquery:='/'⊕Subquery
    else
    {
  if (i>1)
  % p_i is not the leftmost part of Q_global
        Anchor:=father(P_i)
  else
        Subquery:='//'⊕Subquery
    }
  }
  else
  % P_i does not exist
    if (Subquery <> '') and (i=1)
                Subquery:='//'⊕Subquery;
i:=i-1;
until (i=0) or (Subquery='') or IsRoot(P_i+1);
return Subquery;
```

Fig. 2. Pseudo-code of the algorithm for finding a subquery

The flowchart in Fig. 4 illustrates the algorithm given in Fig.2. In order to explain our algorithm, we will walk though it using the two local schema examples of Fig. 1.

```
Input
        P: a node to be found and
        Anchor: current anchor
Output
        Return true if P is found and
               false if P is not found; and
        Anchor


Algorithm

Function Check (P, Anchor)

Q:=nil; // khoi tao queue Q
QQ.next:=Q; // the end node of Q
Dad='';
% we assume root.next=nil;
if Anchor = LeftmostLeafNode
{
 while (Anchor  <> nil and Anchor.value <>P)
      { if Anchor.father.value <>Dad
                {
                    QQ.next:=Anchor.father;
                    QQ:= Anchor.father;
                    Dad:=Anchor.father.value;
                }
                Anchor:=Anchor.next
       }

 if (Anchor <> nil)  return true;
 Anchor:=Q;
 while (Anchor<>nil and Anchor.value <> P)
    {
      if Anchor.father.value <>Dad
        {
         QQ.next:=Anchor.father;
         QQ:= Anchor.father;
         Dad:=Anchor.father.value;
        }
       Anchor:=Anchor.next;
     }
  if Anchor<>nil
     return true % we found P
  else
     return false % we did not find P

}
else % Anchor is not at the leaf node
{
 while ((Anchor<>nil ) and (Anchor.value <>P))
       Anchor:=Anchor.father;
 if (Anchor<>nil )
       return true % we found P
 else
      return false; % we did not find P
}
return;
```

Fig. 3. Pseudo-code of the algorithm for checking the existence of a node from the bottom to the top in a tree

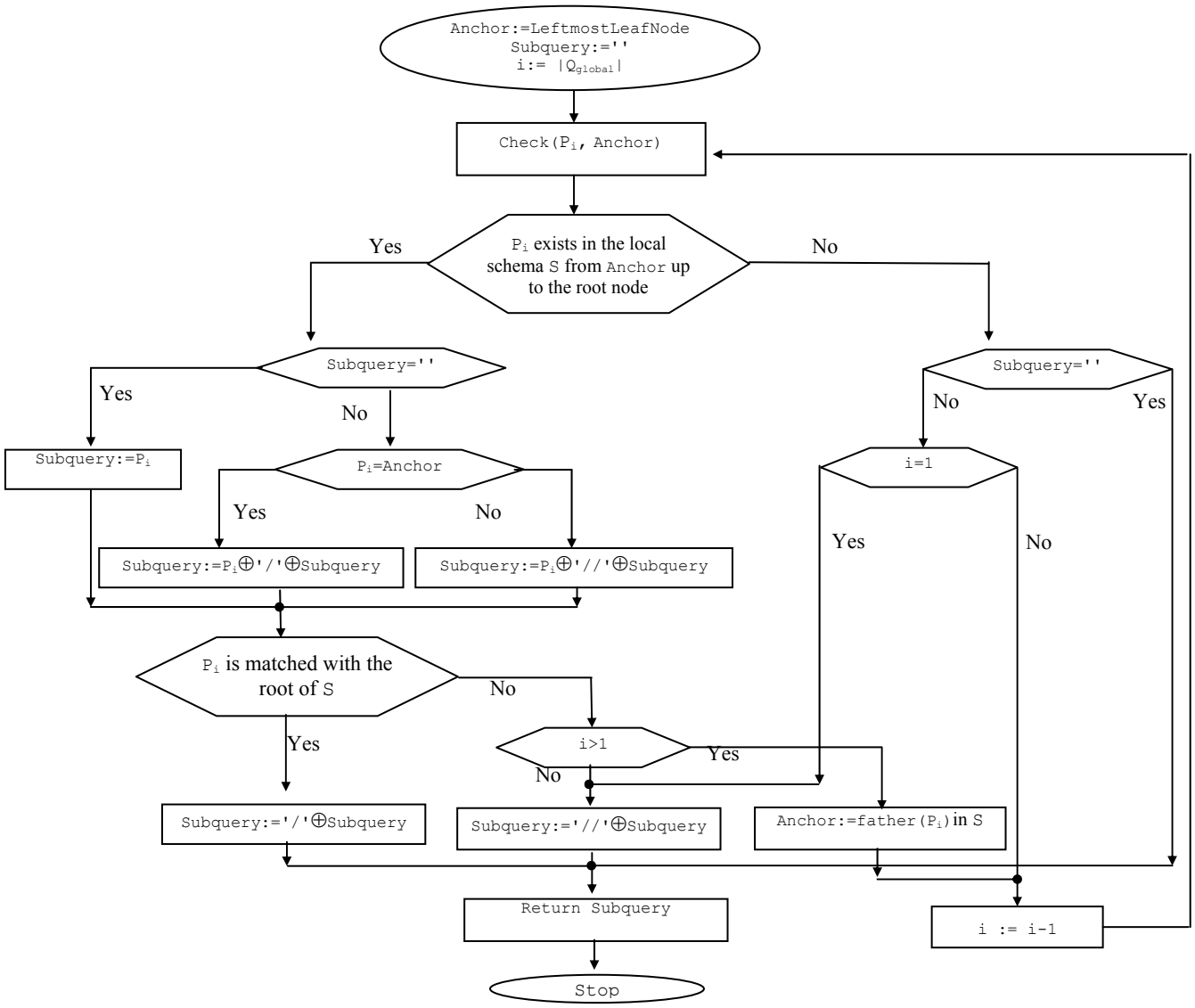


Fig. 4. Flowchart of the algorithm in Fig. 2 for finding a subquery

