UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

VANESSA DE PAULA BRAGANHOLO

From XML to Relational View Updates: applying old solutions to solve a new problem

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Prof. Dr. Carlos Alberto Heuser
Advisor

Profa. Dra. Susan B. Davidson
Coadvisor

Porto Alegre, November 2004

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Braganholo, Vanessa de Paula

From XML to Relational View Updates: applying old solutions to solve a new problem / Vanessa de Paula Braganholo. – Porto Alegre: PPGC da UFRGS, 2004.

198 f.: il.

Thesis (doctor) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2004. Advisor: Carlos Alberto Heuser; Coadvisor: Susan B. Davidson.

1. Updates through views. 2. XML. 3. Relational databases. I. Heuser, Carlos Alberto. II. Davidson, Susan B. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice-Reitor: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitora Adjunta de Pós-Graduação: Prof^{a}. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

To my parents...

ACKNOWLEDGMENTS

To my parents José Carlos and Alice, who always supported me. It is impossible to list everything they did an still do for me. This work would not be a reality without them. To my sisters Karla, Tatiana, Larissa and Marcela, and to my beautiful niece Bruna, thanks for all your love.

All my heart to Leonardo Murta, who understood my long year at University of Pennsylvania, and who supported me in all possible ways. He was there in the worst snow storms; he was there when the apartment heat was off in a night of -15°C; he was there when a group of boys threw a snow ball in our backs. But more importantly, he was there to make me laugh, to give me love and to face the challenges with me. For all of this, I'll be thankful forever.

To my advisor Carlos Heuser, for all the excellent work and advices. Thanks for all the opportunities and for the guidance on this work. Guidance which started by the choice of the topic, which I loved since the first time he suggested, passes through the core of the solution, and continues until now with advices about my career. To my co-advisor Susan Davidson, who received me at University of Pennsylvania as if I were one of her own students. Thanks for everything she did for me, and for her great contribution to this work. Not less important, to my undergraduate advisor Clesio Saraiva dos Santos, who taught me to do and to like doing research.

To the UFRGS Database Group as a whole, for the suggestions and for the harmonious relationship. A special thanks to those who shared the 215 lab with me for all these years.

To my special friends Renata Galante and Carina Dorneles, who had enormous patience to hear me during my crises, who read and reviewed most of the papers resulting from this thesis, and who read and reviewed the thesis itself. Thanks for being more than colleagues. Thanks for being my real friends. I'll always remember our movies together, coffees, lunches, gym, laughs... Also, thanks to Mirella Moro for keeping me company in almost all of the conferences I've been to. Her sense of humor is fantastic, and she is a wonderful company. A special thanks to the "group of five" (you know who you are).

I would also like to thank Angelo Agra for implementing the *Relational View Updater* Module of the PATAXÓ System, which is the prototype implementation of the ideas of this thesis. Sérgio Mergen also contributed in the implementation of several methods of PATAXÓ. Another special thanks to Leonardo Murta, who helped me start using CVS to manage versions of the source code. Thanks for him and for Alexandre Dantas for helping me with complicated data structures in Java.

To Marcelo Arenas, Byron Choi, Carina Dorneles, