## A bottom-up algorithm for query decomposition

Le, T.T.T.; Doan, D.D.; Bhavsar, Virendrakumar C.; Boley, Harold



**Publisher's version / Version de l'éditeur:**

*International Journal of Innovative Computing and Applications (IJICA), 3, 1, 2007*

National Research Council Canada
Conseil national de recherches Canada

Institute for Information Technology
Institut de technologie de l'information

NRC · CNRC

# *A Bottom-up Algorithm for Query Decomposition* *

Le, T.T.T., Doan, D.D., Virendrakumar, C., Bhavsar, C., Boley, H.
December 2007

* published in The International Journal of Innovative Computing and Applications (IJICA). December 2007. NRC 50342.



Canada

# A Bottom-up Algorithm for Query Decomposition

**Thi Thu Thuy Le, Dai Duong Doan, and Virendrakumar C. Bhavsar**
Faculty of Computer Science,
University of New Brunswick,
Fredericton, New Brunswick, Canada
E-mail: {Thuy_Thi_Thu.Le, Duong_Dai.Doan, bhavsar} AT unb.ca

**Harold Boley**
Institute for Information Technology -e-Business, NRC,
Fredericton, New Brunswick, Canada
E-mail: harold.boley AT nrc.gc.ca

**Abstract:** In order to access data from various 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 executable in local schemas. This algorithm considers parts of a path expression from right to left, i.e., the algorithm traverses from the bottom to the top of a schema tree depending on the structure of local schemas. Compared to top-down approaches, such as by Lausen and Marron, our algorithm can reduce the time for forming subqueries for local (e.g., XML) schemas to a large extent.

**Keywords:** Query Decomposition, Bottom-up Strategy, Database Integration.

**Reference** to this paper should be made as follows: Le, T.T.T., Doan, D.D., Bhavsar, V.C., and Boley, H. (2007) `A Bottom-up Algorithm for Query Decomposition', *Int. J. of Innovative Computing and Applications*, Vol. XXX, Nos. XXX, pp.XXX.

**Biographical notes: Thi Thu Thuy Le** received her bachelor degree (with honors) in Information Technology from the Hue University of Science, Vietnam, in 1999, and her master degree in Computer Science from the Asian Institute of Technologies, Thailand, in 2002. She is currently a Ph.D. candidate in the Faculty of Computer Science, University of New Brunswick, Fredericton, Canada. Her research interests include database integration, query processing for integrated databases, and spatio-temporal databases.

**Dai Duong Doan** received his bachelor degree (with honors) in Informatics from the University of Hue, Vietnam, in 2000, and his master degree in Computer Science from the Asian Institute of Technologies, Thailand, in 2002. He is currently a Ph.D. candidate in the Faculty of Computer Science, University of New Brunswick, Fredericton, Canada. His research interests include database integration, semantic web, and bioinformatics.

**Dr. Virendrakumar C. Bhavsar** received the B.Eng. (Electronics and Telecommunications) from University of Poona, India, in 1971, and the M.Tech. (Electrical Eng.) and Ph.D. (Electrical Eng.) degrees from the Indian Institute of Technology (I.I.T.), Bombay, in 1973 and 1982, respectively. He holds the Information Systems Professional (ISP) designation from CIPS, Canada. He was on the faculty of the Department of Computer Science and Engineering, I.I.T. Bombay from 1974-83. Since 1983 he has been at the University of New Brunswick, Fredericton, where he is currently a Professor and the Dean of the Faculty of Computer Science. He is also the Director of the Advanced Computational Research Laboratory. He is the President of the SAI Super and Intelligent Computer Systems, Inc. He has authored over 130 research papers in journals and conference proceedings and has edited three volumes. His current research interests include

parallel and distributed processing, artificial intelligence applications in e-Business, e-Learning and bioinformatics, and the semantic web.

**Dr. Harold Boley** is Adjunct Professor at the Faculty of Computer Science, University of New Brunswick, Canada, and leader of the Semantic Web Laboratory at NRC IIT, Fredericton. His current focus is Semantic Web knowledge representation combining ontologies and rules. He received his PhD and Habilitation degrees in Computer Science from the Universities of Hamburg and Kaiserslautern, respectively. He developed the Relational-Functional Markup Language (RFML) before starting and co-leading the Rule Markup Initiative (RuleML). As member of the Joint Committee he co-designed the Semantic Web Rule Language (SWRL), which combines the W3C-recommended Web Ontology Language OWL and RuleML. He also led the design of a First-Order Logic Web language (FOL RuleML) and contributed to the design of the Semantic Web Services Language (SWSL). He is co-editor of the W3C Rule Interchange Format Working Group (RIF Basic Logic Dialect).

