Rewriting Queries over XML Views

by

Theophanis Tsandilas

A thesis submitted in conformity with the requirements
for the degree of Master
Graduate Department of Computer Science
University of Toronto

Abstract

Rewriting Queries over XML Views

Theophanis Tsandilas

Master

Graduate Department of Computer Science

University of Toronto

2001

XML is widely used as the standard format of data representation and data exchange on the Web, and database research has recently focused on exploiting its experience in traditional databases to handle similar issues in the context of the new data format. This thesis deals with the problem of query composition in the context of XML-QL, which is one of the most popular XML query languages. More specifically, we study how a user query defined over one or more views can be rewritten, so that the rewritten query only refers to source data. This problem is extensively studied in the context of relational and object-oriented databases, but several issues are not yet examined in XML query languages.

The thesis presents a simple rewriting algorithm which is based on a matching procedure between properties of the user query and properties of the view query. The purpose of this procedure is to derive the bindings of the user-query variables without materializing the view. The underlying data model is object-based and unordered. Special issues that we study include multiple view references, equality conditions between variables in the user query, and element grouping in the view.

To my parents

Panagiotis and Konstantina

for their love and continuous support.

Contents

Chapter 1

Introduction

1.1 Preliminaries

The notion of views is essential in traditional database systems. They have been studied extensively in the context of the relational and object database models, but the relevant research is still immature in the context of XML [Con98].

The recent emergence of XML as the standard exchange format between applications on the Internet has important implications for the database community. Classical data models are not adequate to describe XML data. The introduction of a data model also requires the development of new storage mechanisms and new query languages implying a series of new issues such as indexing, query optimization, and database design.

In this new world of XML, the importance of views is even more crucial than in standard database applications. XML is only a syntax and cannot uniquely describe a certain portion of data. Different authors may use totally different XML structures or names to describe the same data. XML views will play a significant role in translating information between sources and applications that use different schemata and ontologies. They will also provide the means to integrate multiple heterogeneous sources. Having in mind that large volumes of data are stored using traditional data models, e.g., relational,

views may also be involved in migrating data from these models to XML.

Issues that have been already studied include XML data models and query languages [SLR98, FSW+99], XML algebras [BT99, BMR99, FSW00, BFS00], and some preliminary work on query optimization [MW99]. The most popular XML query languages that have emerged are: XML-QL [DFF+98, DFF+99], Lorel [GMW99], XQL [RLS98], YATL [CDSS98], UnQL [BFS00], XQuery [CCF+01], and more recently Quilt [Don00]. The language that is used in this thesis is XML-QL. Existing work on XML views involves issues like view definition [BLP+98, LPVV99], maintenance of materialized views [LD00], active views [MSA+99, Abi99], and XML views over relational data [MF00, Bar99].

1.2 Motivation

This thesis studies the problem of *query rewriting* in the context of XML views. More specifically, given an XML query (we call it a *user query*) that is defined over one or more *virtual* (non-materialized) XML views, we find an equivalent query (we call it a *rewritten query*) that is defined over the source data. The term "query rewriting" also refers to the reverse problem: given a query defined over a database and a set of views over the same database, find equivalent rewritten queries defined over this set of views. *Query composition* is another term that is used to describe the problem that we study, since the user query is composed with the view query in order to produce a single query.

Query rewriting is just one of the methods of evaluating queries over views. Other ways to process queries over a virtual view are: (i) materialize the view on demand; (ii) rewrite the view so only the portion of the view that is relevant to the query is materialized; and (iii) translate the query and the view into expressions of a compositional algebra and apply the composition and any possible optimizations at the algebraic level.

The second method is clearly more efficient than the first one. The query rewriting approach is even more efficient, since view rewriting requires the partial materialization

of the view and the execution of the original query over this view. However, we should mention that query rewriting can lead to huge rewritten queries whose optimization may be difficult or even infeasible [AGM+97].

The third method is similar to the query rewriting approach, but composition is based on algebraic transformations rather than transformations at the level of the query language. Mostly due to the lack of a group-by operator in the relational algebra to express grouping, early composition and unnesting techniques were based on transformations over SQL queries instead of applying algebraic transformations. The correctness and completeness of such transformations can be more clearly and strictly proved using a formal algebra[1]. Later approaches used algebraic techniques to perform query composition and unnesting optimizations, mainly in the world of object-oriented databases [CM93, Feg98]. However, due to limitations in existing optimizers, query rewriting algorithms can be more efficient than algebraic transformations that are based on these optimizers. Having in mind that query optimization is still at its infancy in the context of XML and that optimizers for XML query languages are "under construction", the development of query rewriting techniques is significant.

Example 1.1 *Consider the view that is expressed by the following XML-QL query:*

```
V: WHERE
     <Library>
        <book>
           <title>$t</>
           <author>$a</>
           <publisher>Wise Reader</>
        </>
     </> IN ``library.xml''
   CONSTRUCT
       <author>
          <name>$a</>
          <book>$t</>
       </>
```

The above statement creates the view of the authors and their books that are published by

[1]Transformations over SQL resulted in a number of bugs which were finally corrected [GW87].

"Wise Reader". The following query asks for the book titles that are written by "Marcel Proust":

```
Q: WHERE
     <view>
        <author>
           <name>Marcel Proust</>
           <book>$t</>
        </>
     </> IN V
   CONSTRUCT
       <title>$t</>
```

According to the full materialization approach, we have to compute the view's result and then apply the query over this result. The drawback of this approach is that the query is only interested in the books of a specific author, so the computation of the the whole view query is redundant.

The view rewriting approach suggests the transformation of the view and the insertion of the condition $a = "Marcel Proust" *in the view definition (we can also replace the variable $a by the string "Marcel Proust"). The drawback mentioned above is eliminated, but we still need to evaluate two separate queries.*

Finally, the query rewriting approach suggests the transformation of the query as follows:

```
Q: WHERE
     <Library>
        <book>
           <title>$t</>
           <author>Marcel Proust</>
           <publisher>Addison-Wesley</>
        </>
     </> IN ``library.xml''
   CONSTRUCT
       <title>$t</>
```

The query is directly applied on the source data, so the desired result is produced by a single query.

Query rewriting becomes more crucial when there are many levels of views, i.e., views defined over other views. In this case, instead of evaluating a potential big number of queries returning redundant portions of data, we compute a single query that produces only the desired data.

1.3 Related work and scope of the thesis

Rewriting queries defined over views can be seen as a special case of the general problem of query unnesting, which is extensively studied in the context of relational [Kim81, GW87] and object-oriented databases [CM93, Feg98].

In the relational world, query composition is simply performed by merging the FROM and WHERE clauses of the corresponding queries, as long as views are defined as simple SPJ queries. However, when a view is defined as *SELECT DISTINCT*, transformations to move or to pull up *DISTINCT* should preserve the number of duplicates correctly [PHH92]. The situation is also complex when aggregates are present in the view [Mur92]. Aggregates are not examined by this thesis.

The problem is similar in the world of object-oriented databases. Queries defined over views correspond to the case of nesting in the *FROM* clause, when no predicate dependency exists between the inner and the outer block of a nested query. Such nested expressions can be transformed into simple join expressions [CM93].

There are several issues that make the problem different in the context of XML and XML query languages.

1. The schema of XML data is not always known. As a result of this, the schema of the view is only partially known.

2. More than one way of matching the user-query variables with the view variables may be possible, since an element name can appear multiple times even under the same parent element.

3. XML query languages allow the use of tag variables and regular path expressions which make the problem of matching between the variables of the view and the query harder.

4. Skolem functions are used to group data.

5. In the XML world, order is often important. The existence of index operators makes the problem even harder. This thesis does not address the order problem.

Query composition in the context of semistructured data is first addressed in MSL [PAGM96], a *datalog-like* query language. A similar approach is presented by Papakonstantinou and Vassalos [PV99a] for TSL, which is also a datalog-based query language for semistructured data. The unification algorithms that the above frameworks propose are analogous to our matching algorithm. The problem is also examined by Buneman et al. [BFS00] in the context of UnQL [BDHS96], a query language and algebra based on structural recursion. The query composition is performed on the algebraic level. However, the underlying data model is *value-based* as opposed to the *object-based* model that we study. In the XML-QL data model XML data are represented as labeled graphs, whereas in the UnQL data model, XML data are represented as sets of label/value pairs. The composition algorithm presented in the SilkRoute framework [MF00] is the closest to our approach. Views are expressed as RXL (Relational to XML Transformation Language) queries, which inherit the SELECT and FROM clauses from SQL and the CONSTRUCT clause from XML-QL. In other words, RXL queries transform relational data to XML data. User queries are expressed as XML-QL queries. We can enumerate several differences between our approach and SilkRoute.

1. Object identifiers are not always explicitly defined in the constructor of a view. In several cases, these identifiers should be present in the constructor of the rewritten query in order to ensure that the correct object-elements are produced. The way we derive them is different in XML-QL than in RXL queries.

2. A variable in the CONSTRUCT clause of an RXL view can only be mapped to a variable appearing in the WHERE clause of the XML-QL query, since it can only represent atomic values (like variables in SQL queries). In our case, a variable in the constructor of the view can also be mapped to a whole pattern subexpression, since it can represent complex non-atomic values. This difference is reflected in the rewriting algorithm.

3. We investigate how equality conditions between variables in the user query are handled.

4. We present a different approach to handle grouping by Skolem functions in the view, which is based on splitting the pattern expressions of the user query. We also indicate that a grouping performed by a Skolem function should sometimes be expressed by means of a query in order to perform a correct variable substitution during the rewriting process.

1.4 Organization of the thesis

The thesis is organized as follows.

- In Chapter 2, we describe XML-QL and introduce the terms that are used in the next chapters.

- In Chapter 3, we present our query composition approach, when no explicitly defined object identifiers exist in the CONSTRUCT clause of the view.

- In Chapter 4, we introduce explicitly defined object identifiers in the user query and suggest a solution to the problem.

- Finally, in Chapter 5, we give a conclusion and present the main issues for future work.

Chapter 2

Terminology

2.1 XML-QL

XML-QL is a declarative XML query language that can extract data from existing XML documents and construct new XML documents. An XML-QL query consists of three parts: (i) a *pattern part*, which matches elements in the input document and binds variables; (ii) a *filter* part, which provides conditions for the bound variables; and (iii) a *constructor* part, which specifies the result in terms of the bound variables. The pattern and the filter part can include multiple pattern and filter expressions, respectively. Patterns and filters appear in the WHERE clause, while the constructor appears in the CONSTRUCT clause of the query.

Example 2.1 *The following query returns all the titles of books written by "Dostoyevski", published after 1990:*

```
Q: WHERE
     <Library>                        //pattern
        <book published = $year>
           <title>$t</>
           <author>Dostoyevski</>
        </>
     </> IN ``library.xml'',
     $year > 1990                     //filter
   CONSTRUCT
```

```
<title>$t</>                    //constructor
```

Patterns and constructors have the structure of normal XML data, but they can also contain variables. Patterns can also include *regular path expressions*, which can be used to specify element paths of arbitrary depth and tag variables, which can be considered as a way of querying the schema of data.

Example 2.2 *The following query returns all the authors and their writings. The schema of the data is not known. We do not know what kind of writings exist, e.g., whether $w is bound to a book or an article element, and in what depth under Library they appear. By means of regular path expressions and tag variables, we pose a query that does not require the knowledge of the exact schema.*

```
Q: WHERE
     <Library.*>
        <$w>
           <title>$t</>
           <author|writer>$a</>
        </>
     </> IN ``library.xml''
   CONSTRUCT
       <author ID=AuthorID($a)>
           <name>$a</>
           <title>$t</>
       </>
```

In the above example, the results are grouped by the name of the authors. This is achieved by using the *Skolem function AuthorID*, which generates a new identifier for each distinct value of a. According to the data model that we use and describe in the next paragraph, duplicates are eliminated. Therefore, only a single *name* element will appear under each produced *author* element. Skolem functions are powerful and the rewriting algorithm is based on their use.

