Fragmentation and Query Decomposition in XML

Research Report
RP-353

Porto Alegre, April 2006
ACKNOWLEDGMENTS

This research is partially supported by CNPq (XML-Broker Research Project, Edital Universal – Digitex) and FAPERGS/CNPq (Recuperação Semântica, PRONEX).
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>4</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>5</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>6</td>
</tr>
<tr>
<td>2 DATA MODEL AND QUERY LANGUAGE</td>
<td>8</td>
</tr>
<tr>
<td>2.1 Conceptual Model</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Conceptual Base</td>
<td>9</td>
</tr>
<tr>
<td>2.3 CXPath</td>
<td>10</td>
</tr>
<tr>
<td>3 FRAGMENTATION OPERATORS</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Local Conceptual Model</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Split Operator</td>
<td>13</td>
</tr>
<tr>
<td>3.3 Vertical Fragmentation</td>
<td>15</td>
</tr>
<tr>
<td>3.4 Horizontal Fragmentation</td>
<td>16</td>
</tr>
<tr>
<td>4 MAPPING BETWEEN LOCAL MODEL AND XML SOURCE</td>
<td>19</td>
</tr>
<tr>
<td>5 DECOMPOSITION ALGORITHM</td>
<td>22</td>
</tr>
<tr>
<td>5.1 Parse Query</td>
<td>22</td>
</tr>
<tr>
<td>5.2 Handle Horizontal Fragments</td>
<td>23</td>
</tr>
<tr>
<td>5.3 Navigate Inside Fragments</td>
<td>24</td>
</tr>
<tr>
<td>5.4 Join Split Fragments</td>
<td>26</td>
</tr>
<tr>
<td>5.5 Handle Selection Predicates</td>
<td>26</td>
</tr>
<tr>
<td>5.6 Recursive Call</td>
<td>28</td>
</tr>
<tr>
<td>5.7 Handle Vertical Fragmentation</td>
<td>30</td>
</tr>
<tr>
<td>6 CONCLUSION AND FUTURE WORK</td>
<td>33</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>34</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A conceptual model and some XML instances</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>An example of conceptual base</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Global conceptual model for the examples of fragmentation</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>Local conceptual models produced through split fragmentation operations</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>Example of vertical fragmentation of $LCM_1$ (Figure 3.2)</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>Example of horizontal fragmentation</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>Schema for $LCM_2$ XML source</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Schemas for $LCM_3$ XML sources</td>
<td>20</td>
</tr>
<tr>
<td>5.1</td>
<td>XQuery generated by the decomposition algorithm to example 2.1</td>
<td>28</td>
</tr>
</tbody>
</table>
ABSTRACT

The problem of data integration (query decomposition, data fragmentation) has been widely studied in literature, but the inherent hierarchical nature of XML data presents problems that are specific to this data model. Each many-to-many conceptual relationship must be mapped to a specific hierarchical structure in XML. Different XML sources may implement the same many-to-many conceptual relationship in different ways. In our approach the problem of integration of XML data sources is decomposed in two problems: (1) that of fragmentation of a global graph-like model (e.g., an ER model) into several local graph-like models conceptually representing data sources and (2) that of mapping the local graph-like model into an XML tree-like schema. This paper presents a set of fragmentation operators specifically designed for our approach, as well as a query decomposition mechanism that allows a query stated at the conceptual level to be decomposed into an XQuery statement at the XML level. As the query language at the conceptual level, we adopt CXPath (conceptual XPath) a query language we have defined in previous work.

**Keywords:** Database fragmentation, Query decomposition, XML.
1 INTRODUCTION

In this paper we handle the problem of querying several XML data sources that possibly have different schemata and relate to a common domain. We apply the mediator approach, in which queries are submitted against a mediated or global schema. In this context an important decision is to choose the abstraction level of the data model used at the global level. One approach taken by some authors is to use the same abstraction level at the global and local levels [20, 6, 12, 13]. In the integration of XML sources this means that the global schema is an XML model. This approach has the potential advantage of simplifying the translation of queries from the global schema to the local schemata, but presents also a major problem: due to the inherent hierarchical structure of XML documents, non-hierarchical conceptual constructs like many-to-many relationships must be hierarchically represented in the XML schemata.

In previous work we have followed another approach, namely that of using a more abstract data model at the global level. In [16, 17, 7] we propose the use of an entity-relationship like conceptual model for the global and local levels, and show how each local XML model is abstracted to a local conceptual model, and how these local conceptual models are integrated in the global conceptual model. Queries against the conceptual model are stated in CXPath (conceptual XPath), an XPath based language, and translated into regular XPath by a query rewriting approach [4].

However, the mechanism proposed in [4] is limited to the translation of one source at a time. The local queries generated by this mechanism are stated against one single source, limiting this approach to queries that don’t need to interact between several sources to build the answer. In this work we address this limitation, that is how to decompose a global query stated against the conceptual model into local queries stated against several sources. Given this context, there are two major problems which need to be handled. One is to define how a global conceptual model is fragmented into several local models. The other problem is to define an algorithm, based in the fragmentation of the models, that will decompose a global query into several local queries and then construct the answer.

In this paper, we propose a solution for both problems. First, fragmentation operators for XML data are introduced, considering the existence of a global conceptual model. We define three fragmentation operators against the conceptual model and the conceptual base: split fragmentation, vertical fragmentation and horizontal fragmentation. Additionally, we propose a query decomposition algorithm, based on the use of mapping information between the conceptual model and the XML documents. This algorithm generates XPath expressions to access the individual sources,
and then integrates this queries into a single XQuery [24] expression, handling the sources fragmentation and constructing the answer for the query. The behavior of the algorithm is based on how the sources are fragmented. It means that it implements fragmentation reduction algorithms, similarly to what is done in distributed databases [19].

The problem of querying integrated sources has been investigated by several authors [15, 23, 11, 14, 21]. An approach that is similar to ours is [14]. In this approach queries are also stated at a conceptual level. Queries are expressed in a constrained user interface, binded to a domain ontology, and then decomposed to the individual databases. The difference to our work is that the sources are not XML documents, but relational databases instead. An approach that integrates XML data sources is the one proposed in [15]. In this approach, the queries are stated against an hierarchical global model (XML model), instead of against a graph-like global model as we propose in this work. In [11], the mediator schema is object-oriented like, and the query language defined to query this model is a variant of OQL. The approach that is more closely related to ours is the YAKOB mediator system [21]. In YAKOB queries against several sources related to a common domain are stated against a concept-based integration model based on RDF and then translated to the individual sources. The queries are expressed in CQuery, a language to express queries over the ontological model. CQuery is based on XQuery, whereas the language we apply is based on XPath. Another difference between YAKOB and the approach presented in this paper is that the our approach bases on a set of fragmentation operators specifically designed for the decomposition of a conceptual schema into XML schemata.

This paper is organized as follows. Section 2 presents the data model and the query language used in our approach. Section 3 describes the fragmentation operators. Section 4 describes the mapping information and Section 5 describes the decomposition algorithm and its application to an example. Section 6 is dedicated to the conclusions and future work.
2 DATA MODEL AND QUERY LANGUAGE

This section contains a formal definition of the conceptual model and the corresponding conceptual base that are used in our approach.

2.1 Conceptual Model

The conceptual model is a high level abstraction of the XML data source models. It is build by a bottom-up approach. First, each local XML model is abstracted to a local conceptual model and after that local conceptual models are integrated resulting in a global conceptual model [16, 17].

The conceptual model is a simplified version of an ORM model [9]. It defines concepts and relationships between concepts. There are two kinds of concepts: lexical and non-lexical. Lexical concepts represent objects that have a textual content. Lexical elements abstract atomic XML elements, like #PCDATA elements or attribute values. Non-lexical concepts do not have a direct textual representation and are abstractions of XML elements that contain other elements. Relationships at the conceptual schema level are abstractions of two kinds of constructs at the XML levels. A relationship may have an a navigational implementation, i.e. it may represent an access path relationship at the XML level. A relationship may further have an associative implementation, i.e, it may represent an identifier reference between data in two different XML data sources. For details on the process of abstraction of an XML model into a conceptual model, please refer to [17]. A non-lexical concept may have an identifier. An identifier is a set of relationships that uniquely identifies an instance of the concept.

Figure 2.1 shows an example of a conceptual model for students enrollment in an university. There are three non-lexical concepts: Course, Enrollment and Student, which are represented by solid rectangles. Each of these concepts is related to at least one lexical concept (dotted rectangle). For example, Student is related to Name and Number. The diagram represents also the cardinality of the relationships between concepts. For example, in this model, Course and Enrollment have a one-to-many relationship. The identifier relationships are labeled with ‘ID’ (for example Student is identified by the relationship with Number).

The model in Figure 2.1 may have several different implementations in XML. In Figure 2.1 two XML instances with different schemata but corresponding to this conceptual model are depicted: one that implements the many-to-many relationship between student and courses from the student perspective, i.e., the student element contains the courses in which he is enrolled, and the other that implements the relationship from the course perspective, i.e, the course element contains the students
A conceptual model is given in definition 2.1.

**Definition 2.1** A conceptual model is a tuple $CM = < NL, L, R >$, where:

- $NL$ is the set of non-lexical concepts.
- $L$ is the set of lexical concepts.
- $R$ is the set of relationships between concepts. A relationship $r \in R$ is a tuple $< c_1, c_2, c_d, c_i, \{n\} >$, where $c_1$ and $c_2$ are concepts, with $c_1 \in NL$ and $c_2 \in NL \cup L$, and $c_d$ and $c_i$ are the cardinalities, where $c_d$ is the direct cardinality - from $c_1$ to $c_2$ - and $c_i$ is the inverse cardinality - from $c_2$ to $c_1$. A cardinality $c_d$ or $c_i$ is a tuple $< \text{min}, \text{max} >$, where min is the minimum cardinality and max is the maximum cardinality. $n$ is the optional relationship name.
- The function $ID(c) \rightarrow ID_c$ returns the set of identifiers of a given concept $c \in NL$. An identifier $id \in ID_c$ is a relationship $r \in R$, where $r.c_1 = c$ and $r.c_2 \in L$.

### 2.2 Conceptual Base

The conceptual base is an abstract database containing the instances of a given conceptual model. It is an abstract representation of an XML document. A formal definition of the conceptual base is given in definition 2.2.
Definition 2.2 A conceptual base of a given conceptual model CM is a tuple $CM_B =< NL_B, L_B, R_B >$, where:

- $NL_B$ is the set of non-lexical instances. A non-lexical instance $nl \in NL_B$ is a tuple $< oid, c >$, where $oid$ is the object id and $c \in CM.NL$.

- $L_B$ is the set of lexical instances. A lexical instance $l \in L_B$ is a tuple $< c, v >$, where $c \in CM.L$ and $v$ is the value.

- $R_B$ is the set of relationship instances. A relationship instance $r \in R_B$ is a tuple $< i_1, i_2 >$, where $i_1 \in NL_B$, $i_2 \in NL_B \cup L_B$, and $< i_1.c, i_2.c > \in CM.R$.

- The conceptual base $CM_B$ obeys the identifier and cardinality constraints defined in the conceptual model CM.

2.3 CXPath

CXPath [4] is a query language, based in XPath [24] syntax, which is used to build queries over a conceptual schema. The queries written in CXPath are path expressions over the concepts of the global schema. It’s also possible to write selection predicates in the path expression.

Below are represented some examples of CXPath over the conceptual model of figure 3.1.

Example 2.1. Retrieve the name of the students registered in the course with code “INF001” in the semester “2006/1”.

```
/Course[CourseCode="INF001"]/Enrollment[Semester="2006/1"]
/Student/Name
```

Example 2.2 Retrieve the name of the professors of the “Database” department who taught in the year “2001”.

```
/Teaches[Period="2001"]/Professor[Department="Databases"]/Name
```
Several authors in literature address the problem of fragmentation of XML documents [1, 2, 12, 13, 3]. However, none of them considers a graph-like global model when defining fragmentation operators.

One natural approach to handle the problem of fragmentation of XML documents is to look at the past work on distribution design of relational databases [19] and object-oriented databases and adapt these concepts to the XML data model. This is what has been done in [12, 13], where the horizontal and vertical fragmentation operations of relational databases [18, 5] were adapted to the XML model. Additionally, a fragmentation operation, called split fragmentation, was taken from the object-oriented data model [22].

In this section, a distribution specification for XML documents that are described by a graph-like conceptual model is presented. In contrast to Schewe’s approach [12, 13], the fragmentation operators are applied over the conceptual model and the conceptual bases, instead of the XML documents. The result of the fragmentation process is a set of local conceptual models and local conceptual bases, which are then directly mapped to XML documents by the process described in [17].

Notice that our approach has the advantage of clearly separating two different problems, the problem of fragmentation and the problem of abstraction of implementation details in XML. If we apply fragmentation directly at the XML model level, we will have to deal with the problem that the same conceptual information may have several different representations at the XML level. In our approach, documents that have different representation but have conceptually the same content will be described by a single local conceptual model.

Below, we define first the local models that are obtained by the fragmentation operators and after that we discuss the fragmentation operators themselves.

### 3.1 Local Conceptual Model

A local conceptual model is a subset of a given global conceptual model. A global conceptual model can be fragmented into several local models through fragmentation operators. The operators that produce local conceptual models are the vertical fragmentation and the split fragmentation, which will be detailed below. After the fragmentation, it should be possible to reconstruct the conceptual model from the several local models. This means that local models must contain some specific information in order to allow this reconstruction.

When two non-lexical concepts that are associated by a relationship in the global conceptual model are fragmented into different local models, identifier references...
are to be included in these local models. An identifier reference is an information that implements a relationship between two local conceptual models in the same way a foreign-key implements a relationship between two relations in a relational database. The local model in which an identifier reference is included will depend on the cardinality of the relationships. Each identifier reference is mapped to the identifier of the related non-lexical concept. This will be explained in detail in the split fragmentation.

Figure 3.1 shows an example of global conceptual model and Figure 3.2 shows local models that may obtained by fragmentation of this conceptual model. The non-lexical concepts Enrollment, Course, Professor and Teaches, which are related in the global conceptual model shown in Figure 3.1, were fragmented into the local conceptual models in Figure 3.2. Some relationships in the global model are implemented in the local models through identifier references. For example, take the relationship between Enrollment and Course that appears at the global level. As the concepts Enrollment and Course are distributed in two different local models (LCM3 and LCM2 respectively) the relationship at the global conceptual level must be implemented through an identifier reference at the local level. In this example, the concept CourseCode was added to the local model LCM3. The instances of this concept are references the Course identifier in local model LCM2. The same idea is valid for the concept ProfessorEmail in LCM2.

A formal definition of the local conceptual model is given in definitions 3.1 and 3.2.

**Definition 3.1** A sub conceptual model $CM'$ of a given global conceptual model $GCM = < NL, L, R >$ is tuple $CM' = < NL', L', R' >$, where:

- $CM'.NL' \subseteq GCM.NL$.
- $CM'.L' \subseteq GCM.L$.
- $CM'.R' \subseteq GCM.R$. For all relationships $r \in CM'.R'$, $r.c_1 \in CM'.NL'$, $r.c_2 \in CM'.NL' \cup CM'.L'$.
Figure 3.2: Local conceptual models produced through split fragmentation operations

- $CM'.ID(nl') \subseteq GCM.ID(nl)$, where $nl' \in CM'.NL'$, $nl \in GCM.NL$, and $LCM'.ID(nl') \subseteq CM'.R'$.

**Definition 3.2** A local conceptual model is a tuple $LCM = < CM', IDREF >$, where:

- $CM'$ is a sub conceptual model of a given global conceptual model $CM$.

- $IDREF$ is the set of identifier references. An identifier reference $idref \in LCM.IDREF$ is a tuple $< ref, DEST >$, where $ref \in LCM.R'$ and $DEST$ is a set of tuples $< LCM_d.id >$, where $LCM_d$ is the destination local conceptual model, and $id \subseteq LCM_d.ID(nl)$, where $nl \in LCM_d.NL$.

### 3.2 Split Operator

The split fragmentation operator is based on the operator presented in [12], where a given element of an XML document is replaced by a reference to a new element. This operator originates from previous work on object-oriented databases [22]. There, a complex operation in a class is replaced by a reference to a new class. In this work, the split operator has a different behavior. It takes a conceptual model and divides it into two distinct local conceptual models. The relationships in the global model are split into different local models and implemented through identifier references. The creation of identifier references depends upon the cardinality of the relationships.

Figure 3.2 presents an example of the result of a split operation applied to the conceptual model in figure 3.1. To produce the three local conceptual models shown in this example, it is necessary to apply the split operator twice. First, the global conceptual model is split into the local conceptual model $LCM_1$ and the local model $LCM_{tmp}$. This last local model is then split into local models $LCM_2$ and $LCM_3$. 
The first split operation is

\[ SPLIT(CM, \{\text{Professor}\}) \rightarrow < LCM_1, LCM_{tmp} > \]

The split operator takes a conceptual model \((CM)\) and a set of non-lexical concepts \((\{\text{Professor}\})\), and creates two local conceptual models \((LCM_1\) and \(LCM_{tmp}\)). The first local model \((LCM_1)\) contains the non-lexical concepts specified as parameter \((\{\text{Professor}\})\), together with all lexical concepts related to these non-lexical concepts, including identifiers and identifier references, as well as all relationships among them. The second local model \((LCM_{tmp})\) includes the remaining concepts and the relationships among them. The relationships between non-lexical concepts separated into the different local models are replaced by identifier references among the local conceptual models. In the example, the relationship between \(\text{Teaches}\) and \(\text{Professor}\) that appears at the global level is replaced by a reference from \(LCM_{tmp}.\text{Teaches}\) to \(LCM_1.\text{Professor}\).

The second split operation is

\[ SPLIT(LCM_{tmp}, \{\text{Course, Teaches}\}) \rightarrow < LCM_2, LCM_3 > \]

This operation results in the local model \(LCM_2\) containing the concepts \(\text{Course}\) and \(\text{Teaches}\) and in the local model \(LCM_3\) containing the remaining concepts. The relationship between \(\text{Enrollment}\) and \(\text{Course}\) that appears in \(LCM_{tmp}\) is replaced by a reference from \(LCM_3.\text{Enrollment}\) to \(LCM_2.\text{Course}\).

Definitions 3.3 and 3.4 formalize these ideas. Definition 3.3 is an auxiliary definition that addresses the identifier reference creation. It defines a function that implements a global relationship between two non-lexical concepts, \(c_1\) and \(c_2\), in two local conceptual models, through the creation of identifier references. The result of this function is a local conceptual model with a set of identifier references to the destination concept, \(c_2\), located into a different local conceptual model.

**Definition 3.3** Let \(GCM\) be a conceptual model, \(r\) be a relationship in \(GCM\), and \(LCM_1\) and \(LCM_2\) be local conceptual models. The function

\[ \text{generateIDRefs}(GCM, r, LCM_1, LCM_2) \rightarrow LCM_1 \]

generates the identifier references in \(LCM_1\) that implement the relationship \(r\). Definition 3.4 handles the split operator itself.

- \(\text{ID}(r.c_2).c_2 \subseteq LCM_1.L\).
- \(\{r_1, r_2, \ldots, r_n\} \subseteq LCM_1.R\), where \(r_i.c_1 = r.c_1\), \(r_i.c_2 \in GCM.ID(r.c_2).c_2\), \(r_i.c_i = < 1, 1 >\) and \(r_i.c_d = r.c_i\).
- \(\{\text{idr}_1, \text{idr}_2, \ldots, \text{idr}_n\} \subseteq LCM_1.IDREF\), where \(\text{idr}_i.ref \in LCM_1.R\), \(\text{idr}_i.DEST.LCM_d = \{LCM_2\}\) and \(\text{idr}_i.DEST.id \subseteq LCM_2.ID(r.c_2)\).

**Definition 3.4** The function \(SPLIT(CM, C) \rightarrow < LCM_1, LCM_2 >\) defines the split fragmentation operation, where \(CM\) is a (local) conceptual model, \(C\) is a set of non-lexical concepts \(C \subseteq CM.NL\), and \(LCM_1\) and \(LCM_2\) are local conceptual models produced as output, such that:

- \(LCM_1.NL = C\).
• \( LCM_1.L \) such that, for all \( r \in CM.R \), where \( r.c_1 \in C \) and \( r.c_2 \in CM.L \), \( r.c_2 \in LCM_1.L \).

• \( LCM_1.R \) such that, for all \( r \in CM.R \), where \( r.c_1 \in C \) and \( r.c_2 \in C \cup CM.L \), \( r \in LCM_1.R \).

• \( LCM_1.ID(nl_1) \) such that, for all \( id \in CM.ID(nl_2) \), where \( nl_1 \in LCM_1.NL \), \( nl_2 \in CM.NL \), \( id.c_1 \in C \), then \( id \in LCM_1.ID(nl) \).

• \( LCM_1.IDREF \) such that, if \( CM \) is a local conceptual model, then for all \( idref \in CM.IDREF \), where \( idref.c_1 \in C \), \( idref \in LCM_1.IDREF \).

• \( LCM_2.NL = (CM.NL - C) \).

• \( LCM_2.L \) such that, for all \( r \in CM.R \), where \( r.c_1 \in (CM.NL - C) \) and \( r.c_2 \in CM.L \), \( r.c_2 \in LCM_2.L \).

• \( LCM_2.R \) such that, for all \( r \in CM.R \), where \( r.c_1 \in (CM.NL - C) \) and \( r.c_2 \in (CM.NL - C) \cup CM.L \), \( r \in LCM_2.R \).

• \( LCM_2.ID(nl_2) \) such that, for all \( id \in CM.ID(nl_1) \), where \( nl_1 \in CM.NL \), \( nl_2 \in LCM_2.NL \), \( id.c_1 \in (CM.NL - C) \), then \( id \in LCM_2.ID(nl_2) \).

• \( LCM_2.IDREF \) such that, if \( CM \) is a local conceptual model, then for all \( idref \in CM.IDREF \), where \( idref.c_1 \in (CM.NL - C) \), \( idref \in LCM_2.IDREF \).

• for all relationships \( r \in CM.R \), where \( r.c_1 \in LCM_1.NL \) and \( r.c_2 \in LCM_2.NL \), then:

  - If \( r.c_d = < 0, 1 > \) and \( r.c_i = < 1, 1 > \), generateIDRef\( (CM, r, LCM_2, LCM_1) \subseteq LCM_2 \).

  - If \( r.c_d = < 0, 1 > \) and \( r.c_i = < 0, 1 > \), then two disjunct definitions are valid:

    1. generateIDRef\( (CM, r, LCM_1, LCM_2) \subseteq LCM_1 \).

    2. generateIDRef\( (CM, r, LCM_2, LCM_1) \subseteq LCM_2 \).

  - If \( r.c_d.max = 1 \) and \( r.c_i.max = N \), then

     generateIDRef\( (CM, r, LCM_1, LCM_2) \subseteq LCM_1 \).

3.3 Vertical Fragmentation

Vertical fragmentation is based on the corresponding relational operator [18], which produces fragments projecting subsets of attributes. In this work, each fragment contains subsets of lexical concepts of the conceptual model.

Figure 3.3 shows an example of vertical fragmentation. The local conceptual model \( LCM_1 \) in Figure 3.2 is fragmented into two local conceptual models: \( LCM_{V1} \) and \( LCM_{V2} \). The operations that produces this models are the following:

\[
VF(LCM_1, \{Departments\}) \rightarrow LCM_{V1}
\]
This operator takes a conceptual model and a parameter \( C \), containing a set of lexical concepts, and produces a fragmented conceptual model \( LCM_{V} \). The fragmented model contain only the lexical concepts specified in \( C \), plus all the identifier and identifier reference concepts. Definition 3.5 formalizes this idea.

**Definition 3.5** The vertical fragmentation operation is a function \( VF(CM, C) \rightarrow LCM_{V} \), where \( CM \) is a (local) conceptual model, \( C \) is a set of lexical concepts \( C \subseteq CM.L \), and \( LCM_{V} \) is the local conceptual model produced as output, such that:

- \( LCM_{V}.NL = CM.NL \).
- \( r \in LCM_{V}.R \) such that \( r.c_{1} \in CM.NL \) and \( r.c_{2} \in (CM.NL \cup C \cup CM.ID(r.c_{1}).c_{2} \cup CM.IDREF.ref.c_{2}) \).
- \( LCM_{V}.L = (C \cup CM.ID.c_{2} \cup CM.IDREF.ref.c_{2}) \).
- \( LCM_{V}.ID(nl) = CM.ID(nl) \), for all concepts \( nl \in CM.NL \).
- \( LCM_{V}.IDREF = CM.IDREF \), if \( CM \) is a local conceptual model.

### 3.4 Horizontal Fragmentation

As in vertical fragmentation, horizontal fragmentation is based on the corresponding relational operator [5], which produces fragments that correspond to subsets of the tuples. This operator produces fragments that correspond to a subset of the instances of a given conceptual base. Each fragment is defined through a selection predicate. If the instance obeys the predicate, it is included in the fragment.

As the relational operator, this operator produces two kinds of horizontal fragments: primary and derived. A primary fragment contains all instances selected by the direct application of the selection predicate. A derived fragment includes all instances that are related to the instances selected in the primary fragment or to other instances in the derived fragment.

Figure 3.4 shows an example of horizontal fragmentation. In this figure, small circles represent concept instances and lines represent relationship instances. A sample conceptual base of the local conceptual model \( LCM_{3} \) (Figure 3.2) is horizontally fragmented through the following operation: \( HF(LCM_{3}, \sigma) \), where the selection predicate \( \sigma \) is (semester='2005/1'). The black node in Figure 3.4 corresponds to a direct match of an instance in the conceptual base with the selection predicate. The
grey nodes belong to the derived fragments. The horizontal fragment is equal to the selected data graph.

The definition of horizontal fragmentation is given below. First, the function \( \text{getDataGraph}() \) is defined. This function is responsible for obtaining the derived fragments for a given primary fragment. This function is specified in Definition 3.6. The Definition 3.7 defines the horizontal fragmentation operator. It takes a conceptual base and a parameter \( \sigma \), containing a selection predicate, and produces a fragmented conceptual base. The fragmented conceptual base includes all the primary and derived fragments.

**Definition 3.6** The function \( \text{getDataGraph}(CM_B, i_R) \rightarrow CM_B' \) returns a conceptual base containing the data graph which a given instance \( i_R \) belongs. Given a conceptual base \( CM_B \) and an instance \( i_R \in (CM_B.L_B \cup CM_B.NL_B) \), a conceptual base \( CM_B' \) is mapped as the function output, such that:

- For all instances \( i_d \in (CM.L_B \cup CM.NL_B) \), if exists a set of instances \( \{i_1, i_2, \ldots, i_{n-1}\} \subseteq (CM.L_B \cup CM.NL_B) \) and a set of relationships \( \{r_1, r_2, \ldots, r_n\} \), where \( r_1.i_1 = i_1, \ r_1.i_2 = i_R, \ r_2.i_1 = i_1, \ r_2.i_2 = i_2, \ r_3.i_1 = i_2, \ r_2.i_2 = i_3, \ldots, \ r_n.i_1 = i_{n-1}, \ r_n.i_2 = i_d \), then if \( i_d \in CM_B.NL_B \), then \( i_d \in CM_B'.NL_B \), else \( i_d \in CM_B'.L_B \).
- For all relationships \( r \in CM_B.R_B \), if \( r.I_1 \in CM_B'.NL_B \) and \( r.I_2 \in (CM_B'.NL_B \cup CM_B'.NL_B) \), then \( r \in CM_B'.R_B \).

**Definition 3.7** The function \( HF(CM_B, \sigma) \rightarrow CM_B1 \) defines the horizontal fragmentation operator, where \( CM_B \) is a conceptual base of the conceptual model \( CM \), \( \sigma \) is the selection predicate and \( CM_B1 \) is conceptual base produced as output, such that:
• For each instance $i_1 \in CM_B.L_B$, if $i_1$ obeys $\sigma$, then $getDataGraph(CM_B, i_1) \subseteq CM_{B_1}$. 
4 MAPPING BETWEEN LOCAL MODEL AND XML SOURCE

This section describes the information that is maintained regarding the mapping between a local schema and the corresponding local XML schemata. This information is used by the decomposition algorithm during the translation of a global query into a local query. The mapping approach applied is known in literature as global-as-view approach [8]. This approach was used also in a previous work of our research group [4], being here extended with additional information required by the decomposition mechanism.

Two types of mappings are applied: absolute and relative mappings. Absolute mappings are used to describe how a concept at the conceptual level is found at the XML level. Absolute mappings are given by absolute XPath expressions, one for each concept and each source. Relative mappings are used to describe how a relationship traversal at the conceptual level is mapped to a navigation between elements at the XML level. Relative mappings are given by relative XPath expressions that navigate from an element to another at the XML level.

Below we present examples of mapping information for the local conceptual models shown at figure 3.2, considering the structure of the XML files shown at figures 4.1 and 4.2.

Table 1 shows absolute mappings. For each non-lexical concept in a given local conceptual model and for each XML source, there will be an XPath expression that retrieves the XML elements which represent that concept. For example, consider $LCM_2$ (Figure 3.2) and the single XML source (courses.xml) that is associated to it (Figure 4.1). $LCM_2$ contains two non-lexical concepts: Course and Teaches. The concept Course at the conceptual level is mapped to the XPath expression

```
<Courses>
  <Course>
    <CourseCode>INF061</CourseCode>
    <Name>Database Foundations</Name>
    <Professor>
      <Period>
        <Semester>2001/6</Semester>
        <ProfessorEmail>prof1@inf.ufrgs.br</ProfessorEmail>
        <ProfessorEmail>prof2@inf.ufrgs.br</ProfessorEmail>
      </Period>
    </Professor>
  </Course>
</Courses>
```

Figure 4.1: Schema for $LCM_2$ XML source
Figure 4.2: Schemas for $LCM_3$ XML sources

Table 4.1: Absolute Mapping Information

<table>
<thead>
<tr>
<th>concept</th>
<th>$LCM$</th>
<th>url</th>
<th>expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course</td>
<td>$LCM_2$</td>
<td>courses.xml</td>
<td>/Courses/Course</td>
</tr>
<tr>
<td>Teaches</td>
<td>$LCM_3$</td>
<td>courses.xml</td>
<td>/Courses/Course/Professor/Period</td>
</tr>
<tr>
<td>Enrollment</td>
<td>$LCM_3$</td>
<td>enroll2005.xml</td>
<td>/students/students/enrolls/enroll</td>
</tr>
<tr>
<td>Enrollment</td>
<td>$LCM_3$</td>
<td>enroll2006.xml</td>
<td>/Enrollments/Enrollment</td>
</tr>
<tr>
<td>Student</td>
<td>$LCM_3$</td>
<td>enroll2005.xml</td>
<td>/students/student</td>
</tr>
<tr>
<td>Student</td>
<td>$LCM_3$</td>
<td>enroll2006.xml</td>
<td>/Enrollments/Enrollment/Students/Student</td>
</tr>
</tbody>
</table>

/Courses/Course at the XML source. In the same way the mapping of the concept $Teaches$ to this source is given by the XPath expression /Courses/Course/Professor/Period.

Table 2 shows examples of relative mappings. For each relationship traversal in a local conceptual model and for each XML source, there is a relative XPath expression that defines how the relationship traversal at the conceptual level is mapped at the XML level. For example, consider the relationship between the concepts $Course$ and $CourseCode$ in the local conceptual model $LCM_2$ (Figure 3.2). The traversal of this relationship from $Course$ to $CourseCode$ is mapped to the relative XPath expression $CourseCode$. This expression specifies that, in order to navigate in this XML source from the element that corresponds to the concept $Course$ to the element that corresponds to the concept $CourseCode$, the relative XPath expression $CourseCode$ must be applied. The relative XPath expression that maps the traversal of the same relationship in the reverse direction (from $CourseCode$ to $Course$) is .. (the XPath ancestor axis operator).

A formal definition of the mapping information is given below:

**Definition 4.1** A mapping information is a tuple $MI = < CM, LMI >$, where $CM$ is a global conceptual model and $LMI$ is a set of local mapping information. A local mapping information is a tuple $LMI = < LCM, AM, RM >$, where:
### Table 4.2: Relative Mapping Information

<table>
<thead>
<tr>
<th>source</th>
<th>destination</th>
<th>LCM</th>
<th>url</th>
<th>expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course</td>
<td>CourseCode</td>
<td>LCM2</td>
<td>courses.xml</td>
<td>CourseCode</td>
</tr>
<tr>
<td>Course</td>
<td>CourseName</td>
<td>LCM2</td>
<td>courses.xml</td>
<td>Name</td>
</tr>
<tr>
<td>CourseCode</td>
<td>Course</td>
<td>LCM2</td>
<td>courses.xml</td>
<td>..</td>
</tr>
<tr>
<td>CourseName</td>
<td>Course</td>
<td>LCM2</td>
<td>courses.xml</td>
<td>..</td>
</tr>
<tr>
<td>Enrollment</td>
<td>Semester</td>
<td>LCM3</td>
<td>enroll2005.xml</td>
<td>semester</td>
</tr>
<tr>
<td>Enrollment</td>
<td>CourseCode</td>
<td>LCM3</td>
<td>enroll2005.xml</td>
<td>course</td>
</tr>
<tr>
<td>Enrollment</td>
<td>Student</td>
<td>LCM3</td>
<td>enroll2005.xml</td>
<td>../</td>
</tr>
<tr>
<td>Semester</td>
<td>Enrollment</td>
<td>LCM3</td>
<td>enroll2005.xml</td>
<td>..</td>
</tr>
<tr>
<td>Student</td>
<td>Enrollment</td>
<td>LCM3</td>
<td>enroll2005.xml</td>
<td>enroll/enrolls</td>
</tr>
<tr>
<td>Student</td>
<td>Number</td>
<td>LCM3</td>
<td>enroll2005.xml</td>
<td>number</td>
</tr>
<tr>
<td>Student</td>
<td>Name</td>
<td>LCM3</td>
<td>enroll2005.xml</td>
<td>name</td>
</tr>
<tr>
<td>Number</td>
<td>Student</td>
<td>LCM3</td>
<td>enroll2005.xml</td>
<td>..</td>
</tr>
<tr>
<td>Name</td>
<td>Student</td>
<td>LCM3</td>
<td>enroll2006.xml</td>
<td>..</td>
</tr>
<tr>
<td>Enrollment</td>
<td>Semester</td>
<td>LCM3</td>
<td>enroll2006.xml</td>
<td>Semester</td>
</tr>
<tr>
<td>Enrollment</td>
<td>CourseCode</td>
<td>LCM3</td>
<td>enroll2006.xml</td>
<td>CourseCode</td>
</tr>
<tr>
<td>Enrollment</td>
<td>Student</td>
<td>LCM3</td>
<td>enroll2006.xml</td>
<td>Students/Student</td>
</tr>
<tr>
<td>Semester</td>
<td>Enrollment</td>
<td>LCM3</td>
<td>enroll2006.xml</td>
<td>..</td>
</tr>
<tr>
<td>CourseCode</td>
<td>Enrollment</td>
<td>LCM3</td>
<td>enroll2006.xml</td>
<td>..</td>
</tr>
<tr>
<td>Student</td>
<td>Enrollment</td>
<td>LCM3</td>
<td>enroll2006.xml</td>
<td>../</td>
</tr>
<tr>
<td>Student</td>
<td>Number</td>
<td>LCM3</td>
<td>enroll2006.xml</td>
<td>Number</td>
</tr>
<tr>
<td>Student</td>
<td>Name</td>
<td>LCM3</td>
<td>enroll2006.xml</td>
<td>Name</td>
</tr>
<tr>
<td>Number</td>
<td>Student</td>
<td>LCM3</td>
<td>enroll2006.xml</td>
<td>..</td>
</tr>
<tr>
<td>Name</td>
<td>Student</td>
<td>LCM3</td>
<td>enroll2006.xml</td>
<td>..</td>
</tr>
</tbody>
</table>

- **LCM** is a local conceptual model of MI.CM.
- **AM** is a set of absolute mappings. An absolute mapping is a tuple $AM = <c, url, expression>$, where $c \in \text{LCM.CM'}.\text{NL}$, $url$ is the url of the source and expression is the XPath expression which map the concept $c$ into the source url.
- **RM** is a set of relative mappings. A relative mapping is a tuple $RM = <r, url, expression>$, where $r \in \text{LCM.CM'}.\text{R}$, $url$ is the url of the source and expression is the XPath expression which map the concept $c$ into the source url.
In this section, we describe how a query expressed at the conceptual level through a CXPath expression (Section 2.3) is rewritten into an XQuery expression at the XML level. More specifically, the decomposition algorithm first translates the CXPath expression into several XPath expressions, one for each XML source, and then groups these XPath subqueries into a single XQuery expression.

The main steps of the decomposition algorithm are the following:

1. Parse the CXPath query
2. Handle horizontal fragments
3. Navigate inside fragments
4. Join split fragments
5. Implement the selection predicates

The decomposition algorithm is explained here by discussing the results of each step of the algorithm on an example CXPath query. The overall process is described in the algorithm 1. Further, the version of the algorithm explained here does not handle vertical fragmentation. The necessary changes in the algorithm to include this type of fragmentation are discussed in section 5.7.

In the following subsections we will show how the CXPath expression below (Example 2.1 of Section 2) is translated when it is executed against the global conceptual model described in Figure 3.1 and the local conceptual models described in Figure 3.2, with the mapping information declared in tables 4.1 and 4.2 are considered.

/Course[CourseCode="INF001"]/Enrollment[Semester="2006/1"]
/Student/Name

5.1 Parse Query

The first decomposition step is to parse the query and generate a tuple for each concept in the CXPath query. CXPath subqueries that appear in the selection predicates are handled by recursive calls of this algorithm as explained below. Each tuple contains a tuple identifier (object id), the concept, the local conceptual model
Algorithm 1: decomposeQuery

<table>
<thead>
<tr>
<th>Algorithm 1: decomposeQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td>input : A global query CXPath, mapping information MI, the binded concept bindedOid and bindedConcept</td>
</tr>
<tr>
<td>output : A local query XQuery</td>
</tr>
</tbody>
</table>

1 begin
2      parsedQuery ← parseQuery(CXPath, MI, bindedOid, bindedConcept)
3      if bindedConcept ≠ empty then
4      binded = true
5      else
6          binded = false
7      add the result of handleHorizontalFragments(parsedQuery, MI, binded) into canonicalQuery.FOR
8      add the result of navigateInsideFragments(parsedQuery, MI) into canonicalQuery.FOR
9      add the result of joinSplitFragments(parsedQuery, MI) into canonicalQuery.WHERE
10     XQuery ← writeFinalQuery(canonicalQuery)
11 end

<table>
<thead>
<tr>
<th>Table 5.1: Result of parseQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

which the concept belongs to, its sources, and the selection predicates associated to the concept in the XPath expression.

The tuple identifier is used during the generation of the FOR clause of the resulting XQuery expression to name each variable with a $v$ followed by the object id. Table 5.1 contains the result of this step for the example query. The parse query formalization is specified in the definition 5.1.

**Definition 5.1** A parsed query parsedQuery is a set of tuples $PQ = \langle oid, C, LCM, sources, predicate \rangle$, where:

- $oid$ is the object oid of the parsed concept.
- $C$ is a concept, where $C \in MI.CM.NL \cup MI.CM.L$.
- $LCM$ is a local conceptual model, where $LCM \in MI.LMI.LCM$.
- $sources$ is a set of url, where $url \in MI.LMI.AM.url$.
- $predicate$ is a string.

### 5.2 Handle Horizontal Fragments

In this step, the algorithm begins to build the XQuery expression. For each local conceptual model that is referenced by the concepts in the CXPath expression, the binding of an XQuery variable will be constructed. These bindings will be used in the FOR clause of the resulting XQuery expression. In the case of the example, the bindings in Table 5.2 will be constructed.

These variable bindings are constructed as follows. For each local conceptual model, the algorithm takes the first concept in the parsed query (Table 5.1). In the
example, these will be the concepts Course in $LCM_2$ and Enrollment in $LCM_3$. For each concept a binding is constructed. This binding contains a reference to each XML source that is associated to the local model, as well as the absolute mapping expressions (Section 4) for the concept in each source, that specifies how instances of this concept are to be found in the XML source. This process is formalized in algorithm 2.

The first concept of $LCM_2$ found in the example query is Course. $LCM_2$ corresponds to the XML source courses.xml and the absolute mapping expression for Course in courses.xml is /Courses/Course. This leads to the first binding shown in Table 5.2. Local conceptual model $LCM_3$ is represented by two different XML data sources, enroll2005.xml and enroll2006.xml. In this case the binding will correspond to the union of the elements that represent the concept Enrollment in these sources (second line in Table 5.2).

<table>
<thead>
<tr>
<th>var</th>
<th>expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v1</td>
<td>doc(&quot;courses.xml&quot;)/Courses/Course</td>
</tr>
<tr>
<td>$v2</td>
<td>doc(&quot;enroll2005.xml&quot;)/students/student/enrolls/enroll</td>
</tr>
</tbody>
</table>

### Algorithm 2: handleHorizontalFragments

```java
int begin 1
LCM ← empty
foreach PQ in parsedQuery do 2
   if (PQ.LCM ≠ LCM) AND (binded is not true) then 3
      expression ← empty
      foreach source in PQ.sources do 4
         increment counter
         if counter > 1 then 5
            path ← findAbsolutePath(PQ.concept, source, MI)
            expression ← expression + 'doc("' + source + ' ")/Path + path
      end 6
      ForClause.var ← "$v" + PQ.oid
      ForClause.expression ← expression
      add ForClause into FOR
   end 7
   LCM ← PQ.LCM
end 8
```

5.3 Navigate Inside Fragments

In the second step we have defined variables that are bound to absolute XPath expressions that will be used to iterate over each fragment, i.e., over each local conceptual model. In this step we will implement the navigation inside each fragment by relative XPath expressions. To handle this problem, we adopted the same approach as in [4].

The algorithm compares each pair of adjacent concepts in the parsed query. If the pair belongs to the same local conceptual model, it means that a relationship between two concepts is being traversed. To implement this relationship traversal at the XML source level, the relative path expression defined by the relative mapping for this
traversal (Section 4) is bound to a variable. Algorithm 3 formalizes this process. For example, consider the relationship traversal from Enrollment to Student in LCM$_3$. This traversal is implemented in source enroll2005.xml by the relative XPath expression ../.., and in source enroll2006.xml by the relative XPath expression Students/Student (see Table 4.2). This leads to the first line in Table 5.3.

This lines defines a binding for a new variable $v3$ that is bound to an expression that is relative to the $v2$ variable. Recall that $v2$ is bound to an absolute XPath expression that retrieves the XML elements that represent the Enrollment concept, source of the relationship traversal being handled. The $v3$ binding constructed in this step specifies the traversal of the relationship from Enrollment to Student at the conceptual level. As this relationship is implemented in two different XML sources, the terms in form base_uri(.) are used to ensure that the relative path is being applied over the correct source.

Analogously a variable $v4$ is bound to the relative XPath expression that specifies the traversal from concept Student to concept Name.

<table>
<thead>
<tr>
<th>var</th>
<th>expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v3$</td>
<td>$v2[base_uri(.)=&quot;enroll2005.xml&quot;]/../..</td>
</tr>
<tr>
<td>$v4$</td>
<td>$v3[base_uri(.)=&quot;enroll2005.xml&quot;]/name</td>
</tr>
</tbody>
</table>

**Algorithm 3: navigateInsideFragments**

```
input : A parsed query parsedQuery and mapping information MI
output : A set of FOR clauses FOR
1 begin
2   counter ← 0
3   foreach PQ in parsedQuery do
4       increment counter
5       expression ← empty
6       baseURI ← empty
7       if counter = 1 then
8           previousPQ ← PQ
9           Next
10      if PQ.LCM = previousPQ.LCM then
11         counterSources ← 0
12        foreach source in PQ.sources do
13            increment counterSources
14            path ← findRelativePath(previousPQ.concept, PQ.concept, source, MI)
15            if Count(PQ.sources) > 1 then
16                baseURI ← "[base_uri(.)=" + source + "]"
17                if counterSources > 1 then
18                    expression ← expression + '|' + path
19                expression ← expression + '(' + previousPQ.oid + baseURI + ')' + path + ')'
20            ForClause.var ← "$v" + PQ.oid
21            ForClause.expression ← expression
22            add ForClause into FOR
23       previousPQ ← PQ
24 end
```
Table 5.4: Result of joinSplitFragments - WHERE clauses

<table>
<thead>
<tr>
<th>expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v2[base_uri(.)=&quot;enroll2005.xml&quot;]/course</td>
</tr>
</tbody>
</table>

5.4 Join Split Fragments

A CXPath query may access concepts from different local models, which means that several split fragments (local models) need to be accessed and joined to answer the query. For each local model, a variable bound to an absolute XPath expression that retrieves elements in the fragment was constructed in the second step. In this step, the criteria to join the elements from different sources is constructed. This criteria will take part of the WHERE clause in the resulting XQuery expression. The join criteria will be constructed using the identifiers and identifier references that were defined during the the split operation that generated the local models from the global model (Section 2.1)

In the parsed query table (Table 5.1), for each adjacent pair of concepts that belong to different local models, a join must be defined. For each such pair of concepts, the algorithm finds the identifiers and the identifier references that are used to navigate from one fragment to the other. In the example (Table 5.1) there are is a single pair of concepts that corresponds to the navigation from one fragment to the other, namely the traversal from concept Course to concept Enrollment. This relationship is implemented by identifier reference Enrollment.CourseCode that references the identifier Course.CourseCode.

Next, for each XML source, the algorithm takes the relative path expressions that implement the traversal from Course to the identifier CourseCode and from Enrollment to the identifier reference CourseCode. Finally an equality predicate among this two relative XPath expressions is included in the WHERE clause as shown in Table 5.4.

The algorithm 4 formalizes this process.

5.5 Handle Selection Predicates

This step of the algorithm is in charge of handling the selection predicates that may appear in the CXPath query. This selection predicates will be rewritten into terms in the WHERE clause of the resulting XQuery expression. The algorithm supports a simplified form of the selection predicate, which has the following structure:

CXPathLeft comp [CXPathRight OR constant]

where CXPathLeft and CXPathRight are CXPath subqueries, which may be relative or absolute expressions, comp ∈ {=, ≠, <, >, ≤, ≥} is a comparison operator, and constant is a string.

The first step of the handle selection predicates algorithm consists into parsing the predicates into the global query. In the example, the result will be that shown in table 5.5.

The CXPath subqueries that may appear in the selection predicate are translated by a recursive execution of the decomposition algorithm. For example, the selection
Algorithm 4: joinSplitFragments