## 1 INTRODUCTION

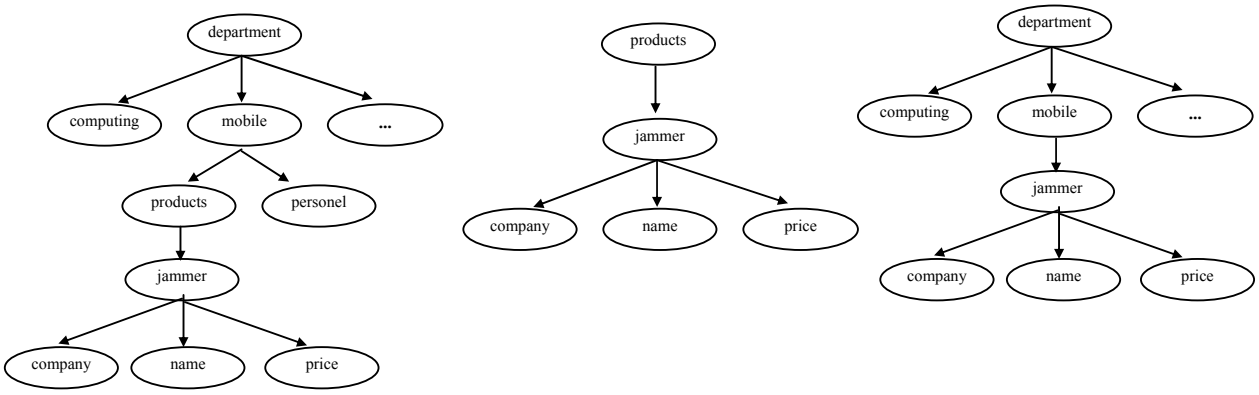
One of the most important challenges of Web applications is the utilization of available heterogeneous Web data sources to automatically share or interoperate data. Systems such as XSIS (Doan and Wuwongse (2003)), InfoSleuth (Bayardo et al. (1997)), FLORID (Ludascher et al. (1998)), and LoPiX (May (2005)) can help users, who want to get relevant data from distributed and heterogeneous sources, to avoid generating these data from scratch. However, data integration (data interoperation and data interchange) is not an easy task. It usually requires several steps, such as: (i) creating a global schema and a set of mappings for data sharing between participating sources, (ii) resolving data conflicts, (iii) decomposing queries of users, and (iv) optimizing these queries for efficient answering. In Global-As-View (GAV) integration systems (Baru et al. (1999), Doan and Wuwongse (2003); Ludascher et al. (1998), Le and Doan (2004), Le and Doan (2005)), 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 the schema of its local source; thus, it can be executed to get the relevant data. Related work about XML-based integration systems is given in (Baru et al. (1999), Baru et al. (1998), Bi and Lamb (2001), Rodriguez-Gianolli and Mylopoulos (2001), Le and Wuwongse (2003)). A common characteristic of these systems is that a global view (i.e., a global schema) is usually built to reconcile discrepancies among heterogeneous data sources. Based on this global view, a set of mappings (Le and Doan (2005), Le and Wuwongse (2003)) is defined to describe the correspondences of elements between local sources and those of the global view. A mediator (Doan and Wuwongse (2003)), 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. Moreover, in a dynamic environment, participating databases evolve with time; thus, maintaining mappings between these databases becomes a great challenge (Doan and Halevy (2005), McCann et al. (2005)). The current paper proposes an algorithm which alleviates the need for mappings.

An overview of our proposed approach is given in Section 2. Section 3 gives a query decomposition example. The assumptions of our approach are stated in Section 4. Section 5 describes our algorithm for query decomposition, including a flowchart and examples. An extension of our algorithm to process additional cases of input queries is given in Section 6. Section 7 focuses on our algorithm analysis and comparisons. Finally conclusions are given in Section 8.

## 2 PROPOSED APPROACH

Lausen and Marron (LM) (Lausen and Marron (2002)) have proposed a query decomposition approach without the use of mappings. They give a *top-down* algorithm for query decomposition. In our approach, a user's query (e.g., an XPath query (XSLT (2007))) is decomposed into sub-queries without mappings using a *bottom-up* algorithm.

Consider a global XPath query *'/$p_1$/.../$p_i$/.../$p_n$'*. In the top-down strategy, the leftmost part $p_1$ is evaluated first. 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, that the user wants to extract from the integrated system. Therefore, we need to determine whether or not this part exists in 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 our bottom-up strategy, we first evaluate the rightmost part, and then sequentially proceed from the right to the left parts of the input query, hence 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.




**a.** Global schema **b.** `SESP` schema **c.** `BIGGER` schema

**Figure 1** *Example of a global schema and two local schemas (from Lausen and Marron (2002))*

## 3 QUERY DECOMPOSITION EXAMPLE

To explain our algorithm, an example from Lausen and Marron (2002) is used. Assume that we have two local schemas (Figure 1.b and 1.c) of two databases, namely *SESP* and *BIGGER*, represented as trees. We also assume a global schema (Figure 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 Doan and Wuwongse (2003)). Our task, query decomposition, is to process input queries so that they can extract relevant data from the two above local databases using the global schema. An example of an input XPath query is $Q_{global}$ = *'/department/mobile/products/jammer*[*price*<20]*'*, finding the content of all *jammer* elements having *price* less than 20, 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*<20]*'* for the *SESP* schema and $Q_{BIGGER}$ *='/department/mobile/jammer*[*price*<20]*'* for the *BIGGER* schema.

## 4 ASSUMPTIONS

In order to apply our algorithm, we make the following assumptions similar to those in Lausen and Marron (2002). There are several data sources participating in the system. Each of them, represented in XML, has its own local schema. Since these data sources are created by different designers, there often exist conflicts between their respective structures. Moreover, these schemas share a predefined global schema. For example, in Figure 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 schemas *SESP* and *BIGGER*. However, when data between such schemas need to be interoperated, they must all conform to the global schema as their integration view (see Figure 1). 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. Further, we assume that there are two built-in functions for finding the occurrences and the position of a node in an XML tree.

Moreover, we assume the following about the form of input and output queries: An input query according to the global schema has the form of a path expression $'/p_1/p_2/.../p_i/.../p_{n-1}/p_n'$. The decomposed subqueries according to the local subschemas have the form of path expressions $'[/][/p^*_1][/p^*_2]...[/p^*_i]...[/p^*_{n-1}]/p^*_n'$, where asterisks denote possible renamings, *'*[]*'* as in EBNF denotes optional parts, and a single *'/'* as in XPath denotes an (immediate) child part while *'//'* (possible because *'*[/]*'* can denote *'/'*) denotes an (arbitrarily remote) descendant. Since we assume that an arbitrary number of left parts can be omitted (cf. *'//'*) and that the right-most part must always be present (cf. $p_n$), our bottom-up strategy turns out to be superior to LM's top-down strategy: Because of the possible *'//'*-openness of a decomposed subquery down from the root, it will be worthwhile to first search for the fixed $p_n$ on the leaf level of a local schema and work (basically) upward from there.

## 5 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.

```
1  Input
2          A local schema S
3          A user query Q_global based on a global schema
4  Output
5          A decomposed query Subquery for S
6
7  Algorithm
8
9  Function BottomUpDecomposition(S, Q_global)
10 Anchor:= LeftmostLeafNode;
11 Subquery:='';
12 i:=|Q_global|;
13 repeat
14   if Check(P_i, Anchor)
15   {
16     if Subquery=''
17         Subquery:=P_i
18     else
19     {
20        if P_i=Anchor
21           Subquery:=P_i⊕'/'⊕Subquery
22        else
23           Subquery:=P_i⊕'//'⊕Subquery;
24     }
25     if IsRoot(P_i)
26         Subquery:='/'⊕Subquery
27     else
28     {
29   if (i>1)
30   % p_i is not the leftmost part of Q_global
31         Anchor:=father(P_i)
32   else
33         Subquery:='//'⊕Subquery
34     }
35   }
36   else
37   % P_i does not exist
38     if (Subquery <> '') and (i=1)
39                 Subquery:='//'⊕Subquery;
40 i:=i-1;
41 until (i=0) or (Subquery='') or IsRoot(P_i+1);
42 return Subquery;
```