2.1.1 Data Model

XML-QL supports two different data models: an unordered and an ordered data model.

The *unordered* data model considers that an XML document is a graph, in which each node is represented by a unique *object identifier* (OID). A graph has a unique *root* node and is labeled as follows: graph edges are labeled with element tags, nodes are labeled with sets of attribute-value pairs, and leaves are labeled with string values. Graph edges correspond to element tags, while nodes correspond to element contents. Since every element is denoted by a tag which contains a single label, it may not be clear how a certain node can be pointed by two different edges. An element can refer to other elements by means of IDREF attributes. The value of an IDREF attribute is an OID. XML-QL blurs the distinction between IDREF attributes and elements, and therefore, labeled edges represent both element tags and IDREF attribute names. Since a certain element can be referenced by several other elements, a graph node can be pointed by more than one edge. Attributes are associated to nodes. Two nodes can be connected by several edges, but with the following restriction: a node cannot have two outgoing edges with the same labels and the same values, i.e., between any two nodes there can be at most one edge with a given label, and a node cannot have two leaf children with the same label and the same string value. This condition means that XML documents denote sets.

In the *ordered* data model, the corresponding document graph contains a totally ordered set of nodes. The order of the nodes corresponds to the natural order of elements in the document. Given a total order on nodes, we can enforce a local order on the outgoing edges of each node. In the ordered model, the above restriction is not retained, so arbitrarily many edges with the same source, same edge label, and same destination value can appear. According to this model, duplicates are not eliminated and XML documents denote bags.

In our study, the underlying model is unordered, so rewritten queries do not necessarily preserve the right order in the output.

2.1.2 Syntax

As XML-QL is still evolving, its syntax varies in different publications [DFF+98, Fer99]. In Appendix A, we present the XML-QL syntax that we consider. XML-QL queries can also be expressed as functions. For simplicity, we consider only functions that have no arguments:

```
function GreekCities(){
    WHERE
      <city>
         <name> $c </>
         <population> $p </>
         <country> Greece </>
      </> IN ``cities.xml''
    CONSTRUCT
       <city>
          <name> $c </>
          <inhabitants> $p </>
       </>
}
```

2.1.3 Semantics

We continue with our XML-QL semantics definition. Consider an XML-QL query *WHERE* P *CONSTRUCT* C and the set of variables $x_1, ..., x_k$ that are bounded by one or more conditions that appear in P. Let G be a graph that corresponds to an XML document. We construct a table $R(x_1, ..., x_k)$ with one column for each variable. Each row corresponds to a binding of the variables that satisfies all conditions in P. We should mention that a certain row is produced when each pattern expression in P matches a certain XML subgraph of the source data. More information about how pattern expressions match source XML data can be found in the XML-QL proposal [DFF+98]. Two different rows always correspond to two different matched XML subgraphs for at least one pattern expression of P.

In each row, a variable can be bound either to an OID that corresponds to internal non-leaf node in the graph, to a string value (leaf node, attribute value or tag label).

Let $C(x_1, ..., x_k)$ be the template contained in the CONSTRUCT clause depending on a subset of the variables $x_1, ..., x_k$. If $x_1^i, ..., x_k^i$ are the bindings in row i of the table R, where $i = 1..n$, then, for that row, we construct the XML fragment $C_i := C(x_1^i, ..., x_k^i)$. Each C_i is called *tuple*. The query's answer contains all the produced tuples, without taking into consideration any order.

The above description does not clarify how a variable could be bound to the content of an XML fragment which contains both elements and string values. For instance consider that a variable *var* is bound to the content of the following XML fragment:

```
<city>
   Athens
   <country> Greece </country>
</city>
```

It is not clear whether the edge *city* points to a leaf node or to an internal node. In such cases, extra edges labeled as $CDATA$ encapsulate the string values [DFF+99]:

```
<city>
   <CDATA> Athens </CDATA>
   <country><CDATA> Greece </CDATA></country>
</city>
```

It is now clear that the variable *var* is bound to the OID of the internal node that is pointed by the edge *city*.

2.1.4 Splitting pattern expressions

Consider the following pattern expression:

$$WHERE\ E_1 ... E_n\ IN\ Source$$

where $E_1, ..., E_n$ are distinct XML-QL patterns. The pattern expression is equivalent to the following set of pattern expressions:

$$WHERE\ E_1\ IN\ Source, ..., E_n\ IN\ Source$$

2.1.5 Object Identifiers

Each element appearing in the CONSTRUCT part of a query includes an OID which is probably not explicitly defined. However, it is implicitly determined by a Skolem function. For instance, consider the function *GreekCities*() in Subsection 2.1.2. For each binding of the query's variables with the source data, a distinct *city* element is produced. So a new identifier is constructed for each distinct binding, i.e., we should define Skolem functions that produce a unique value for each row of the table R, which contains the variable bindings. In Subsection 2.1.3, we mentioned that two different rows in R are produced when at least one pattern expression of P matches two different XML subgraphs of the source data. Two subgraphs are different if they contain at least two nodes with different values. By different values, we mean either different OIDs or unequal string values in the case of leaf nodes. We should mention that two graphs can be different even if they contain identical leaf nodes. Therefore, in order to ensure that a different result is produced for each row in R: (i) a different identifier variable is assigned to each element that appears in P, and (ii) a unique OID is created for each element in the constructor of the query by means of Skolem functions defined over the above variables. If $id_1, ..., id_p$ are the identifier variables that are assigned to the elements that appear in the pattern part of a query, each element $< e_i >$ in the constructor part can be written as $< e_i \ ID = f_i(id_1, ..., id_p) >$. The pattern part may contain several pattern expressions. It is conceivable that when the OID of an element is explicitly defined, no additional transformation is applied.

According to the above description, the function *GreekCities*() is written as follows:

```
V: function GreekCities(){
    WHERE
      <city ID = $i1>
         <name ID = $i2> $c </>
         <population ID = $i3> $p </>
         <country ID = $i4> Greece </>
      </> IN ``cities.xml'',
    CONSTRUCT
       <city ID = f1($i1,$i2,$i3,$i4)>
```

```
            <name ID = f2($i1,$i2,$i3,$i4)> $c </>
            <inhabitants ID = f3($i1,$i2,$i3,$i4)> $p </>
        </>
}
```

In the case of RXL queries [MF00], the implied OID definitions are derived by using the keys of the relations that are involved in the FROM clause of the query.

The problem changes when the pattern part of the user query involves regular path expressions. A regular path expression does not require a specific schema for the source data. Since the schema of the data that can be matched by the pattern part is not known, it is not possible to derive the implicit OIDs of the elements in the constructor of the query. In Chapter 3, we see that due to this fact, there are cases where no rewritten query can be constructed, when regular path expressions appear in the pattern part of the view.

2.1.6 Value equality

In the case of leaf nodes, two nodes are equal if they correspond to equal string values. In the case of internal nodes, the nodes are equal if they correspond to the same OID. Consider the following query which returns publications that are written by more than one authors:

```
WHERE
      <publications>
         <author>
            <name> $n1 </>
            <publication> $p1 </>
         </>
         <author>
            <name> $n2 </>
            <publication> $p2 </>
         </>
      </> IN ``publications.xml'',
      $n1!=$n2, $p1=$p2
CONSTRUCT
     <author>
         <name> $n1 </>
         <publication> $p1 </>
```

```
</>
<author>
    <name> $n2 </>
    <publication> $p1 </>
</>
```

If the *publication* elements in the source data contain atomic values, then the comparison between $\$p1$ and $\$p2$ is based on the string values to which they are bound. Otherwise, the OIDs to which the two variables are bound should be identical, i.e., $\$p1$ and $\$p2$ should correspond to the same publication object. Content equality is not adequate.

We can observe that the publications that are bound to the variable $\$p1$ are reproduced twice in the result. Since each publication element that is created has a unique OID (no Skolem functions are declared), each binding results in two different nodes, referenced by two different publication edges. So now, these nodes refer to two different publications.

2.2 Views

We do not provide any special view definition syntax. Views are defined as normal XML-QL functions without arguments and schema restrictions for the output. References to views are expressed as function calls. However, we restrict our problem to function calls that appear after 'IN', and not in the interior of elements. In addition, we do not study view definitions with constructors that contain nested sub-queries.

A well-formed XML document should have a single root element tag. The XML-QL's implementation [Fer99] adds a tag named *XML* to the result of a query. We can assume that the result of a view query has a root element named *view*. A query over a view should take into consideration this external element.

2.3 Query Rewriting

Consider a view defined as an XML-QL query V. V takes as input an XML document XD_1 and returns an XML document $XD_2 = V(XD_1)$. Moreover, consider a user query Q, which takes as input XD_2 and returns an XML document XD_3. We are required to construct an XML-QL query Q', which takes as input the XML document XD_1 and returns XD_3, i.e., $Q'(XD_1) = Q(XD_2) = Q(V(XD_1)) = (Q \circ V)(XD_1)$.

We give a brief description of the intuition behind the rewriting process that we follow. The structure of an XML-QL constructor is very similar to the structure of XML documents. When no explicitly defined OIDs and nested sub-queries appear in the constructor of V, its schema is similar to the schema of the individual tuples that are produced by the query. In other words, the element tags and attributes that appear in the constructor of V also appear in each tuple of the resulting XML document XD_2. The idea is to match the user-query pattern with the constructor of V instead of matching it with XD_2. Since a constructor as opposed to an XML document also contains variables, the result of the matching procedure is a set of mappings between variables, constants and element tags. The matching and composition algorithms are presented in Chapter 3.

Explicitly defined OIDs and sub-query blocks in the constructor of a query may result in aggregated XML data, so the structure of the produced tuples may be different from the structure of the constructor. As mentioned in the previous section, sub-query blocks in the constructor of the view are not studied in this thesis. Explicitly defined OIDs in the view definition are studied in Chapter 4. Finally, we also assume that no regular path expressions appear in the pattern part of the view and the user query.

2.3.1 Constructor and Pattern trees

We consider that all the variables, as well as the Skolem function declarations that appear in a query are stored in a symbol table. We denote as T_Q the symbol table that contains all the variables and Skolem function declarations that appear in the body of a query Q. For each variable, we store its name, while for each Skolem function, we store its name, and its list of parameters. Parameters may be either constants or pointers to symbols in T_Q.

We represent constructors by trees instead of graphs, which facilitates the matching procedure. The internal nodes of a *constructor tree* represent elements, while the leaf nodes represent non-tag variables and constants (variables and constants that are not included in an element tag) that appear in a constructor. Handling the tag elements as terminal symbols and ignoring the end tags, a constructor tree is similar to the syntax tree of the corresponding constructor.

A constructor tree can be expressed using the following grammar:

```
 1:          Node ::= LeafNode | InternalNode
 2:      LeafNode ::= LeafVariable | LeafConstant
 3:  InternalNode ::= (TagLabel, Children, AttributesSet)
 4:      TagLabel ::= TagVariable | ElementName
 5:      Children ::= Node | Children, Node

 6: AttributesSet ::= {OID, Attributes}
 7:           OID ::= (ID, <Skolem>)
 8:    Attributes ::= Attribute | Attributes, Attribute
 9:     Attribute ::= (AttrName, AttrValue)
10:     AttrValue ::= AttrVariable | AttrConstant

11:  LeafVariable ::= <VAR>
12:   TagVariable ::= <VAR>
13:  AttrVariable ::= <VAR>
14:   ElementName ::= <IDENTIFIER>
15:      AttrName ::= <IDENTIFIER>
16:  LeafConstant ::= <STRING>
17:  AttrConstant ::= <STRING>
```

The terminal symbols $< VAR >$ and $< Skolem >$ represent references to variables and Skolem function definitions, respectively, in the symbol table. A node is descendent

of another node that represents an element e_{parent} if it represents an element, a variable or a constant which is nested under e_{parent}. The root node represents the root element that is produced by the query and is named as *view* if Q is a view definition. In the sequel, internal nodes are named after their label (TagLabel), and leaf nodes are named after the variable or the string value that they represent.

A graphical representation of a constructor tree is displayed in Figure 2.1.

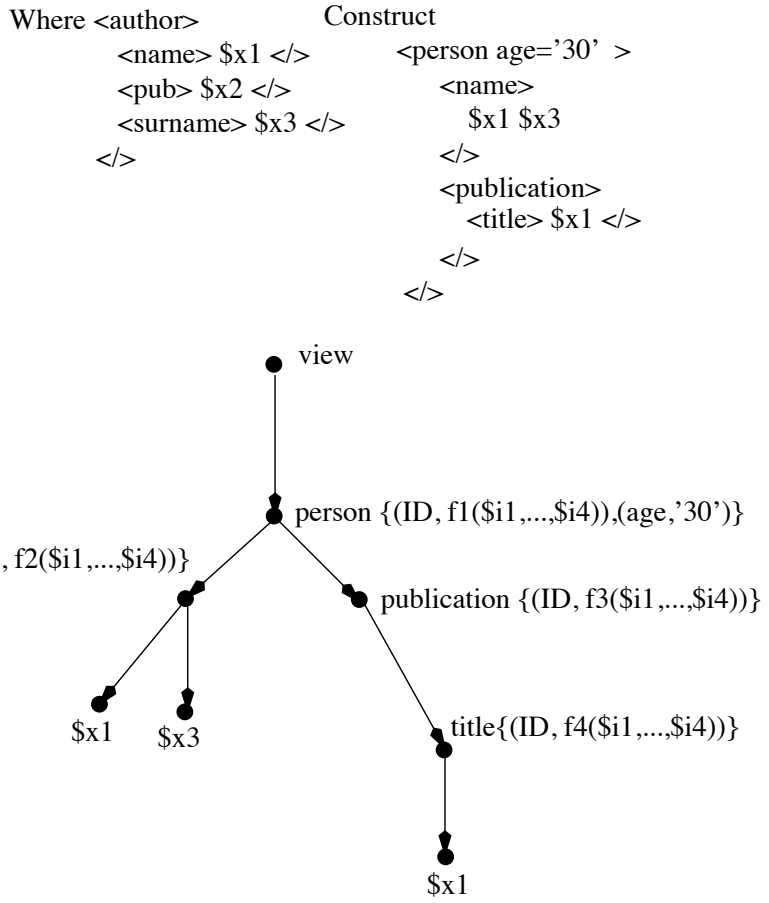

Figure 2.1: A query and the corresponding constructor tree

We consider two sets of primitive functions that can be used to access and handle tree nodes:

Symbol(b):**boolean** Return *true* if b can reduce to the non-terminal symbol *Symbol* of the grammar presented above. For instance, if *node* is a leaf node that represents a

variable, the function calls $LeafNode(node)$ and $LeafVariable(node)$ return *true*, while the function call $LeafConstant(node)$ returns *false*.

get$Symbol$**(b):**$Symbol$ Return the $Symbol$ part of b. For instance, if $node$ is an internal node with the form $(author, \{child_1, child_2\}, \{(ID, f(x_1))\})$, then the function call $getTagLabel(node)$ returns $author$, and the function call $getOID(getAttributesSet(node))$ returns $(ID, f(x_1))$.

Given a node of a constructor tree, we can reproduce the *constructor segment* that corresponds to the subtree rooted by this node. A constructor segment is defined by the non-terminal symbol *Contents* in the XML-QL grammar presented in Appendix A.

A pattern tree $PT_{Q,i}$ is defined similarly and represents the i^{th} pattern expression of the pattern part of Q. Skolem functions do not appear in pattern trees. OID attributes may also be omitted. So, the lines 7-8 of the grammar presented above can be simply replaced by the following line:

```
7: AttributesSet ::= '{' Attributes '}'
```

The OID attribute is handled as any other attribute. We should mention that leaf nodes in a pattern tree cannot have sibling nodes, since patterns with the form `<e>` $\$var_1$ $\$var_2$ `</>` or `<e>` $\$var$ *const* `</>` are not valid.

We assume that the root node of a pattern tree has always a single child. Pattern expressions of the form:

```
WHERE
    <view>
       <e_1>...</>
       ...
       <e_2>...</>
    </> IN V
```

are always translated into:

```
WHERE
     <view>
```

```
   <e_1>...</>
</> IN V
,...,
<view>
   <e_2>...</>
</> IN V
```

After this transformation, applying a pattern expression over the view is equivalent to applying the pattern expression over each distinct view tuple.

2.3.2 Matching pattern trees to constructor trees

The procedure of matching a pattern part of a query to the constructor of another query is analogous to matching the pattern part of a query to an XML document. A certain pattern can match an XML document in several different ways, resulting in different variable bindings. Similarly, a pattern can match a constructor in several different ways. Each of these *matchings* involves several independent mappings that involve variables, constants, element labels, constructor segments and patterns.

The matching procedure between a pattern expression and a constructor is performed by means of the corresponding pattern and constructor tree. Nodes of the pattern tree are matched to nodes of the constructor tree by comparing their properties, e.g., the label names and the attribute values. Since pattern and constructor trees also involve variables, the comparison may also involve variables. Since variables can be bound to any value, the comparison between two variables or between a variable and a constant always evaluates to true.

Consequently, an internal node of a pattern tree can match an internal node of the pattern tree if the comparison between their labels, attribute names and attribute values evaluates to true. All the attributes of the pattern node should be matched, but it is not necessary to match all the attributes of the constructor node. For instance, the pattern node that represents `<$t continent=''Europe''>` can match the constructor node that represents `<country ID=f($x) continent=$c>`.

The comparison between constant leaf nodes is based on their string values. However, a leaf node may consist of a variable. Such a variable can be bound not only to atomic values, but also to OIDs representing whole XML subgraphs. In order to handle this case, we consider that a variable in a leaf node can be matched to an internal node or a collection of sibling nodes.

Not any pair of nodes is compared. A pattern node can match a constructor node only if their parent nodes can be matched. In other words, in order to match two nodes, we should match all their ancestors.

Matches between components of the constructor and the pattern tree are represented by triples that obey the following syntax:

```
 1:      Match ::= (TagVariable, TagVariable, 'null')
 2:              | (TagVariable, ElementName, 'null')
 3:              | (ElementName, TagVariable, 'null')

 4:              | (AttrVariable, AttrVariable, 'null')
 5:              | (AttrVariable, AttrConstant, 'null')
 6:              | (AttrConstant, AttrVariable, 'null')

 7:              | (LeafNode, LeafNode, <Skolem>)
 8:              | (LeafVariable, InternalNode, <Skolem>)
 9:              | (InternalNode, LeafVariable, 'null)
10:              | (LeafVariable, SiblingNodes, <Skolem>)
11:              | (SiblingNodes, LeafVariable, 'null')

12: SiblingNodes ::= Node SiblingNodes | Node Node
```

A match can involve tag labels, attribute values and nodes that are matched during the matching procedure. When two internal nodes are matched, the produced matches represent their matched attributes and labels that involve variables.

The first argument of a match represents the matched component of the pattern tree, while the second argument represents the matched component of the constructor tree. As shown above, leaf nodes that represent variables can match collections of sibling nodes. If the leaf node belongs to the pattern tree, the meaning of such a match is that the corresponding variable should be bound to the OID of an XML subgraph whose

structure is defined by the constructor segment that the matched nodes represent. This OID is represented by the third argument of the triple. It is used when the variable participates in equality filters and the equality should be based on the OID. If the leaf node belongs to the constructor tree, the meaning is that the pattern represented by the matched nodes should be applied on the corresponding variable.

In general, a match represents a set of bindings of a certain subset of the user-query variables.

Example 2.3 *Consider the following pair of view and user query:*

```
V: function View(){
    WHERE
      <a ID = $i1>
         <$t1 ID = $i2> $x1 </>
         <$t2 ID = $i3> $x2 </>
      </> IN ``source.xml'',
    CONSTRUCT
       <b ID = f1($i1,$i2,$i3) a='Hi'>
          <$t ID = f2($i1,$i2,$i3)> $x1 </>
          <b2 ID = f3($i1,$i2,$i3)> $x2 </>
       </>
}

Q: WHERE
     <view>
        <b a=$v>
           <b1> $y1 </>
           <b2><b3> $y2 </></>
        </>
     </> IN View()
   CONSTRUCT
       <result>
          <d1> $y1 </>
          <d2> $y2 </>
       </>
```

The following set of matches can be identified: ($v, 'Hi', null), ('b1', $t, null), ('b2', $t, null), ($y1, $x1, f2($i1,$i2,$i3)), (b3, $x1, f2($i1,$i2,$i3)), and (b3, $x2, f3($i1,$i2,$i3)).

The purpose of the matching procedure between a constructor and a pattern tree is to match the whole pattern tree, but not necessarily the whole constructor tree. Moreover,

the same node of a constructor tree can be matched by several nodes of the pattern tree, but each node of the pattern tree should be matched only once.

Several different matchings can exist between a constructor and a pattern tree, i.e., the pattern tree can match the constructor tree in several different ways. Each matching is represented by the corresponding set of matches. Each set of matches should obey the following restrictions: (i) there should be a single match for each tag or attribute variable of the pattern tree; and (ii) each leaf node of the pattern tree or exactly one of its ancestors should appear in a single match.

Considering the matches presented in Example 2.3, two possible matchings can be identified.

1. The first contains the matches ($v, 'Hi', null), ('b1', $t, null), ($y1, $x1, f2($i1,$i2,$i3)) and (b3, $x2, f3($i1,$i2,$i3)).

2. The second contains the matches ($v, 'Hi', null), ('b1', $t, null), ('b2', $t, null), ($y1, $x1, f2($i1,$i2,$i3)) and (b3, $x1, f2($i1,$i2,$i3)).

The matching procedure is presented in detail in Chapter 3.

Chapter 3

Query composition - Base case

In this chapter, we present our solution to the problem of composing queries that are defined over XML views. Explicit OID definitions are not studied in this chapter. They are introduced in Chapter 4.

3.1 Formalizing the problem

Consider a view $V(db)$ which is defined by a query $V(db)$:

$$\begin{array}{l}\text{WHERE} \\ \quad P_V(db, x_1, ..., x_n) \\ \quad F_V(x_1, ..., x_n) \\ \text{CONSTRUCT } C_V(x_1, ..., x_n)\end{array} \qquad (3.1)$$

where P_V is the pattern part, F_V is the filter part, C_V is the constructor, and $X = \{x_1, ..., x_n\}$ is the set of variables that appear in V. The query is applied on the XML document db.

Consider the user query $Q(V)$ that is defined over V:

$$\begin{array}{l}\text{WHERE} \\ \quad P_Q(V, y_1, ..., y_m) \\ \quad F_Q(y_1, ..., y_m) \\ \text{CONSTRUCT } C_Q(y_1, ..., y_m)\end{array} \qquad (3.2)$$

where P_Q, F_Q and C_Q are defined over the variables $Y = \{y_1, ..., y_m\}$, and $X \cap Y = \emptyset$[1].

Our problem is the construction of a query $Q'(db)$ which is defined over the source data and returns the same result as Q. Q' has the following format:

$$\begin{array}{l} \text{WHERE} \\ \quad P_{Q'}(db, x_1, ..., x_n) \\ \quad F_{Q'}(x_1, ..., x_n, y_1, ..., y_m) \\ \text{CONSTRUCT } C_{Q'}(x_1, ..., x_n, y_1, ..., y_m) \end{array} \qquad (3.3)$$

The rewriting process involves two sequential steps. During the first step, we discover the matchings between P_Q and C_V. These matchings are used in the second step to produce the rewritten query Q'. The first step is presented in Section 3.2, and the second one is presented in Section 3.3.

3.1.1 Assumptions

We assume that P_Q contains only a single pattern expression, which is the one that refers to V. We introduce multiple pattern expressions in P_Q in Section 3.5. We also assume that there are no equality conditions between variables in $\{y_1, ..., y_m\}$. This means that there are no filters $y_i = y_k$ in F_Q, and furthermore, each y_i appears only once in P_Q. Equality conditions between variables are studied in Section 3.6. Finally, we assume that both P_V and P_Q do not contain regular path expressions. In Section 3.4, we justify this assumption.

3.2 Matching the user query against the view

3.2.1 The matching algorithm

Let PT_Q be the pattern tree that represents the pattern part P_Q of the user query Q, and CT_V be the constructor tree that represents the constructor C_V of the view V.

[1] If $X \cap Y \neq \emptyset$, we can always find a renaming X' of the set of variables in X, such that $X' \cap Y = \emptyset$.

Algorithm 3.1 describes how the matchings between PT_Q and CT_V are discovered. The algorithm consists of the function *findMatchings($node_p$, $node_c$)*, which performs the matching procedure recursively. It takes as arguments a node $node_p$ from CT_V and a node $node_c$ from PT_Q and returns all the matchings between the subtrees that are defined by these two nodes. We use the functions that are defined in Section 2.3 to access and handle nodes.

Lines 4-9 check whether the nodes represent constants. If they both represent equal constants, a single matching exists that contains only the match between the two nodes. Lines 12-14 check whether the tag labels of the two nodes are identical or at least one of them is a tag variable. If none of these situations holds, no matching exists. Lines 17-20 checks whether the attributes of the elements that correspond to the nodes match. The function *matchAttr($node_p$, $node_c$)* returns false if and only if (i) both $node_p$ and $node_c$ represent internal nodes, and (ii) there is at least one attribute in $node_p$ which is either not included in the attribute list of $node_c$ or the corresponding attribute values do not match. Two attribute values match if either their values are equal or at least one of them is a variable. The function *findAttributeMatches($node_p$, $node_c$)* returns the matches that contain the matched pairs of attribute values, when at least one value in each pair is a variable. Lines 27-30 check whether $node_p$ has a single child which represents a variable. If this is the case, one single matching exists containing the match between this child and the whole collection of subnodes of $node_c$.

Lines 34-43 handle the case when more than one nodes are under $node_p$. None of these nodes can be a variable. In this case, we consider each child $child_p$ of $node_p$ independently and find the matchings between $child_p$ and the children of $node_c$. After all the combinations of children pairs have been identified, a number of different sets of matchings are discovered and stored in the list *matchings*[]. Each set of matchings corresponds to a different pair of children nodes of $node_p$ and $node_c$. The function *merge(matchings[])* results in a single set of matchings S by combining the matchings in

matchings[] as follows. Each set of matchings corresponds to a single child or a subset of children of $node_p$. We consider all the combinations $combination_i$ of matchings sets S_{ij}, such that each $combination_i$ corresponds to disjoint children subsets, and every child of $node_p$ appears in such a subset. The matchings sets S_{ij} that correspond to a certain $combination_i$ are combined resulting in a single set S_i. More specifically, each matching in S_i is produced by considering one matching from each S_{ij} and merging them. All the possible combinations between matchings in each S_{ij} are considered. Following this process we ensure that every leaf node in the subtree under $node_p$ or one of its ancestors appears in a single match of each matching in S_i. Finally, all the sets S_i are merged into a single set S that contains all the matchings.

If at least one of the tag labels of $node_c$ or $node_p$ is a variable, the match between the two labels is added to each individual matching by calling the function *addMatch()* (Lines 47-50). Finally, Lines 53-54 add the attribute matches to each matching by calling the function *addMatches()*.

Algorithm 3.1

Input: *root node of* CT_V, *root node of* PT_Q

Output: *the set of all the matchings between* CT_V *and* PT_Q

```
1: function Matchings findMatchings(node_p, node_c){
2:
3: % if both nodes represent constants
4:     if (LeafConstant(node_c) or LeafConstant(node_p))
5:         if (LeafConstant(node_c) and LeafConstant(node_p) and node_c = node_p)
6:             return new Matchings(
7:                                 new Matching(
8:                                     new Match(node_p, node_c, null)));
9:         else return null;
10:
11: % If the tag labels do not match
12:     if (!tagVariable(getTagLabel(node_c)) and !tagVariable(getTagLabel(node_p))
13:           and getTagLabel(node_c)!=getTagLabel(node_p))
14:         return null;
15:
16: % Find matches between the attribute values
17:     if (!matchAttr(getAttributesSet(node_p), getAttributesSet(node_p)))
18:         return null;
```

```
19:     else attributeMatches ← findAttributeMatches(
20:                                   getAttributesSet(node_p), getAttributesSet(node_p));
21:
22: % Get the children of the nodes
23:     children_p ← (getChildren(node_p)));
24:     children_c ← (getChildren(node_c)));
25:
26: % If node_p has a single child that represents a variable
27:     if LeafVariable(children_p)
28:         return new Matchings(
29:                     new Matching(
30:                         new Match(children_p, children_c, getOID(getAttributes(node_p))));
31:
32: % If node_p has multiple children, find the matchings
33: % that correspond to all the possible pairs of subtrees and merge them
34:     for each child_c[i] in children_c
35:         if !LeafVariable(child_c[i])
36:             for each child_p[j] in children_p
37:                 matchings[i, j] ← findMatchings(child_p[j], child_c[i]);
38:         else
39:             for each subchildren_p[j] in children_p
40:                 matchings[i, j] ← new Matchings(
41:                                       new Matching(
42:                                         new Match(subchildren_p[j], child_c[i],null);
43:     mergedMatchings ← merge(matchings[]);
44:
45: % If the tag label of at least one node is variable,
46: % include the corresponding match
47:     if (tagVariable(getTagLabel(node_c)) or tagVariable(getTagLabel(node_p)))
48:         for each matching in mergedMatchings
49:             addMatch(matching,
50:                     new Match(getTagLabel(node_p), getTagLabel(node_c), null));
51:
52: % Include the attribute matches
53:     for each matching in mergedMatchings
54:         addMatches(matching, attributeMatches);

55:     return mergedMatchings;
57: }
```

3.2.2 Deriving the variable bindings from the matchings

As mentioned in Section 2.3, the purpose of the matching procedure is to discover how the set of variables of the user query are mapped to the source XML document, without materializing the view. In this subsection, we show that all the bindings that are produced

when the pattern part of the user query is applied over the view can be derived from the matchings that the matching algorithm produces. We also show that no additional bindings are derived from the matchings. In Section 3.3, we use this fact to prove that the rewriting algorithm produces the correct composition query.

Let $X^i = \{x_1^i, ..., x_n^i\}$ be a binding of the set X of the view variables, and let t_i be the corresponding tuple that is produced by V. The structure of V's constructor $C_V(X)$ defines the structure of the produced tuples. Therefore $t_i = C_V(X^i)$. Instead of matching the pattern part $P_Q(Y)$ of the query with each tuple $C_V(X^i)$, the matching algorithm matches $P_Q(Y)$ with $C_V(X)$ by following a similar procedure.

The matching algorithm does not bind variables to specific constants or XML graphs. It finds matches involving components of the view constructor (constants, variables and constructor segments), which include variables that are not yet bound to specific values. However, we should ensure that the matching algorithm produces matchings that derive all the actual bindings of the user-query variables, if the filter part of the user query is ignored. In addition, the matching algorithm should not produce matchings that derive bindings that cannot be produced by $P_Q(Y)$.

Consider a certain binding of the set of variables Y, when applying $P_Q(Y)$ over the output of V. This binding results after matching $P_Q(Y)$ to a certain tuple t_i produced by V. Let X^i be the binding of V's variables that results in t_i. Each variable $y_j \in Y$ is bound to $C_{V,j}(X^i)$, where $C_{V,j}(X^i)$ is a constant or an XML subgraph constructed by the component $C_{V,j}(X)$ of the constructor of V. A binding between y_j and a subgraph is represented by the OID of the node that parents the subgraph.

When P_Q is applied over C_V by means of the rewriting algorithm, there is a matching, where each variable y_j is matched to $C_{V,j}(X)$, which evaluates to $C_{V,j}(X^i)$ when the set of variables X is bound to X^i. If y_j is not a tag or attribute variable, the algorithm also records the OID definition of the parent element of $C_{V,j}(X)$ as the third argument of the match triple. This definition represents the OIDs to which y_j is actually bound

and is used when y_i is involved in equality filters with other variables. Such filters are introduced in Section 3.6.

The previous situation is not always possible. The binding X^i may involve non-atomic values, i.e., a variable $x_k \in X$ may be bound to a subgraph G. Assume that when P_Q is applied over t_i, a variable y_j is bound to a subgraph G_j of G. Even if G_j does not depend on the binding X^i of the set of variables X, we can denote it as $C_{V,j}(X^i)$. The matching algorithm cannot directly represent this binding, since $C_{V,j}(X^i)$ is not available without materializing the view. Instead, it records a match between a sub-pattern $P_k(Y)$ and x_k. Applying $P_k(Y)$ on x_k, y_j is bound to $C_{V,j}(X^i)$, when x_k is bound to G. We conclude, that given a binding of the set of the user query variables, there is a matching produced by the matching algorithm that represents it as described above.

Consider now a matching M returned by the matching algorithm. Assume that each variable y_j, where $j = 1..q$, is matched to a component $C_{V,j}(X)$ of the constructor of V. Moreover, there are matches in which a variable $x_k \in X$ is matched to a pattern $P_k(Y)$ or to a constant c_k. M represents a set of bindings of the variables Y with the following properties. Given a binding X^i of V's variables, each variable y_j, where $j = 1..q$, is bound to $C_{V,j}(X^i)$, while the binding of each y_j, where $j = q+1..m$, is derived by applying a certain pattern $P_k(Y)$ on x_k^i, where $x_k^i \in X^i$. A match between a variable x_k and a constant c_k is translated as follows. The bindings are derived only if $x_k^i = c_k$, where $x_k^i \in X^i$. It is clear that when P_Q is applied over the tuple $C_V(X^i)$ that is generated by V, all these bindings are produced.

3.3 The rewriting algorithm

3.3.1 Detailed algorithm

A formal description of the rewriting procedure is presented by Algorithm 3.2. The algorithm first calls the *findMatchings()* function to discover the matchings between the

pattern tree of the user query and the constructor tree of the view query. For each matching, it produces a different sub-query block. More specifically, if $\{M_1, ..., M_n\}$ is the set of the discovered matchings, the final rewritten query has the form $\{Q'_1\}...\{Q'_n\}$, where Q'_i is the sub-query that corresponds to M_i. Since equality between variables in the user query is not handled by the algorithm, the third argument of the match triples is not used.

Algorithm 3.2

Input: *a view definition V, and a user query Q defined over V*

Output: *the composition query $Q' = Q \circ V$*

```
1: function Query compose(Q, V){
2:
3: % Rename the variables in V so that they do not conflict
4: % with the variables names of Q
5:     renameVariables(V,Q);
6:
7: % Get the parts of Q and V that will form the corresponding parts
8: % of the rewritten query
9:     P_V ← getPatternPart(V);
10:    C_Q ← getConstructorPart(Q);
11:    F_V ← getFilterPart(V);
12:    F_Q ← getFilterPart(Q);
13:
14: % Initialize the rewritten query Q'
15:    Q' ← new Query();
16:
17: % Find the matchings between V's constructor tree and Q's pattern tree
18:    matchings ← findMatchings(getPatternTree(Q), getConstructorTree(V));
19:
20: % For each matching, create a separate sub-query
21:    for each matching in matchings {
22:
23: % Initialize the pattern, filter and constructor parts of the sub-query
24:        P_SQ ← P_V;
25:        F_SQ ← mergeFilters(F_Q,F_V);
26:        C_SQ ← C_Q;
27:
28: % If a variable in V is matched to a constant
29:        for each (m_p, m_c, id) in matching {
30:            if (LeafConstant(m_p) or AttrConstant(m_p) or ElementName(m_p))
31:                addFilter(F_SQ,m_c = m_p);
32:
```

```
33: % If the matched component that corresponds to Q is a variable
34:             if (LeafVariable(m_p) or AttrVariable(m_p) or TagVariable(m_p)){
35:                 substitute(C_SQ,m_p,m_c);
36: % If the variable cannot be substituted in the filter part, the matching is rejected
37:                 if !substitute(F_SQ,m_p,m_c) goto 47;
38:             }
39:
40: % If a variable in V is matched to internal nodes of Q's pattern tree
41:             if (SiblingNodes(m_p) or InternalNode(m_p))
42:                 addPattern(P_SQ, constructPattern(m_p) IN m_c);
43:
44: % Construct the sub-query and add it to Q'
45:         SQ ← new Query(P_SQ,F_SQ,C_SQ);
46:         addSubquery(Q',SQ);
47:     }
48:     return Q';
49: }
```

We give an explanation of the new functions that appear in the algorithm. The function *rename(query*$_1$*,query*$_2$*)* in Line 5 renames the variables in *query*$_1$, so that they do not conflict with the variable names in *query*$_2$. The functions *getX(query)*, where X is a query part (Lines 9-12), return the corresponding part of *query*, while the functions *getConstructorTree(query)* and *getPatternTree(query)* return the constructor and pattern tree of *query*. The function *mergeFilters(filter*$_1$*, filter*$_2$*)* (Line 25) produces a filter that contains both filters *filter*$_1$ and *filter*$_2$. The function *substitute(queryPart, var, component)* (Lines 35, 37) replaces all the appearances of variable *var* in *queryPart* by the constant, variable, element label or constructor segment that corresponds to *component*. If *component* represents an internal node, i.e., *InternalNode(component)* is true, or a collection of sibling nodes, i.e., *SiblingNodes(component)* is true, *var* is substituted by the constructor segments that correspond to the constructor subtrees parented by these nodes. Since each subtree results in a different segment, *var* is replaced by the concatenation of all the individual segments. A constructor segment cannot participate in a filter. Therefore, if the first argument of *substitute()* is a filter, and the third argument is an internal node or a collection of nodes, the function returns false, and the corresponding matching is rejected (Line 37).

The function *addPattern(p, expr)* adds a new pattern expression *expr* into the pattern part *p* of a query (Line 41). The translation of a pattern tree *tree* into a pattern *pattern* is performed by the function *addPattern(tree, pattern)*. Finally, the function *addSubquery(query, subquery)* adds *subquery* to *query* as a sub-query block.

Example 3.1 *Consider the following pair of a view and a query:*

```
V: function View(){
    WHERE
      <a ID = $i1>
        <b ID = $i2>
           <b1 ID = $i3> $x1 </>
           <b2 ID = $i4> $x2 </>
        </>
      </> IN ``source.xml''
    CONSTRUCT
       <c ID = f1($i1,$i2,$i3,$i4)>
         <c1 ID = f2($i1,$i2,$i3,$i4)> $x1 </>

         <c2 ID = f2($i1,$i2,$i3,$i4)> $x2 </>
         <c2 ID = f2($i1,$i2,$i3,$i4)>
            <c5 ID = f2($i1,$i2,$i3,$i4)> $x1 </>
         </>

         <c3 ID = f2($i1,$i2,$i3,$i4)>
            <c4 ID = f2($i1,$i2,$i3,$i4)>
               <c6 ID = f2($i1,$i2,$i3,$i4)> Hello </>
            </>
         </>
       </>
}

Q: WHERE
     <view>
        <c>
           <c1> 100 </>
           <c2><c5> $y1 </></>
           <c3> $y2 </>
        </>
     </> IN View(),
     $y1 < 100
   CONSTRUCT
       <result>
          <d1> $y1 </>
          <d2> $y2 </>
       </>
```

The implicit OIDs have already been included in the element tags of the view's constructor. The matching algorithm discovers the different matchings between the pattern tree of Q and the constructor tree of V. The first matching contains the matches $(100, \$x1, \_)$, $(c5, \$x2, \_)$ and $(\$y2, c4, \_)$, and the second matching contains the matches $(100, \$x1, \_)$, $(\$y1, \$x1, \_)$ and $(\$y2, c4, \_)$. The third argument of each match is ignored. Each matching corresponds to a different sub-query. The first sub-query involves the following transformations.

(i) The filter $\$x1 = 100$ is added in the filter part of the sub-query.

(ii) The variable $\$y2$ is substituted by the constructor segment

$$< c4\ ID = ... >< c6\ ID = ... > Hello < / >< / >$$

(iii) The pattern expression $< c5 > \$y1 < / >\ IN\ \$x2$ is added to the pattern part of the sub-query.

The second sub-query also involves the first and the second transformations. In addition, $\$y1$ is substituted by $\$x1$.

The query that is produced is presented below:

```
Q':
 { WHERE
     <a ID = $i1>
        <b ID = $i2>
           <b1 ID = $i3> $x1 </>
           <b2 ID = $i4> $x2 </>
        </>
     </> IN ``source.xml'',
     <c5> $y1 </> IN $x2,
     $y1 < 100, $x1 = 100
   CONSTRUCT
       <result>
          <d1> $y1 </>
          <d2>
             <c4 ID = f1($i1,$i2,$i3,$i4)>
                <c6 ID = f2($i1,$i2,$i3,$i4)> Hello </>
             </>
          </>
       </>
 }
{ WHERE
     <a ID = $i1>
```

```
        <b ID = $i2>
           <b1 ID = $i3> 100 </>
           <b2 ID = $i4> $x2 </>
        </>
      </> IN ``source.xml'',
      $x1 < 100, $x1 = 100
    CONSTRUCT
        <result>
           <d1> $x1 </>
           <d2>
              <c4 ID = f1($i1,$i2,$i3,$i4)>
                 <c6 ID = f2($i1,$i2,$i3,$i4)> Hello </>
              </>
           </>
        </>
  }
```

At this point, we can explain why implicit OIDs are included in the element tags of the view's constructor as described in Subsection 2.1.5. The XML-QL data model is object based, and XML nodes represent distinct objects which are identified by unique object identifiers. When the OID of a tag element in the constructor of a query is not explicitly defined, then, for each binding of the query's variables, one distinct such element is constructed containing a unique OID.

Consider the following view:

```
V: WHERE ...
   CONSTRUCT
          ...
           <e>...</>
          ...
```

When V is applied over an XML document, for each binding of its variables, a different e element is constructed. Now, consider the following query:

```
Q: WHERE ...
     ... $y ... IN V
   CONSTRUCT
          ... $y ...
          ... $y ...
```

Assume that when the pattern tree of Q is matched to the constructor tree of V, $\$y$ is matched to the node that represents e. If OIDs are ignored, the rewritten query is as

follows:

```
Q': WHERE ...
    CONSTRUCT
          ... <e>...</> ...
          ... <e>...</> ...
```

The above rewritten query constructs two different e elements, i.e., elements with different OIDs, for each binding of the query variables. This is not the output that Q generates. V produces a single e element for each binding. When Q is applied over the resulting document, and $\$y$ is bound to the subgraph that includes an element e, a single element e is created. So, the rewritten query should assign the same OID to two e elements that correspond to the same binding of $\$y$. This is achieved by including the implicit OID definitions in the element tags of V's constructor and preserving them during the matching procedure.

Figure 3.1 exhibits how ignoring implicit OID definitions can lead to incorrect rewritten queries for the view and user query presented in Example 3.1.

3.3.2 Justification of the algorithm

We show that the rewritten query Q' which is produced by Algorithm 3.2 is equivalent to the composition of the user query $Q(Y)$ and the view $V(X)$. In order to simplify our presentation, we assume that no filter conditions appear in Q.

Let $P_V(X)$, $F_V(X)$ and $C_V(X)$ be the pattern, filter and constructor part of V, and $P_Q(Y)$ and $C_Q(Y)$ be the pattern and constructor part of Q. We first apply the matching algorithm and produce a set of matchings. Consider a matching M in this set. For this matching, the rewriting algorithm produces a query block Q'_M which includes: (i) the pattern expression $P_V(X)$ of V; (ii) the filter part $F_V(X)$ of V; (iii) a set of pattern expressions $P_1(Y), ..., P_r(Y)$; (iv) a set of filters $F'(X) = \{(x_1 = c_1), ..., (x_p = c_p)\}$, where $x_1, ..., x_p \in X$; and (v) the constructor $C_Q(C_{V,1}(X), ..., C_{V,q}(X), y_{q+1}, ..., y_m)$. A pattern expression $P_j(Y)$ corresponds to a match between a variable $x_j \in X$ and a pattern, i.e.,

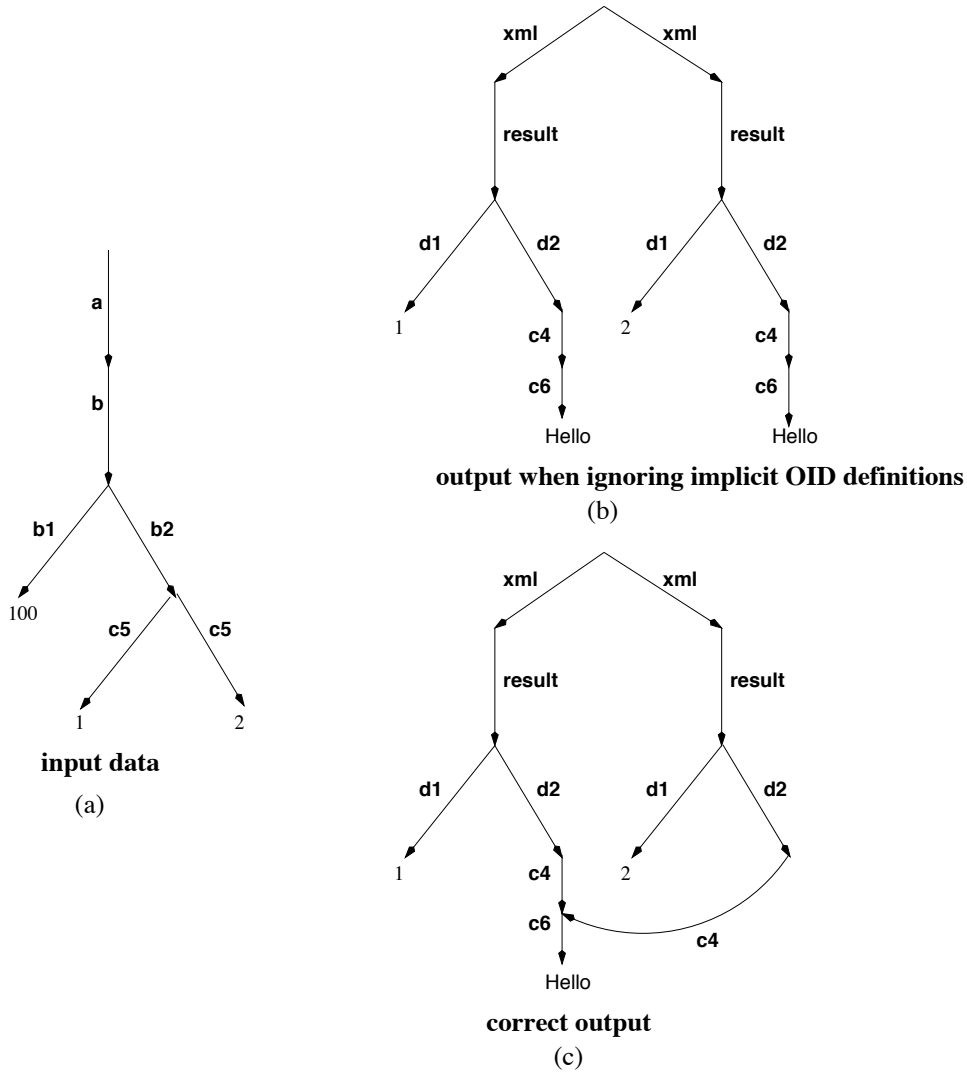

Figure 3.1: Importance of implicit OID definitions

an internal node or a collection of nodes of the corresponding pattern tree. A condition $x_j = c_j$ corresponds to a match between a variable $x_j \in X$ and a constant c_j. Finally, a substitution $C_{V,j}(X)$ in C_Q corresponds to a match between a variable $y_j \in Y$ and a component $C_{V,j}(X)$ of the constructor $C_V(X)$.