Example1

In this example, from a global query Q_{global}= '/department/mobile/products/jammer', we produce a subquery for the local schema SESP (Fig. 1.b) using the algorithm of Figure 4. We initialize Subquery:='' and Anchor:= LeftmostLeafNode. We start the algorithm with P_4:='jammer'. Since P_4 is found in schema SESP, P_4 is a part of the transformed query. Because Subquery='', we obtain Subquery:='jammer'. Now, we have i>1, Subquery:='jammer', P_4 not the root of SESP, and Anchor='product'. Therefore we continue the loop. Iteratively, we take P_3 := 'products' from the query Q_{global}. Since P_3 is found in the SESP schema, P_3 is a part of the transformed query. Because P_3=Anchor, we obtain Subquery:=P_3⊕'/'⊕'jammer' (i.e., Subquery:='products/jammer'). Now, P_3 = 'products' is the root node of the SESP schema and we stop the algorithm. We find that the local query for the SESP schema is Subquery:='products/jammer'.

Example2

In this example, we produce a subquery for the schema BIGGER (Fig. 1.c). Given Q_{global}:='/department/mobile /products/jammer', like in example 1, we initiate Subquery='' and Anchor= LeftmostLeafNode. We take the rightmost part P_4:='jammer' from the query Q_{global} . Now, P_4 is found and Subquery=''. So, we assign Subquery='jammer'. Because Subquery <>'' and P_4 is not the root node, we continue our algorithm by searching from the mobile node up to the root node (Anchor:=father('jammer'):='mobile'). In the next step, we have P_3='products'. We find that P_3 does not exist in BIGGER, Subquery <>'' and i>1. Therefore, the next step is now performed with P_2:='mobile'. Because P_2 is found in the schema, Subquery <>'' and P_2 is not the root node, the subquery becomes 'mobile/jammer' and we go to the next step with P_1:='department', and Anchor='department'. P_1 is found in BIGGER. Since department is the root node of the BIGGER schema, stop the algorithm. The subquery found for

the BIGGER schema is Subquery:='/department/mobile/jammer'.

VI. Additional Cases of Input Queries

A. Constraints in Queries

We can apply our algorithm to process XPath queries that contain constraints (filter expressions). For example, if we have a query Q_{global} := '/department/mobile/products/jammer[price<200]', we have to find the corresponding element of price for subqueries following local schemas. Since price is a child element of jammer, we can apply our algorithm by examining price before jammer. This reduces considerable time for forming a subquery because we can avoid transforming the whole query if price does not exist in a local schema. For example, if we apply the Q_{global} query to the schema in Fig. 5, we can quickly recognize that the corresponding subquery for this schema does not exist when we first transform price.

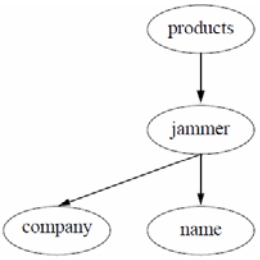
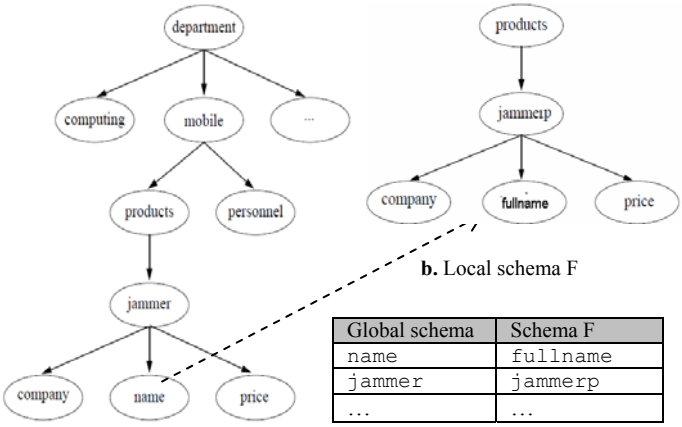


Fig. 5. A local schema without the *price* leaf node (adapted from [7])

However, if constraints of input queries are more general (e.g., [/price<200] or [//price<200]), we can separately apply our algorithm for those constraints before transforming the whole query.

B. Conflicts Between Schemas

Our algorithm shows mismatches between the global schema and local ones just happen with structures. Other naming conflicts [4], such as synonyms and homonyms, are not mentioned. However, we can resolve those conflicts using a dictionary, which contains names of elements in the global schema and their corresponding ones in local schemas. Using the dictionary, we first translate the name of each element of the global query into its corresponding name in the local schema, and then apply the algorithm to it. For example, in Fig. 6, there are naming conflicts between the global schema and the local schema F such as name and fullname, and jammer and jammerp. In this case, we can use a dictionary (Fig. 6.c) to resolve the conflicts. Thus, when finding a subquery for the schema F, instead of using name to search in schema F, we use fullname with the support of the dictionary.



a. Global schema

b. Local schema F

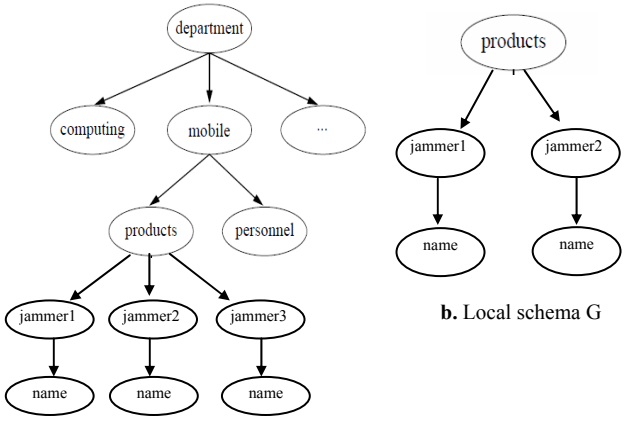
Global schema	Schema F
name	fullname
jammer	jammerp
…	…

c. Dictionary

Fig. 6. Naming conflicts between a global schema and a local schema (adapted from [7])

C. Leaf Nodes with the Same Label

In some special cases, there are leaf nodes with the same label. For example, name is found in several leaf nodes of both the global schema (Fig. 7.a) and the local schema (Fig. 7.b).



a. Global schema

b. Local schema G

Fig. 7. Homonym conflicts in a global schema and a local schema

This situation is actually a homonym conflict, a special type of the naming conflict. As stated in Section IV, our algorithm cannot be applied directly in the presence of naming conflicts. For example, given a query Q_{global} :='jammer1/name', we need to decompose a local query for G. When applying our proposed algorithm, fortunately, if Anchor first points to name, which is the child node of jammer1 in G, we obtain that a local query for G is Subquery :='//jammer1/name'. Otherwise, if Anchor points to name, which is the child node of jammer2 in G, we only obtain Subquery :='//name'. This local query '//name' will provide more solutions than we want (extracted data are name of both jammer1 and jammer2 in G). However, with the top-down strategy [7], the local query produced is Subquery :='//jammer1/name', which is a better solution. It is worth emphasizing that such homonym conflicts do not occur often. Normally, these

conflicts must be solved during the schema integration processes. A possible solution is to rename the homonymous terms (nodes). This is, however, out of scope of this paper (see [4, 5] for details).

VII. Algorithm Analysis

In [7], the authors transform a global XPath query into local subqueries for local schemas using three operators, namely `no transformation`, `subquery generalization` and `subquery elimination`. These operators are used to compute and select suitable elements from the global query to form local subqueries. In their top-down algorithm, the leftmost part (i.e., p_1) of an XPath query '$/p_1/.../p_i/.../p_n$' is first evaluated using these three operators applied to nodes in a local schema. The result from this step is a context C_1 of p_1 in the local schema, such as either `no transformation` (if p_1 is found at the root node), `subquery generalization` (if p_1 is found, but not at the root node) or `subquery elimination` (if p_1 is not found). This step is recursively applied to all parts of the global query from left to right (i.e., from p_1 to p_n). Thus, the result of their algorithm is a sequence of contexts $C_1,...,C_i,...,C_n$ of the query '$/p_1/.../p_i/.../p_n$'. From this sequence of contexts, the corresponding subquery of the input query will be produced. Since each p_i ($i=1..n$) has to be evaluated by all three operators to select the best context, the time to search for information in the local schema is computed as follows.

Suppose the local schema is represented in terms of a binary tree with h as the height of the tree. Let $T(1,2,h)$ and $T(n,2,h)$ represent the time complexity of the evaluation of an arbitrary p_i and a whole query with n parts for a binary tree of height h, respectively. For each p_i ($i=1..n$) of the global query, the three operators, namely `no transformation`, `subquery generalization` and `subquery elimination`, are applied 1, $2^{h+1}-1$ and 1 times, respectively, to evaluate p_i. Therefore,

$$T(1,2,h)=1+(2^{h+1}-1)+1=2^{h+1}+1$$

and

$$T(n,2,h)=n*(2^{h+1}+1).$$

In general, we find that the time complexity of the algorithm in [7] for the whole query given a full k-ary tree is

$$T(n,k,h)=n*\frac{k^{h+1}-1}{k-1}$$

However, as we have discussed in Section II, this top-down query decomposition algorithm is not efficient because in an XPath query the rightmost part (i.e., p_n) plays the most important role. It is the actual result (e.g., `jammer`), which the user wants to get from the integrated system. This can determine whether or not a subquery exists for a specific local schema. If there exists no p_n in a local schema, we can quickly conclude that there is no subquery for this schema. Thus, in our approach, we first evaluate the rightmost part, and then sequentially proceed from the right to the left parts of the input query and from the bottom to the top of the XML tree representing the local schema. The worst case of our algorithm falls into a situation where there exists no subquery for a local schema. In this case, the rightmost part P_n of the global query has to be compared to all nodes of the local schema (i.e., $2^{h+1}-1$ nodes for a binary tree). Therefore, the time complexity of our algorithm is $T(n,2,h)=2^{h+1}-1$ for a binary tree and

$$T(n,k,h)=\frac{k^{h+1}-1}{k-1}$$ for a full k-ary tree.

In the best case, the rightmost part P_n matches with a leaf node of the tree at the first and the same for all P_i nodes at the upper levels of the tree. Therefore, the time complexity of our algorithm in this case is $T(n,2,h)=\min(n,h)$ for a binary tree and also $T(n,k,h)=\min(n,h)$ for a full k-ary tree. Here, the time complexity is $\min(n,h)$ because our algorithm can stop when either all n parts of Q_{global} are processed or all nodes from the bottom to the top of a tree (with the height h) are traversed.