\begin{algorithm}
\begin{algorithmic}[1]
   \State \textbf{input}: A parsed query \texttt{parsedQuery} and mapping information \texttt{MI}
   \State \textbf{output}: A set of WHERE clauses \texttt{WHERE}
   \Procedure{joinSplitFragments}{
   \State \texttt{counter} $\leftarrow 0$
   \ForAll{\texttt{PQ} in \texttt{parsedQuery}}
   \State increment \texttt{counter}
   \State \texttt{expressionLeft}, \texttt{expressionRight}, \texttt{baseURI} $\leftarrow$ empty
   \If{$\texttt{counter} = 1$}
   \State \texttt{previousPQ} $\leftarrow$ \texttt{PQ}
   \EndIf
   \If{$\texttt{PQ}.\texttt{LCM} \neq \texttt{previousPQ}.\texttt{LCM}$}
   \If{$\texttt{getIDREF}(\texttt{previousPQ}.\texttt{concept}, \texttt{PQ}.\texttt{concept}, \texttt{MI}) = \text{empty}$}
   \State \texttt{IDREFS} $\leftarrow$ \texttt{getIDREF}(\texttt{previousPQ}.\texttt{concept}, \texttt{PQ}.\texttt{concept}, \texttt{MI})
   \State \texttt{idrefConcept} $\leftarrow$ \texttt{PQ}.\texttt{concept}
   \State \texttt{idrefSources} $\leftarrow$ \texttt{PQ}.\texttt{sources}
   \State \texttt{idConcept} $\leftarrow$ \texttt{previousPQ}.\texttt{concept}
   \State \texttt{idSources} $\leftarrow$ \texttt{previousPQ}.\texttt{sources}
   \Else
   \State \texttt{IDREFS} $\leftarrow$ \texttt{getIDREF}(\texttt{previousPQ}.\texttt{concept}, \texttt{PQ}.\texttt{concept}, \texttt{MI})
   \State \texttt{idrefConcept} $\leftarrow$ \texttt{previousPQ}.\texttt{concept}
   \State \texttt{idrefSources} $\leftarrow$ \texttt{previousPQ}.\texttt{sources}
   \State \texttt{idConcept} $\leftarrow$ \texttt{PQ}.\texttt{concept}
   \State \texttt{idSources} $\leftarrow$ \texttt{PQ}.\texttt{sources}
   \EndIf
   \EndIf
   \State \texttt{baseURI} $\leftarrow$ empty
   \ForAll{\texttt{REF} in \texttt{IDREFS}.\texttt{REF}}
   \State \texttt{counterSources} $\leftarrow 0$
   \ForAll{\texttt{source} in \texttt{idrefSources}}
   \State increment \texttt{counterSources}
   \State \texttt{path} $\leftarrow$ \texttt{findRelativePath}(	exttt{idrefConcept}, \texttt{REF}.\texttt{ref}.\texttt{c2}, \texttt{source}, \texttt{MI})
   \If{$\texttt{Count}($\texttt{idrefSources}$) > 1$}
   \State $\texttt{baseURI} \leftarrow \left[\text{base_uri(.)}="+source + \\
   \"\]$'
   \State \texttt{expressionLeft} $\leftarrow$ \texttt{expressionLeft} + '$\mid$
   \State \texttt{expressionLeft} $\leftarrow$ \texttt{expressionLeft} + ($\texttt{v} + \texttt{idrefConceptOid} + \texttt{baseURI} + \\
   \texttt{"/" + \texttt{path} + \\
   \texttt{"/"})'$
   \State \texttt{baseURI} $\leftarrow$ empty
   \State \texttt{counterSources} $\leftarrow 0$
   \EndIf
   \EndFor
   \State \texttt{baseURI} $\leftarrow$ empty
   \ForAll{\texttt{source} in \texttt{idSources}}
   \State increment \texttt{counterSources}
   \State \texttt{path} $\leftarrow$ \texttt{findRelativePath}(	exttt{idConcept}, \texttt{REF}.\texttt{id}.\texttt{c2}, \texttt{source}, \texttt{MI})
   \If{$\texttt{Count}($\texttt{idSources}$) > 1$}
   \State $\texttt{baseURI} \leftarrow \left[\text{base_uri(.)}="+source + \\
   \"\]$'
   \State \texttt{expressionRight} $\leftarrow$ \texttt{expressionRight} + '$\mid$
   \State \texttt{expressionRight} $\leftarrow$ \texttt{expressionRight} + ($\texttt{v} + \texttt{idConceptOid} + \texttt{baseURI} + \\
   \texttt{"/" + \texttt{path} + \\
   \texttt{"/"})'$
   \EndIf
   \EndFor
   \EndFor
   \State \texttt{WhereClause} $\leftarrow$ \texttt{expressionLeft} + $\text{"=\"} + \texttt{expressionRight}$
   \State add \texttt{WhereClause} into \texttt{WHERE}
   \EndIf
   \EndFor
   \State \texttt{previousPQ} $\leftarrow$ \texttt{PQ}
   \EndProcedure
\end{algorithmic}
\end{algorithm}

Table 5.5: Result of \texttt{parsedSelectionPredicates}
\begin{tabular}{|c|c|c|c|}
\hline
\texttt{bindedOid} & \texttt{bindedConcept} & \texttt{predicate} & \texttt{CXPathLeft} & \texttt{CXPathRight} \\
\hline
1 & Course & CourseCode="INF001" & CourseCode & \\
\hline
2 & Enrollment & Semester="2006/1" & Semester & \\
\hline
\end{tabular}
Figure 5.1: XQuery generated by the decomposition algorithm to example 2.1

The XQuery expression for $v1-1 ... is obtained by the recursive execution of the decomposition algorithm on the relative CXPath expression CourseCode that appear in the CXPath selection predicate. When the CXPath expressions inside a selection predicate are relative, the context, i.e. the tuple identifier and the concept, must be passed as parameter to the decomposition algorithm. The details regarding the recursive call are detailed in the next subsection. The algorithm which handles the selection predicates parsing and the recursive calls is formalized in the algorithm 5.

The final XQuery for the example 2.1 is shown at figure 5.1.

**Definition 5.2** A parsed selection predicate parsedSelectionPredicates is a set of tuples \( \text{PSP} = \langle \text{bindedOid}, \text{bindedConcept}, \text{selectionPredicate}, \text{CXPathLeft}, \text{CXPathRight} \rangle \), where:

- \( \text{bindedOid} \) is the object oid of the binded concept.
- \( \text{bindedConcept} \) is a concept, where \( \text{bindedConcept} \in \text{MI.CM.NL} \cup \text{MI.CM.L} \).
- \( \text{selectionPredicate} \) is a string.
- \( \text{CXPathLeft} \) is a string.
- \( \text{CXPathRight} \) is a string.

### 5.6 Recursive Call

The recursive executions of the algorithm are similar with the first iteration. The steps that behave in a different way are the parse query and handle horizontal
Algorithm 5: handleSelectionPredicates

input : A parsed query parsedQuery, a canonical query canonicalQuery and mapping information MI
output : A canonical query canonicalQuery

begin
1 { parsedSelectionPredicates ← parseSelectionPredicates(parsedQuery) } parsedPredicate in parsedSelectionPredicates do
2 if NOT isAbsolute(parsedPredicate.CXPathLeft) then
3   XQueryLeft ← decomposeQuery(CXPathLeft, MI, parsedPredicate.bindedOid, parsedPredicate.bindedConcept)
4 else
5   XQueryLeft ← decomposeQuery(CXPathLeft, MI, empty, empty)
6 if parsedPredicate.CXPathRight ≠ empty then
7   if NOT isAbsolute(parsedPredicate.CXPathRight) then
8     XQueryRight ← decomposeQuery(CXPathRight, MI, parsedPredicate.bindedOid, parsedPredicate.bindedConcept)
9   else
10    XQueryRight ← decomposeQuery(CXPathRight, MI, empty, empty)
11 selectionPredicate ← parsedPredicate.predicate
12 selectionPredicate.replace(CXPathLeft, '(' + XQueryLeft + ')')
13 selectionPredicate.replace(CXPathRight, '(' + XQueryRight + ')')
14 add selectionPredicate into canonicalQuery.WHERE
end

Table 5.6: Result of parseQuery in the recursive call

<table>
<thead>
<tr>
<th>oid</th>
<th>concept</th>
<th>LCM</th>
<th>sources</th>
<th>predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Course</td>
<td>LC_M2</td>
<td>courses.xml</td>
<td></td>
</tr>
<tr>
<td>1-1</td>
<td>CourseCode</td>
<td>LC_M2</td>
<td>courses.xml</td>
<td></td>
</tr>
</tbody>
</table>

fragments. The other steps have the same behavior for all levels of recursion. In this subsection this differences will be explained through a detailed execution of the recursive call done for decomposing the relative expression CourseCode into the selection predicate CourseCode="INF001" of the example query.

In this example, the expression CourseCode is relative to the concept which the selection predicate is applied to, that is the Course concept. This concept and the respective object id should be informed to the decomposition algorithm during the recursive call.

The parse query step have a different behavior of the first iteration if the query being decomposed is relative. It includes the binded concepts into the parsed query, and keeps, for this concept only, the object id used in the previous iteration. The object id of the remaining concepts will be a concatenation of the binded oid and a new object id. In the example, the parsed query produced is shown in table 5.6.

The next step of the algorithm is handling the selection predicates. The behavior of this step, when handling an absolute expression, is generating a binding for each local conceptual model in the query, as explained in section 5.2. In a recursive call for decomposing a relative expression, the first local conceptual model in the parsed query is ignored. This is done because the algorithm will use the binding generated in the previous recursive call, instead of declaring a new one. In the example, no binding is generated in this step.

The navigate inside fragments step have the same behavior in all recursive iterations. In the example, it will generate the binding shown in table 5.7. Consider that this is a relative path expression applied over a binding declared in the previous iteration of the algorithm.