Assume that V is applied on an XML document *db*. Let X^i be a binding of the set of variables X, and let $t_i = C_V(X^i)$ be the corresponding tuple that is produced by V. Assume that Q is applied on t_i and a set of bindings is derived. Consider such a

binding Y^k, where each variable $y_j \in Y$ is bound to $C'_{V,j}(X^i)$. The resulting tuple is $t'_k = C_Q(C'_{V,1}(X^i), ..., C'_{V,m}(X^i))$.

As discussed in Subsection 3.2.2, every such binding can be derived from a matching that the matching algorithm produces. Assume that Y^k is derived from M. This means that $C_{V,j} \equiv C'_{V_j}$, where $j \in \{1, ..., q\}$. It also means that there is a pattern expression $P_u(Y)$ which binds y_j to $C'_{V,j}(X^i)$, where $u \in \{1, ..., r\}$ and $j \in \{q+1, ..., m\}$. It also means that $F'(X^i)$ evaluates to true. Consequently, when Q'_M is applied on db, there is a binding $X^i \cup \{C'_{V,q+1}(X^i), ..., C'_{V,m}(X^i)\}$ of its set of variables which results in the tuple $C_Q(C_{V,1}(X^i), ..., C_{V,q}(X^i), C'_{V,q+1}(X^i), ..., C'_{V,m}(X^i)) = t'_k$. So, $Q'_M \subseteq Q$, and therefore, $Q' \subseteq Q$.

Assume now that Q' is applied on db and generates a tuple $t'_i = C_Q(C_{V,1}(X^i), ..., C_{V,m}(X^i))$, which corresponds to a binding $X^i \cup \{y^i_{q+1}, ..., y^i_m\}$ of its set of variables. Let Q'_M be the sub-query of Q' that generates t'_i. Moreover, let M be the matching that corresponds to Q'_M. As showed in Subsection 3.2.2, the matching algorithm does not produce matchings that derive invalid bindings of the user-query variables, i.e., bindings that cannot be derived, when applying the pattern part of the user query on the view. Thus, there is a binding of the set of variables Y, such that y^j is bound to $C_{V,j}(X^i)$. The tuple that Q generates for this binding is $C_Q(C_{V,1}(X^i), ..., C_{V,m}(X^i)) = t'_i$. Therefore, $Q \subseteq Q'$, and $Q' = Q$.

3.4 Regular path expressions

The introduction of regular path expressions in the user query does not require any changes in the rewriting algorithm. However, the matching procedure is more complex than the one presented in Algorithm 3.1. We do not provide a solution here.

Regular path expressions can also appear in a view definition. In Subsection 2.1.5, we concluded that when a query involves regular path expressions we cannot derive the

implicit OIDs in the query's constructor. In addition, in the previous section, we showed that the rewriting can produce incorrect rewritten queries, when the OID definitions are not included in the element tags of the view's constructor. For this reason, we assume that view queries either explicitly define all the OIDs[2] in their constructor or do not contain regular path expressions in their pattern part.

3.5 Multiple pattern expressions in the user query

Our rewriting algorithm considers that a single pattern expression appears in the user query. In this section we present how multiple pattern expressions in the user query can be handled.

Consider that a user query $Q(Y) = Q(Y_1, Y_2)$ contains two pattern expressions $PE_1(Y_1)$ and $PE_2(Y_2)$, where $Y = Y_1 \cup Y_2$ is the set of variables that appear in Q. Since no equality conditions between variables exist, we conclude that (i) $Y_1 \cap Y_2 = \emptyset$, and (ii) the filter part $F_Q(Y)$ of Q can be split into two different sets of filter conditions $F_1(Y_1)$ and $F_2(Y_2)$.

Assume that PE_1 is defined over a view $V_1(db_1, X_1)$, where db_1 denotes an XML document, and X_1 is a set of variables such that $X_1 \cap Y = \emptyset$. Also assume that PE_2 is defined over an XML document db_2. Our goal is to construct a query $Q'(db_1, db_2, X_1, Y_1, Y_2) = Q(V, db_2, Y_1, Y_2)$.

Let X_1^i be a binding of the variables in X_1 when the V_1 is applied on db_1. Similarly, let Y_2^j be a binding of the variables in Y_2 when Q is applied on db_2. For each pair X_1^i, Y_2^j, a set of tuples $C_Q(X_1^i, Y_2^j)$ is produced by Q, where C_Q is the constructor of Q.

We transform Q into a query Q_{V_1} by removing $PE_2(Y_2)$ and $F_2(Y_2)$. Any variable $y \in Y_2$ which appears in the constructor of the new query is handled as a constant. For each binding X_1^i of the set of variables X_1, Q_{V_1} produces a set of tuples $C_Q(X_1^i, Y_2)$.

[2]Explicitly defined OIDs are introduced in Chapter 4.

Let $Q'_{V_1}(db_1, X_1, Y_1, Y_2)$ be the query that is produced when applying the rewriting algorithm on Q_{V_1}. Consider the following query:

Q':
WHERE
$PE_2(db_2, Y_2)$, $F_2(Y_2)$
CONSTRUCT
$Q'_{V_1}(db_1, X_1, Y_1, Y_2)$

The above query produces a set of tuples $C_Q(X_1^i, Y_2^j)$ for each pair X_1^i, Y_2^j of bindings of the variable sets X_1 and Y_2, respectively. So, we conclude that $Q' = Q$.

The situation is similar if we assume that PE_2 is defined over a view $V_2(db_2, X_2)$, where $X_2 \cap X_1 \cap Y = \emptyset$. Following the procedure described above, we produce the following query:

Q':
WHERE
$PE_2(V_2, Y_2)$, $F_2(Y_2)$
CONSTRUCT
$Q'_{V_1}(db_1, X_1, Y_1, Y_2)$

Then, we apply the rewriting algorithm to Q', and produce the query

$$Q''(db_1, db_2, X_1, X_2, Y_1, Y_2) = Q'(db_1, V_2, X_1, Y_1, Y_2) = Q(V_1, V_2, Y_1, Y_2).$$

The nesting of the generated query Q'' can be eliminated. Assume that Q'_{V_1} is composed of the sub-queries $Q'_{V_1,1}, ..., Q'_{V_1,q}$ and Q'' is composed of the sub-queries $Q''_1, ..., Q''_r$. The CONSTRUCT clause of each Q''_i consists of the union of all the $Q'_{V_1,j}$ sub-queries. So each Q''_i can be split into q sub-queries. The j^{th} such sub-query contains the patterns and filters of the WHERE clause of both Q''_i and $Q'_{V_1,j}$, as well as the constructor of $Q'_{V_1,j}$. Consequently, the resulting query consists of the union of $q \times r$ sub-queries. Each sub-query corresponds to a certain pair of matchings (M_1, M_2), where M_1 is matching between PE_1 and V_1's constructor, while M_2 is a matching between PE_2 and V_2's constructor.

The above solution can be generalized for user queries that contain more than two pattern expressions. Pattern expressions can also be defined over variables. We do not provide a solution for this case.

3.6 Handling variable equality in the user query

In the previous sections, we assumed that no equality conditions between variables exist in the user query. Equality can be expressed either by including equality conditions in the filter part, e.g., $\$x = \$y$, or by placing the same variable in different positions in the pattern part of a query. If the same variable appears more than once in the pattern part of the user query, we can use additional variables and declare their equality by filters. For instance, consider the following pattern expression, which matches people whose name is identical to their surname:

```
WHERE
    <view>
       <person>
          <name> $y </>
          <surname> $y </>
       </>
    </> IN V
```

This pattern expression can be rewritten as follows:

```
WHERE
    <view>
       <person>
          <name> $y </>
          <surname> $y' </>
       </>
    </> IN V, $y'=$y
```

Let v_1 and v_2 be two variables in a user query Q which is defined over a set of views $\{V_1, ...V_q\}$. Assume that a filter $v_1 = v_2$ appears in the filter part of Q. The variables v_1 may appear either in the same or in different pattern expressions of Q.

Let Q' be the rewritten query that is produced, when the filter $v_1 = v_2$ is omitted. As shown in the previous paragraph, Q' consists of the union of several sub-queries, and each such sub-query corresponds to a set M of matchings between the pattern expressions of Q and the corresponding views. Let Q'_M be the sub-query that corresponds to M. M represents a set of bindings of the variables that appear in Q, which produce a certain set of tuples, when the filter $v_1 = v_2$ is omitted. When the above filter is included, Q produces a subset of these tuples. In order to produce the same subset of tuples with the rewritten query, we should also include the appropriate filter in the filter part of Q'_M. The filter $v_1 = v_2$ can be directly included in the filter part of Q'_M, only if both v_1 and v_2 have not been substituted by the rewriting algorithm.

Assume that only v_1 has been substituted by a component $C_i(X_i)$ of the constructor of a view V_i. If $C_i(X_i)$ is a constant c, then we just add the filter $v_2 = c$ to Q'_M. If $C_i(X_i)$ is a constructor segment, which is not a variable, the filter $v_1 = v_2$ always returns false for the corresponding bindings of v_1 and v_2. The reason is that v_1 can only be bound to the OID of a source element, and this OID cannot be equal to the OID of an element produced by V_i. In this case, the binding M is rejected, i.e., the sub-query Q'_M is removed. Finally, if $C_i(X_i)$ is a variable $x \in X_i$, special care is required. The fact that x is bound to an OID does not indicate that v_1 is bound to the same OID. To be more precise, v_1 is bound to the OID of an element that V_i produces. In this case the filter $v_1 = v_2$ returns false, even if the equality $v_2 = x$ is true. However, x can also be bound to a constant. In order to handle this case, we introduce the external function *atomic(v)* which returns true if and only if v is bound to an atomic value. The filter that we add to Q'_M is

atomic(v) AND $v_1 = v_2$

Assume now that both v_1 and v_2 have been substituted by $C_i(X_i)$ and $C_j(X_j)$, respectively, where $C_i(X_i)$ is a component of V_i's constructor, and $C_j(X_j)$ is a component of V_j's constructor. The following six cases can be identified.

Case 1: $C_i(X_i) \equiv C_j(X_j) \equiv c$, where c is a constant. No filter is added to Q'_M, since $v_1 = v_2$ always evaluates to true for the set of bindings that correspond to M.

Case 2: $C_i(X_i)$ is a constant c, and $C_j(X_j)$ is a variable $x \in X_j$. The filter $x = c$ is added to Q'_M. The case where $C_i(X_i)$ is a variable and $C_j(X_j)$ is a constant is symmetric.