**Figure 2** *Pseudo-code of the algorithm for finding a subquery*

```
1  Input
2          P: a node to be found
3          Anchor: current pointer
4  Output
5
6  If P is found, return Anchor pointing to P.
7  Otherwise, return Anchor pointing to nil.
8
9  Algorithm
10
11 Function Check (P, Anchor)
12
13 % We assume root.next=nil;
14 % For the first time a search is performed on the tree, we begin our
15 % search with leaf nodes and then their ancestors.
16 if Anchor = LeftmostLeafNode
17  {NodeName=''
18  Q:= Initialize the queue
19  Enqueue(Q, all leaf nodes from left to right
20  starting from the lowest level)
21  while (Anchor <> nil and Anchor <> P)
22  {
23   if (Anchor <> root) and (Anchor.father.name <> NodeName)
24       {Enqueue(Q,Anchor.father);
25        NodeName:=Anchor.father.name;}
26   Anchor:=Dequeue(Q);
27  }
28  if (Anchor <> nil)  return Anchor;
29 }
30 else
31 % Anchor is not at the leaf node
32 % We search for P from the ancestors of Anchor
33
34 {
35   while ((Anchor <> nil) and (Anchor <> P))
36         Anchor:=Anchor.father;
37   if (Anchor <> nil) return Anchor
38 }
39 return nil;
```

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

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 $Q_{global} = '/p_1/p_2/.../p_i/.../p_{n-1}/p_n'$ following a global schema into a subquery following the 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 in the local schema, we can immediately conclude that there is no decomposed 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 (i.e., the nodes on the path from the marked node to the root of the tree). We then sequentially take $p_i (i = (n - 1), (n - 2), ..., 1)$ of the query to evaluate. We check whether $p_i$ exists in the local schema, proceeding to the ancestor node(s) if it does 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 in a tree. If $p_i$ is found, it will be concatenated to the local subquery.

In the algorithm (Figure 2), ⊕ denotes concatenation. The flowchart in Figure 5 depicts the algorithm given in Figure 2. In order to explain our algorithm, we will walk though it using the two local schema examples of Figure 1.

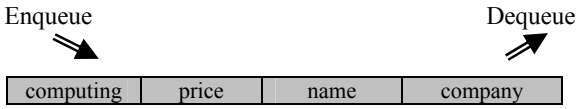



**Figure 4** *State of Q after execution of statements 19-20 from Figure 3 for the BIGGER schema*

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 Figure 2 checks

whether $p_i$ exists from the *Anchor* up to the root node. The main task of *Check*(*P*, *Anchor*) is to find the 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. In order to achieve this goal, we construct a queue $Q$ as follows. All the leaf nodes in the tree of a local schema are enqueued, beginning from the leaves at the lowest level and proceeding to the leaves of successively higher levels, always in a left-to-right manner. The state of $Q$ after execution of statements 19-20 from Figure 3 for the *BIGGE*R schema (see Figure 1) is as given in Figure 4.