VIII. Conclusion

We have proposed a bottom-up algorithm for query decomposition without predefined mappings. The algorithm can be applied to distributed XML-based data repositories, which may contain conflicts between their respective structures. Having the same motivation as [7] but following a different strategy, we have proposed a more efficient query decomposition algorithm. Our contributions are as follows: (i) a more efficient algorithm for query decomposition is proposed, that is n times better than that of [7] in the worst case and in the best case its time complexity is only $T(n,k,h)=\min(n,h)$, compared to

$$T(n,k,h)=n*\frac{k^{h+1}-1}{k-1}$$

of [7], (ii) a global query is efficiently processed based on its constraints, because our algorithm can stop as soon as a local schema is found not to satisfy these constraints, (iii) our algorithm can work with naming conflicts between local schemas and the global one using a dictionary. Our algorithm can also be extended to work not only with XPath queries, but also with general path expressions like those in Object-Oriented Databases [8, 9].

References

[1] C. Baru, A. Gupta, B. Ludaescher, R. Marciano, Y. Papakonstantinou, and P. Velikhov, "XML-Based Information Mediation with MIX," *In Demo Session. ACM-SIGMOD'99, Philadelphia, PA,* 1999.

[2] C. Baru, B. Ludaescher, Y. Papakonstantinou, P. Velikhov, and V. Vianu, "Features and Requirements for an XML View Definition Language: Lessons from XML Information Mediation," *W3C's QueryLanguage Workshop*, 1998.

[3] Y. Bi and J. Lamb, "Facilitating Integration of Distributed Statistical Databases Using Metadata and XML," Available online: http://webfarm.jrc.cec.eu.int/ETK-NTTS/Papers/final_papers/en187.pdf, 2001. Last accessed on September 8, 2006.

[4] D.D. Duong and L.T.T. Thuy, "Classification and Reconcilement of Conflicts between Heterogeneous XML Schemas," *Proceeding of the 10th Conference on Artificial Intelligence and Applications,* TAAI, 2005.

[5] D.D. Duong and V. Wuwongse, "XML Database Schema Integration Using XDD," *Proceedings of Advances in Web-Age Information*

Management Conference, China: Lecture Notes in Computer Science, Springer Verlag, vol. 2762, 2003, pp. 92-103.

[6] P. Gianolli and J. Mylopoulos, "A semantic approach to XML based data integration" *Proceedings of the 20th. International Conference on Conceptual Modeling (ER)*, Yokohama, Japan, 2001.

[7] G. Lausen and P.J. Marron, "Adaptive evaluation techniques for querying XML-based E-Catalogs," DBLP, 2002, pp. 19-28.

[8] Ludascher, R. Himmeroder, G. Lausen, W. May and C. Schlepphorst, "Managing semistructured data with FLORID: a Deductive object-oriented perspective," *Journal of Information Systems*, vol. 23, no. 8, 1998, pp. 1-25.

[9] W. May, "Logic-based XML data integration: a semi-materializing approach," *Journal of Applied Logic*, vol. 3, No. 1, 2005, pp. 271–307.

[10] The MIX (Mediator of Information using XML). Available online at:

http://db.ucsd.edu/Projects/MIX/, 1999. Last accessed on September 8, 2006.

[11] L.T.T. Thuy and D.D. Duong, "Query Decomposition Using the XML Declarative Description Language" *Proceedings of International Conference of Computational Science and Its Applications*, Singapore: Lecture Notes in Computer Science, Springer Verlag, vol. 3481, 2005, pp. 1066-1075.

[12] L.T.T. Thuy and D.D. Duong, "Integration of XML Databases," *The Journal of Hue University – Vietnam*, vol. 22, 2004, pp 45-52.

[13] L.T.T. Thuy and V. Wuwongse, "Query Processing of Integrated XML Databases" *Proceedings of the 5th International Conference on Information Integration and Webbased Applications & Services*, Jakarta, Indonesia, 2003, pp. 335-344.

[14] XML Path Language (XPath). Available online at:

http://www.w3.org/TR/xpath. Last accessed on September 8, 2006