The remaining steps will produce no results for the example query. The final
Table 5.7: Result of \texttt{navigateInsideFragments} in the recursive call - FOR clauses

<table>
<thead>
<tr>
<th>var</th>
<th>expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v1-1$</td>
<td>$v1/CourseCode$</td>
</tr>
</tbody>
</table>

Table 5.8: Result of \texttt{parseQuery} - vertically fragmented models

<table>
<thead>
<tr>
<th>oid</th>
<th>concept</th>
<th>$LCM_i$</th>
<th>sources</th>
<th>predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Teaches</td>
<td>$LCM_2$</td>
<td>courses.xml</td>
<td>Period=&quot;2001&quot;</td>
</tr>
<tr>
<td>2</td>
<td>Professor</td>
<td>$LCM_{V1}$</td>
<td>prof1.xml</td>
<td>Department=&quot;Databases&quot;</td>
</tr>
<tr>
<td>3</td>
<td>Professor</td>
<td>$LCM_{V2}$</td>
<td>prof2.xml</td>
<td>Department=&quot;Databases&quot;</td>
</tr>
<tr>
<td>4</td>
<td>Name</td>
<td>$LCM_{V2}$</td>
<td>prof2.xml</td>
<td></td>
</tr>
</tbody>
</table>

query result will be the following:

forall $v1-1$ in $v1/CourseCode$ return $v1-1$

5.7 Handle Vertical Fragmentation

The inclusion of a handling vertical fragments step into the decomposition algorithm implies some changes into all other steps. In this subsection, we briefly describe this changes, and also make a short description of the handle vertical fragmentation algorithm itself. This will be explained through the decomposition of the following CXPath query (Example 2.2 of Section 2), executed against the conceptual model in figure 3.1 and the local conceptual models $LCM_2$, $LCM_{V1}$ and $LCM_{V2}$ described in figures 3.2 and 3.3, respectively.

/Teaches[Period="2001"]/Professor[Department="Databases"]/Name

The first change that need to be made is in the parse query algorithm. To handle the vertical fragments, this algorithm should include in the parsed query one tuple for each local conceptual model and for each concept of the query being decomposed, instead of only one tuple for each concept. With this change, each vertically fragmented concept will have several tuples. In the example, the concept Professor, which is represented in the local models $LCM_{V1}$ and $LCM_{V2}$, have two tuples in the parsed query, as shown in table 5.8.

The handle horizontal fragments algorithm will have the same behavior, except that a same concept may have more than one binding, because of the change in the parse query algorithm. The FOR bindings generated for this example is shown in table 5.9.

The navigate inside fragments algorithm will have a different behavior. Instead of comparing each pair of adjacent concepts into the parsed query, the algorithm should compare all the tuples for a given concept with all the tuples of the adjacent concept. The tuples for a same concept should not be compared between each other. In the example, the tuple 1 should be compared with the tuples 2 and 3 (navigation from Teaches to Professor), and the tuples 2 and 3 should be compared with to tuple

Table 5.9: Result of \texttt{handleHorizontalFragments} - vertically fragmented models

<table>
<thead>
<tr>
<th>var</th>
<th>expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v1$</td>
<td>doc(&quot;courses.xml&quot;)/Courses/Course</td>
</tr>
<tr>
<td>$v2$</td>
<td>doc(&quot;prof1.xml&quot;)/Professor</td>
</tr>
<tr>
<td>$v3$</td>
<td>doc(&quot;prof2.xml&quot;)/professors/professor</td>
</tr>
</tbody>
</table>
Table 5.10: Result of \textit{navigateInsideFragments} - vertically fragmented models

<table>
<thead>
<tr>
<th>var</th>
<th>expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v4$</td>
<td>$v3$_name</td>
</tr>
</tbody>
</table>

Table 5.11: Result of \textit{joinSplitFragments} - vertically fragmented models

expression

\[(v1/ProfessorEmail = v2/Email\)]

4 (navigation from \textit{Professor} to \textit{Name}). The tuples that belong to the same local model are the ones which have a relationship being traversed. In the example, the navigation to the concept \textit{Name} will be implemented with a relative path expression over the binding $v3$, that is the binding which belongs to the same local conceptual model as the tuple 4, that is the $LCM_{V2}$. The generated FOR clause is shown in table 5.10.

The join split fragments algorithm will have a similar change in the behavior. As in the navigate inside fragments algorithm, it will also compare all the tuples of a given concept with all the tuples of the adjacent concept. If all the local conceptual models are different, then a join need to be implemented. In the example query, this is implemented between the tuples 1 and 2. The result of this step is shown in table 5.11.

The handle selection predicates algorithm should have a change in it’s behavior. When the decomposition algorithm is called recursively for a relative expression, the context (binded oid and the binded concept) should be passed as parameter, as explained in section 5.5. If the binded concept is vertically fragmented, it’s necessary to figure out which binded oid need to be associated with that navigation. This is done parsing the relative CXPath expression and checking if the first concept of that expression is represented into the same local conceptual model that one of the binded oids of the binded concept. If this is true, the binded oid with the same local model is passed as parameter. If not, any binding can be used, so for simplicity the algorithm will take the first one. In the example, the expression \textit{Department}, relative to the concept \textit{Professor}, is represented in the local conceptual model $LCM_{V1}$. The binding 2 of the parsed query represents the concept \textit{Professor} into the local conceptual model $LCM_{V1}$, so this binding is passed as parameter to the decomposition algorithm. The result of the decomposition is shown bellow.

\[\text{for } v2-1 \text{ in } v2/Department \text{ return } v2-1\]

The handle vertical fragments algorithm should be included as one of the steps of the decomposition algorithm. This step have a similar behavior to the join split fragments algorithm. It’s responsible by the generation of the criteria to join the different vertical fragments. As in join split fragments algorithm, this criteria will take part of the WHERE clause, using the \textit{identifier} of the fragmented concept.

The behavior of this algorithm is the following. For each adjacent pair of tuples in the parsed query with the same concept, a join must be defined. The algorithm get the identifiers of this concept and, for each XML source, take the relative path expression that implements the traversal from the fragmented concept to each one of the identifiers. Then, an equality predicate between the two bindings is included into the WHERE clause. In the example, there are one pair of tuples in the parsed query with the same concept, that is the tuples 2 and 3 for the concept \textit{Professor}. 

for $v2-1$ in $v2/Department$ return $v2-1$
Table 5.12: Result of handleVerticalFragments - WHERE clauses

<table>
<thead>
<tr>
<th>expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(v2/Email = v3/email)</td>
</tr>
</tbody>
</table>

The identifier for the concept Professor is the concept ProfessorEmail, so the next step of the algorithm is taking the relative path expressions between this concepts, for each XML source and for each binding. In the source prof1.xml of binding 2, this relationship is implemented with the relative path expression Email, while in the source prof2.xml of binding 3, this relationship is implemented with the expression email. The WHERE clause generated is shown in table 5.12.
6 CONCLUSION AND FUTURE WORK

This paper presents an approach to handle the problem of querying integrated XML documents. This approach is composed by two parts. In the first part, a set of fragmentation operators for XML documents that belong to a specific domain described by a conceptual schema are proposed. In the second part, a query decomposition mechanism is presented that, given mapping information between the fragmented schemas and the XML documents, translates a CXPath query expression at the global level into an XQuery expression at the XML level.

One of the main contributions of our approach is to clearly separate the problem of fragmentation of an global schema into several local schemata from the problem of abstracting from different XML representations of the same conceptual information. This separation is achieved by the introduction of a conceptual layer at the local source level and at the global (mediated) level as well. The fragmentation operators are defined at this conceptual level.

However, some issues related to the query decomposition mechanism are still open. The vertical fragmentation is not handled by the algorithm, and the problem of instances integration, i.e. how to identify and integrate the same instance that is represented into different sources must still be investigated.

Another issue that is not covered by this work is that of query optimization. The queries generated by this algorithm are not the most efficient ones, and most of the times the performance of the query execution can be improved simply rewriting it. A further investigation can decide if the better decision for handling the optimization is improving the generated XQuery or using a XML algebra, like TAX [10], after the decomposition is concluded.
REFERENCES