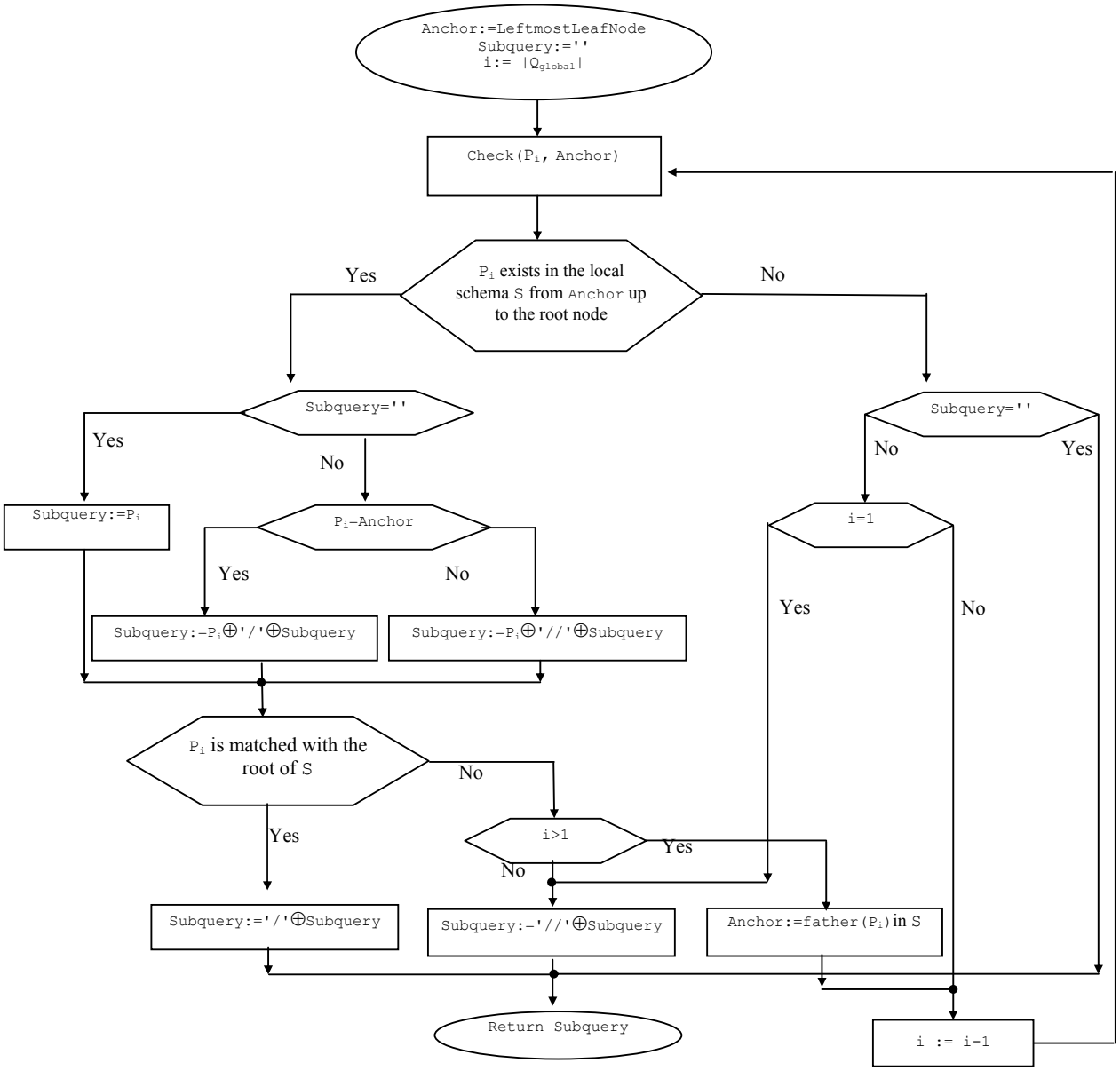



**Figure 5** *Flowchart of the algorithm in Figure 2 for finding a subquery*

If it is the first time *Check*(*P*, *Anchor*) is called, we begin our search with the leaf nodes enqueued in $Q$. Each time we dequeue 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 enqueue the father node in $Q$. After processing all the leaf nodes, we only need to process the remaining nodes in $Q$ (i.e., all father nodes of leaf nodes). We apply this strategy iteratively until $Q$ is empty. Note that we might visit all nodes of the tree, bottom-up, in the worst case. With this strategy each node is

visited at most once. If the previous *Check*(*P*, *Anchor*) call has found a node, for the next calls we simply search the ancestors of that previously marked node, i.e., *Anchor*.

### 5.1 Example 1

In this example, from a global query (with $n = 4$) $Q_{global}$ =' */department/mobile/products/jammer'*, we produce a subquery for the local schema *SESP* (Figure 1.b) using the algorithm of Figure 2. 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* := $p_3 \oplus$ '/' $\oplus$ '*jammer*' (i.e., *Subquery :='products/jammer'*). Now, $p_3$ ='*products*' is the root node of the *SESP* schema and the algorithm stops. We find that the local query for the *SESP* schema is *Subquery :='products/jammer'*.

### 5.2 Example 2

In this example, we produce a subquery for the schema *BIGGER* (Figure 1.c). Again 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 *mobil*e 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, the algorithm stops. The subquery found for the *BIGGER* schema is *Subquery* :=' */department/mobile/jammer'*.

## 6 ADDITIONAL CASES OF INPUT QUERIES

### 6.1 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*<20]', 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 considerably the 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 Figure 6 (a subschema of Figure 1.b), we can quickly recognize that the corresponding subquery for this schema does not exist when we first transform *pric*e.