Case 3: $C_i(X_i) \equiv f(x_1, ..., x_s)$, $C_j(X_j) \equiv f(x'_1, ..., x'_s)$, and $i = j$, i.e., both the components are Skolem function declarations of the same Skolem function, and appear in the same view. The filters $x_1 = x'_1, ..., x_s = x'_s$ are added to Q'_M.

Case 4: Both $C_i(X_i)$ and $C_j(X_j)$ are constructor segments of the same view, i.e., $i = j$, and they are not variables. Moreover, the OID definitions that appear as arguments in the corresponding matches of M are $id_i \equiv f(x_1, ..., x_s)$ and $id_j \equiv f(x'_1, ..., x'_s)$, respectively. As discussed in Subsection 3.2.2, these OID definitions represent the actual bindings of v_1 and v_2. Thus, the filters $x_1 = x'_1, ..., x_n = x'_n$ are added to Q'_M.

Case 5: $C_i(X_i) \equiv x$ and $C_j(X_j) \equiv x'$, where x and x' are variables. We do not know whether x and x' are bound to atomic values or to OIDs. Consequently, we do not know whether the equality between v_1 and v_2 involves constants or OIDs. For this reason, we use again the external function $atomic()$. If $id_i \equiv f(x_1, ..., x_s)$, $id_j \equiv f(x'_1, ..., x'_s)$ and $i = j$, where id_i and id_j are defined as in Case 4, we add the filter

$$(atomic(c_1) \text{ AND } x = x') \text{ OR } (\text{NOT}(atomic(c_1)) \text{ AND } x_1 = y_1, ..., x_s = y_s)$$

Otherwise, since v_1 and v_2 cannot be bound to the same OID, we add the filter

$$atomic(c_1) \text{ AND } x = x'$$

Case 6: None of the previous cases is valid, e.g., $C_i(X_i)$ is a constant and $C_j(X_j)$ is a Skolem function. In this case, the filter $v_1 = v_2$ evaluates to false for every binding that corresponds to M. The sub-query Q'_M is removed.

Example 3.2 *We consider the following pair of view and user query:*

```
V: function View(){
    WHERE
      <a>
         <a1> $x1 </>
         <a2> $x2 </>
      </> IN ``source.xml''
    CONSTRUCT
       <b>
          <b1> $x1 </>
          <b2> $x2 </>
       </>
   }

Q: WHERE
     <view>
       <b>
         <b1> $y1 </>
         <b2> $y2 </>
       </>
     </> IN View(), $y1=$y2
   CONSTRUCT
       <result> $y1 </>
```

The nodes $y1 *and* $y2 *of the pattern tree of* Q *match the nodes* $x1 *and* $x2*, respectively, of the constructor tree of* V *and are required to be equal. The matching corresponds to Case 5. The variables can only be bound to string values, since the OIDs of* b1 *and* b2 *are always different. The rewritten query is presented below:*

```
Q':
   WHERE
      <a>
         <a1> $x1 </>
         <a2> $x2 </>
      </> IN ``source.xml'',
      atomic($x1) AND $x1=$x2,
    CONSTRUCT
<result> $x1 </>
```

It is clear that filters that involve inequality between variables of the user query are handled similarly.

Chapter 4

Introducing explicit OID definitions

The rewriting algorithm that we presented in the previous chapter assumes that no explicitly defined OIDs exist in the constructor of the view. In this chapter, we show how explicit OID definitions can be handled without changing the core of Algorithm 3.2. We assume that the same Skolem function does not define the OID of more than one element. In other words, element merging is not handled.

Explicit OID definitions by means of Skolem functions are used to control how the result of a query is grouped. For instance, consider the view query:

```
function Classes(){
   WHERE
        <couple>
            <boy> $b </>
            <girl> $g </>
            <age> $a </>
        </> IN ``couples.xml''
   CONSTRUCT
        <class ID = f($a)>
            <boy> $b </>
            <girl> $g </>
        </>
}
```

This query matches couples of boys and girls and groups them by their age. Each couple contains a boy and a girl of the same age. Each *class* element of the resulting document contains all the boys and girls of a certain age. This means that the structure

of the XML document that the query produces does not strictly follow the structure of the query's constructor.

4.1 Splitting pattern trees

Consider the following user query, which is defined over the view *Classes*:

```
Q: WHERE
        <view>
            <class>
                <boy> $x </>
                <girl> $y </>
            </>
       </> IN Classes()
   CONSTRUCT
        <couple>
            <boy> $x </>
            <girl> $y </>
        </>
}
```

The above query creates couples between boys and girls from the set of children that appear in each class. All the possible combinations of couples between boys and girls of the same age is produced. If we apply Algorithm 3.2 to the above view and user query the resulting rewritten query is:

```
Q': WHERE
         <couple>
             <boy> $b </>
             <girl> $g </>
             <age> $a </>
         </> IN ``couples.xml''
    CONSTRUCT
        <couple>
            <boy> $b </>
            <girl> $g </>
        </>
}
```

It is clear that the above query does not produce the same result. The problem originates from the fact that a certain *couple* element in the view can contain more than one *boy*

and *girl* element. Unfortunately, the matching algorithm considers that the structure of each *couple* strictly follows the structure of the constructor of the view, which involves a single *boy* and a single *girl* element.

However, if we split the pattern tree of Q so that the *boy* and the *girl* node can match the constructor tree independently, the problem is resolved. We transform Q as follows:

```
Q: WHERE
          <view>
              <class ID=$i1>
                 <boy> $x </>
              </>
          </> IN Classes(),
          <view>
              <class ID=$i2>
                 <girl> $y </>
              </>
          </> IN Classes(), $i1 = $i2
     CONSTRUCT
         <couple>
            <boy> $x </>
            <girl> $y </>
         </>
}
```

The transformed query is identical to the original one, since the filter $\$i1 = \$i2$ ensures that the two pattern expressions match *boy* and *girl* elements that belong to the same class. These two pattern expressions are not affected by the grouping under the *class* element of the view, since they can match the corresponding *boy* and *girl* elements no matter if these elements appear under the same or a different *class* element. Therefore, we can apply the rewriting algorithm independently for the two pattern expressions as described in Section 3.5 and receive a rewritten query which produces the correct output:

```
Q': WHERE
          <couple>
              <boy> $b </>
              <girl> $g </>
              <age> $a </>
          </> IN ``couples.xml'',
          <couple>
              <boy> $b' </>
              <girl> $g' </>
```

```
            <age> $a' </>
        </> IN ``couples.xml'',
        $a = $a'
    CONSTRUCT
        <couple>
            <boy> $b </>
            <girl> $g' </>
        </>
}
```

The filter $\$a = \$a'$ results from the filter $\$i1 = \$i2$, since $\$i1$ is matched to $f(\$a)$ and $\$i2$ is matched to $f(\$a')$.

In general, the pattern tree that corresponds to a user query is split into a set of ordered lists, whose number is equal to the number of the leaf nodes. Each ordered list consists of the nodes of a different path of the tree. The first node in each list is the root node, while the last node in the list is a leaf node of the tree. Before splitting the tree, all the nodes are assigned a variable as their OID attribute. So, the OID of nodes in different lists that correspond to the same tree node is represented by a common variable.

4.2 Matching grouped contents

When explicit OID definitions appear in the constructor of the view query, the *substitute()* function in Line 35 of Algorithm 3.2 should be adjusted to handle grouping.

Consider the following query:

```
Q: WHERE
        <view>
            <class> $c </>
        </> IN Classes(),
        <teacher> $t </> in ``teachers.xml''
   CONSTRUCT
        <class>
           <teacher> $t </>
           $c
        </>
```

This query assigns teachers to classes. All the possible combinations between teachers and classes are produced. If we apply the matching algorithm to the above query, c is

matched to the collection of the nodes that represent the *boy* and *girl* element of the view's constructor. These elements are grouped under the *class* element by the Skolem function that defines its OID. According to the rewriting algorithm, c is substituted by the constructor segment that corresponds to the matched nodes. Since the parent of these nodes does not participate in the match, the grouping cannot be expressed by a Skolem function.

In general, a match between a variable in the pattern tree of a user query and a node or a collection of nodes in the constructor tree of a view query needs a different manipulation, when the third argument of the corresponding triple involves a Skolem function of an explicit OID definition. More precisely, the variable cannot just be replaced by the constructor segment that the matched nodes represent, since the grouping information should also be included.

We explain how the problem is resolved. A grouping performed by means of a Skolem function can also be derived by using a query block. Consider the following query:

WHERE $P(x_1, ..., x_n)$
CONSTRUCT
...
<e ID = $f(x_k, ..., x_{k+m})$>
$c(x_1, ..., x_n)$
</>
...

The grouping that the Skolem function f performs can be expressed by a query block as follows:

WHERE $P(x_1, ..., x_n)$
CONSTRUCT
...
<e ID = $f(x_k, ..., x_{k+m})$>
{
WHERE $P(x'_1, ..., x'_n)$, $x'_k = x_k, ..., x'_k = x_{k+m}$
$c(x'_1, ..., x'_n)$
}
</>

...

where $X' = \{x'_1, ..., x'_n\}$ is produced after renaming the set of variables $X = \{x_1, ..., x_n\}$, such as $X' \cap X = \emptyset$.

The grouping is expressed by means of a nested query, which scans the input document and places the data that correspond to the same values of x_k under the same e element. Assume that the above query defines a view and consider a user query that is applied over this view. If now, there is a match between a variable of the user query and the collection of nodes that represent the constructor segment $c(x_1, ..., x_n)$, the variable is substituted by the sub-query block:

{
WHERE $P(x'_1, ..., x'_n)$, $x'_k = x_k, ..., x'_k = x_{k+m}$
$c(x'_1, ..., x'_n)$
}

The parameters $\{x_k, ..., x_{k+m}\}$ of the Skolem function are derived from the third argument of the triple that represents the match.

If the rewritten query involves several such substitutions, the resulting sub-query blocks are independent, i.e., there is no nesting between them. Therefore, the same renaming X' can be applied to all the sub-queries. Since the pattern part $P(x'_1, ..., x'_n)$ is common in all these sub-queries, it can be replaced by a single $P(x'_1, ..., x'_n)$ in the WHERE clause of the main query. Before this replacement, we explicitly define the OIDs in all the element tags of the query's constructor in order to ensure that the mappings that correspond to $P(x'_1, ..., x'_n)$ only affect data that participate in groupings.

In our example, after following the procedure presented above, the following rewritten query is produced:

```
Q': WHERE
        <couple>
            <boy> $b </>
```

```
            <girl> $g </>
            <age> $a </>
        </> IN ``couples.xml'',
        <teacher> $t </> in ``teachers.xml''
    CONSTRUCT
        <class>
           <teacher> $t </>
           {
            WHERE
                <couple ID = $i1'>
                    <boy ID = $i2'> $b' </>
                    <girl ID = $i3'> $g' </>
                    <age ID = $i4'> $a' </>
                </> IN ``couples.xml'',
                $a'=$a
            CONSTRUCT
                    <boy ID = g1($i1',$i2',$i3',$i4')> $b' </>
                    <girl ID = g2($i1',$i2',$i3',$i4')> $g' </>
            }
        </>
}
```

We can move the pattern part of the sub-query to the outermost level, after including all the implicit OID definitions:

```
Q': WHERE
        <couple ID = $i1>
            <boy ID = $i2> $b </>
            <girl ID = $i3> $g </>
            <age ID = $i4> $a </>
        </> IN ``couples.xml'',
        <teacher ID = $i5> $t </> in ``teachers.xml'',
        <couple ID = $i1'>
            <boy ID = $i2'> $b' </>
            <girl ID = $i3'> $g' </>
            <age ID = $i4'> $a' </>
        </> IN ``couples.xml'',
    CONSTRUCT
        <class ID = h1($i1,$i2,$i3,$i4,$i5)>
           <teacher ID = h2($i1,$i2,$i3,$i4,$i5)> $t </>
           {
            WHERE $a'=$a
            CONSTRUCT
                    <boy ID = g1($i1',$i2',$i3',$i4')> $b' </>
                    <girl ID = g2($i1',$i2',$i3',$i4')> $g' </>
            }
        </>
}
```

Chapter 5

Conclusions and future work

The problem addressed by this thesis is query composition by means of query rewriting in the context of XML-QL. More precisely, we studied how a user query defined over one or more views can be rewritten so that the rewritten query only refers to source data. User queries and views are expressed as XML-QL queries.

We presented a simple rewriting algorithm which is based on a matching procedure between the pattern part of the user query and the constructor part of the view query. The matching procedure results in sets of matchings which involve components of the above parts of the view and the user query. Matchings represent the bindings of the user-query variables when the user query is applied over the view. These bindings can be derived directly from the bindings of the view-query variables, without materializing the view. In order to facilitate the matching procedure, pattern expressions and constructors are represented by tree structures. The XML-QL data model is object based, i.e., XML elements are assigned a unique OID. In order to produce a correct rewriting, the elements in the constructor of the view query should be assigned an OID by means of Skolem functions. We showed how OIDs are derived when they are not explicitly defined.

The rewriting algorithm does not handle multiple pattern expressions and equality conditions between variables in the user query, explicit Skolem function definitions and

nested queries in the constructor of the view query, as well as regular path expressions. We showed that multiple pattern expressions can be handled by applying the algorithm in multiple steps. Each step performs the rewriting for a different pattern expression. We also demonstrated how the algorithm is enhanced to handle equality conditions between user-query variables. Finally, we presented how grouping introduced by explicit Skolem function definitions can be addressed without changing the algorithm. More specifically, we proposed a technique of splitting the pattern expressions of the user query, so that the produced pattern expressions are not affected by the grouping. We also identified cases in which a grouping performed by a Skolem function should be expressed by means of a nested query in the rewritten query.

This thesis did not present a complete solution for explicit Skolem function definitions. We assumed that the same Skolem function cannot appear in more than one element tag, so element merging is not allowed. We are working on how constructor trees should be extended in order to represent element merging. Nested queries in the constructor of the view query could also be handled by enhancing the properties of constructor trees.

In order to simplify the matching procedure, we did not study regular path expressions in the user query. The problem is similar to matching an XML-QL pattern with regular path expressions to an XML document. Techniques used by existing XML-QL query engines [Fer99] can be adapted to handle this problem. Regular path expressions in the view query cannot be handled, when the OIDs in the constructor of the query are not explicitly defined. In this case, a rewritten query cannot be always produced. This also means that XML-QL is not closed under composition. In general, when a query language involves regular path expressions and the underlying model is object based, it is not closed under composition [BLP+98].

We considered that the schema of the source data is unknown. Information about the schema of the source data, e.g., by means of DTDs or XML Schema, can change the situation in some special cases. For instance, regular path expressions in the pattern part

of a query can be eliminated if the schema strictly restricts the nesting depth of the data. Future work will concentrate on how the schema of the data can be exploited in order to overcome the above limitations and produce optimized rewritten queries. Previous work on this direction has been presented in the context of XMAS [LPVV99, PV99b].

Our study was restricted in the unordered data model. Order makes the problem harder. It is a subject of future work the investigation of under which conditions and how rewriting can be performed when order is considered. Aggregation is another issue not addressed by this thesis. Relevant work on relational [Kim81, GW87, Mur92] and object-oriented models [CM95] can be extended to handle aggregated views in the context of XML.

Appendix A

XML-QL Grammar

XML-QL grammar for views and user queries

```
              XML-QL ::= (Function | Query) <EOF>
            Function ::= 'FUNCTION' <FUN-ID> '(' ')' '{'
                             Query
                         '}'
             Element ::= StartTag  Contents  EndTag
            StartTag ::= '<'(<ID>|<VAR>) ('ID' '=' SkolemID)? Attribute* '>'
            SkolemID ::= <ID> '(' <VAR> (',' <VAR>)* ')'
           Attribute ::= <ID> '='  ('"' <STRING> '"'  |  <VAR> )
              EndTag ::= '<' / <ID>? '>'
             Literal ::= <STRING>
               Query ::= Where  Construct ('{' Query '}')*
               Where ::= 'WHERE' Condition (',' Condition)*
           Construct ::= 'CONSTRUCT' Contents
            Contents ::= (<VAR> | Literal | Element | Query)+
           Condition ::= Pattern BindingAs*  'IN' DataSource  |  Predicate
             Pattern ::= StartTagPattern Pattern* EndTag
    StartTagPattern ::= '<' RegularExpression Attribute* '>'
  RegularExpression ::= RegularExpression '*' |
                        RegularExpression '+' |
                        RegularExpression '.' RegularExpression |
                        RegularExpression '|' RegularExpression |
                        (<VAR> | <ID> | <UNDER>)
          BindingAs ::= 'ELEMENT_AS' <VAR>  | 'CONTENT_AS' <VAR>
          Predicate ::= Predicate <OR> Predicate |
                        Predicate <AND> Predicate |
                        <NOT> Predicate |
                        '(' Predicate ')' |
                        Expression OpRel Expression
         Expression ::= <VAR> | <CONSTANT>
              OpRel ::= '<' | '<=' | '>' | '>=' | '=' | '!='
         DataSource ::= <VAR> | <URI> | <FUN-ID> '(' ')'
```

Bibliography

[Abi99] S. Abideboul. On Views and XML. In *PODS '99. Proceedings of the eighteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 1–9. ACM press, 1999. Invited talk.

[AGM+97] S. Abiteboul, R. Goldman, J. McHugh, V. Vassalos, and Y. Zhuge. Views for semistructured data. In *Workshop on Management of Semistructured Data*, Tucson, Arizona, 1997.

[Bar99] C. Baru. XViews: XML views of relational schemas. In *Proceedings of the 10th International Workshop on Database and Expert Systems Applications*, pages 700–705, Florence, Italy, September 1999.

[BDHS96] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):505–516, 1996.

[BFS00] Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB Journal*, 9(1):76–110, 2000.

[BLP+98] C. Baru, B. Ludaescher, Y. Papakonstantinou, P. Velikhov, and V. Vianu. Features and Requirements for an XML View Definition Language: Lessons

from XML Information Mediation. Position paper in W3C's Query Language Workshop, 1998.

[BMR99] D. Beech, A. Malhotra, and M. Rys. A formal data model and algebra for XML. *Communication to the W3C*, September 1999.

[BT99] C. Beeri and Y. Tzaban. SAL: An algebra for semistructured data and XML. In *Informal Proceedings of Workshop on The Web and Databases, ACM SIGMOD*, June 1999.

[CCF+01] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML Query Language, June 2001. W3C Working Draft.

[CDSS98] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*, volume 27,2 of *ACM SIGMOD Record*, pages 177–188, New York, June 1–4 1998. ACM Press.

[CM93] Sophie Cluet and Guido Moerkotte. Nested queries in object bases. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages, Manhattan, New York City, USA, 30 August - 1 September 1993*, Workshops in Computing, pages 226–242. Springer, 1993.

[CM95] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical Report 95-6, RWTH Aachen, 1995.

[Con98] World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3.org/TR/REC-xml, February 1998.

[DFF+98] A. Deutch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. Submission to the World Wide Web Consortium, August 1998. http://www.w3.org/TR/NOTE-xml-ql.

[DFF+99] A. Deutch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the 8th International World Wide Web Conference*, pages 77–91. Elsevier Science, May 1999.

[Don00] Don Chamberlin and Daniela Florescu and Jonathan Robie. Quilt: an XML query language for heterogeneous data sources. In *Proceedings of WebDB*, Dallas, TX, May 2000.

[Feg98] L. Fegaras. Query unnesting in object-oriented databases. In Laura Haas and Ashutosh Tiwary, editors, *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data: June 1–4, 1998, Seattle, Washington, USA*, volume 27(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 49–60, New York, NY 10036, USA, 1998. ACM Press.

[Fer99] M. Fernandez. XML-QL : A Query Language for XML. User's Guide Version 0.9. http://www.research.att.com/ mff/xmlql/doc/, 1999.

[FSW+99] M. Fernandez, J. Siméon, P. Wadler, S. Cluet, A. Deutsch, D. Florescu, A. Levy, D. Maier, J. McHugh, J. Robie, D. Suciu, and J. Widom. XML query languages: Experiences and exemplars. http://www-db.research.belllabs.com/user/simeon/xquery.ps, 1999.

[FSW00] Mary Fernandez, Jerome Simeon, and Philip Wadler. A Data Model and Algebra for XML Query. *Technical Report, Number Unpublished manuscript*, 2000.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99)*, pages 25–30, Philadephia, Pensylvania, June 1999.

[GW87] Richard A. Ganski and Harry K. T. Wong. Optimization of nested SQL queries revisited. In Umeshwar Dayal and Irv Traiger, editors, *Proceedings of Association for Computing Machinery Special Interest Group on Management of Data 1987 annual conference, San Francisco, May 27–29, 1987*, pages 23–33, New York, NY 10036, USA, 1987. ACM Press.

[Kim81] W. Kim. On optimizing SQL-like nested queries. Technical Report RJ 3063, IBM, San Jose, CA, 1981.

[LD00] H. Liefke and S. Davidson. View Maintenance for Hierarchical Semistructured Data. In *International Conference on Data Warehousing and Knowledge Discovery*, 2000.

[LPVV99] B. Ludaescher, Y. Papakonstantinou, P. Velikhov, and V. Vianu. View definition and DTD Inference for XML. In *Workshop on Query Processing for Semi-Structured Data and Non-Standard Data Formats (in conjunction with ICDT'99)*, Jerusalem, Israel, 1999.

[MF00] D. Suciu M. Fernandez, W-C. Tan. SilkRoute: Trading between Relations and XML. In *Proceedings of the 9th International WWW Conference, Amsterdam*, May 2000.

[MSA+99] J.C. Mamou, C. Souza, S. Abideboul, V. Aguilera, A. Ailleret, B. Amann, S. Cluet, B. Hills, F. Hubert, A. Marian, L. Mignet, B. Tessier, A. M. Vercoustre, and T. Milo. XML repository and Active Views Demonstration.

In *Proceedings of 25th International Conference for Very Large Databases (VLDB'99)*, September 1999. Demonstration.

[Mur92] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *Proceedings of the 18th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Vancouver*, August 1992.

[MW99] J. McHugh and J. Widom. Query Optimization for XML. In *Proceedings of the Twenty-Fifth International Conference on Very Large Databases (VLDB'99)*, September 1999.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object Fusion in Mediator Systems. In T. M. Vijayaraman et al., editors, *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB'96)*, pages 413–424, Los Altos, CA 94022, USA, 1996. Morgan Kaufmann Publishers.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in Starburst. In Michael Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2–5, 1992*, volume 21(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 39–48, New York, NY 10036, USA, 1992. ACM Press.

[PV99a] Y. Papakonstantinou and V. Vassalos. Query Rewriting for Semistructured Data. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMod-99)*, volume 28,2 of *SIGMOD Record*, pages 455–466, New York, June 1–3 1999. ACM Press.

[PV99b] Y. Papakonstatinou and P. Velikhov. Enhancing Semistructured Data Mediators with Document Type Definitions. In *Proceedings of the 15th Inter-*

national Conference on Data Engineering. IEEE Computer Society Press, March 23–26 1999.

[RLS98] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). http://www.w3.org/TandS/QL/QL98/pp/xql.html, 1998.

[SLR98] D. Schach, J. Lapp, and J. Robie. Querying and transforming XML. In *Proceedings of The W3C Query Languages Workshop, Boston*, 3–4 December 1998. http://www.w3.org/TandS/QL/QL98/.