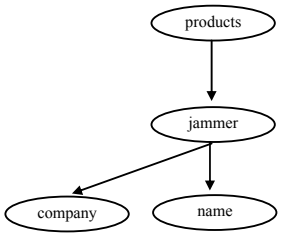



**Figure 6** *A local schema without the **price** leaf node (adapted from Lausen and Marron (2002))*

Moreover, if constraints of input queries are more general (e.g., [*/price* < 20] *or* [*//price* < 20]), we can separately apply our algorithm for those constraints before transforming the whole query.

### 6.2 Conflicts Between Schemas

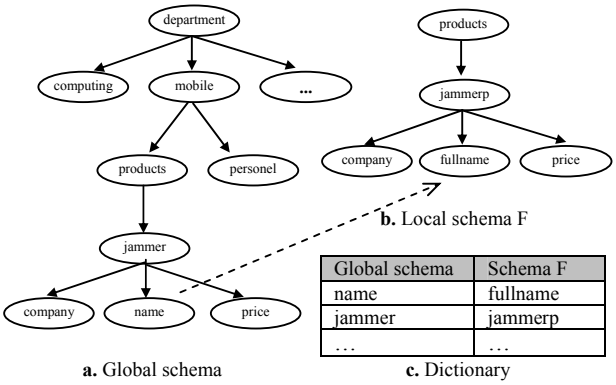
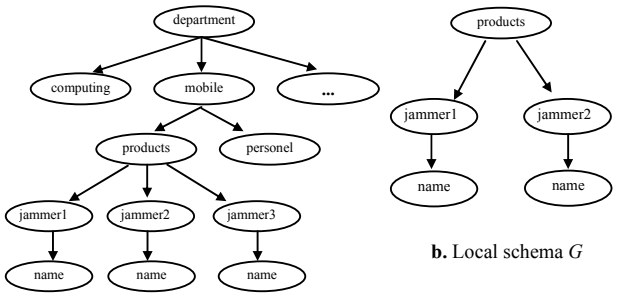



**Figure 7** *Naming conflicts between a global schema and a local schema (adapted from Lausen and Marron (2002))*

Our algorithm shows mismatches between the global schema and local ones concerning their structures. Other naming conflicts (Doan and Le (2005)), such as synonyms and homonyms, are not considered here. However, we can resolve those conflicts using an ontology or a dictionary. Even though naming conflicts can be better resolved using an ontology than a dictionary, it is beyond the scope of this paper (see Doan and Le (2005), Doan and Wuwongse (2003) for a combination of ontologies and rules for schema integration, and Hakimpour and Geppert (2005) for a

utilization of similarity relations between formal ontologies to handle semantic heterogeneity in schema integration for federated databases). In our algorithm, we use 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 Figure 7, 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 (Figure 7.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.

### 6.3 Leaf Nodes with the Same Label




**a.** Global schema

**b.** Local schema *G*

**Figure 8** *Homonym conflicts in a global schema and a local schema*

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 (Figure 8.a) and the local schema (Figure 8.b). This situation is actually a homonym conflict, a special type of the naming conflict. As stated in Section 4, our algorithm cannot be applied directly in the presence of naming conflicts. For example, given a query $Q_{global}$:='*jammer*1/*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 *jammer*1 in *G*, we obtain that a local query for *G* is *Subquery* :='//*jammer*1/*name*'. Otherwise, if *Anchor* points to *name*, which is the child node of *jammer*2 in *G*, we only obtain *Subquery* :='//*name*'. This local query '//*name*' will provide more solutions than we want (extracted data are *name* of both *jammer*1 and *jammer*2 in *G*). However, with the top-down strategy (Lausen and Marron (2002)), the local query produced is *Subquery* :='//*jammer*1/*name*', which is a better solution. These conflicts must be resolved 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 Doan and Le (2005) and Doan and Wuwongse (2003) for details).

## 7 ALGORITHM ANALYSIS

In Lausen and Marron (2002), 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, which can be 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 Lausen and Marron (2002) for the whole query given a full $k$-ary tree is

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

However, as we have discussed in Section 2, this top-down query decomposition algorithm is not always 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 occurs for 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). Recall that the *Check*(*P*, *Anchor*) algorithm can find $p_n$ in a local schema from the leaf nodes up to the root node such that the number of visited nodes in the worst case is equal to the number of nodes in the tree. Therefore, the time complexity of our algorithm is

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

for a binary tree, and

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

for a full *k*-ary tree (the total nodes in a tree). In the best case, the rightmost part $p_n$ matches with a leaf node of the tree at the first node, and the same happens 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 *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.

## 8 CONCLUSION

We have proposed a bottom-up algorithm for query decomposition without predefined mappings. The algorithm can be applied to distributed XML-based data sources, which may contain conflicts between their respective structures. Having the same motivation as Lausen and Marron (2002) 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. In the worst case, our algorithm is *n* times better than that of Lausen and Marron (2002). In the best case, the time complexity of our algorithm is only

$$T(n, k, h) = min(n, h)$$

compared to

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

of Lausen and Marron (2002).

(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 (Ludascher et al. (1998), May (2005)).

## ACKNOWLEDGEMENT

We would like to thank the Vietnamese Government and the Canadian NSERC for their financial support of this work.

## REFERENCES

Baru, C. *et al*. (1999) 'XML-Based Information Mediation with MIX', *Proceedings of the International Conference on Management of Data*, Philadelphia, USA, ACM SIGMOD, pp.597–599.

Baru, C. *et al*. (1998) 'Features and Requirements for an XML View Definition Language: Lessons from XML Information Mediation', In *Query Languages Workshop*, Boston, USA.

Bayardo *et al*. (1997) 'InfoSleuth: agent-based semantic integration of information in open and dynamic environments', *Proceedings of the International conference on Management of Data*, Tucson, USA, ACM SIGMOD, pp.195–206.

Bi, Y. and Lamb, J. (2001) 'Facilitating Integration of Distributed Statistical Databases Using Metadata and XML', *Proceedings of New Techniques and Technologies in Statistics*, Crete, Greece, pp. 769–774.

Doan, A.H. and Halevy, A. (2005) 'Semantic Integration Research in the Database Community: A Brief Survey', *AI Magazine, Special Issue on Semantic Integration.*

Doan, D.D. and Le, T.T.T. (2005) 'Classification and Reconcilement of Conflicts between Heterogeneous XML Schemas', *Proceeding of the 10th Conference on Artificial Intelligence and Applications*, Kaohsiung, Taiwan, pp. 1066–1075.

Doan, D.D. and Wuwongse, V. (2003) 'XML Database Schema Integration Using XDD', *Proceedings of Advances in Web-Age Information Management Conference (WAIM03)*, Chengdu, China, LNCS Vol. 2762, pp. 1066–1075.

Hakimpour, F. and Geppert, A. (2005) 'Resolution of Semantic Heterogeneity in Database Schema Integration Using Formal Ontologies', *Inf. Tech. and Management*,Vol. 6, No. 1, pp.97–122.

Lausen, G. and Marron, P.J. (2002) 'Adaptive Evaluation Techniques for Querying XML-based E-Catalogs', *Proceedings of the Twelfth International Workshop on Data Engineering*, San Jose, USA, pp. 19–27.

Le, T.T.T. and Duong, D.D. (2005) 'Query Decomposition Using the XML Declarative Description Language', *Proceedings of International Conference of Computational Science and Its Applications, Singapore*, LNCS, Vol. 3481, pp.1066–1075.

Le, T.T.T. and Duong, D.D. (2004) 'Integration of XML Databases', *Journal of Hue University*, Vol. 22, pp.45– 52.

Le, T.T.T. and Wuwongse, V. (2003) 'Query Processing of Integrated XML Databases', *The Fifth International Conference on Information Integrationand Web-based Applications and Services (iiWAS2003)*, Jakarta, Indonesia, Vol. 170, pp.335–344.

Ludascher *et al*. (1998) 'Managing Semistructured Data with FLORID: A Deductive Object-Oriented Perspective', *Journal of Information Systems*, Vol. 23, No. 8, pp.589–613.

May, W. (2005) 'Logic-based XML data integration: a semi-materializing approach', *Journal of Applied Logic*, Vol. 3, No. 1, pp.271–307.

McCann, R. *et al*. (2005) 'Mapping maintenance for data integration systems', In *VLDB*, pp. 1018–1030.

Rodriguez-Gianolli, P. and Mylopoulos, J. (2001) 'A Semantic Approach to XML-based Data Integration', *20th International Conference on Conceptual Modeling*, Yokohama, Japan, LNCS, Vol. 2224, pp. 117–132.

'XML Path Language (XPath). Version 1.0', http://www.w3.org/TR/xpath. Last accessed: Febuary 07.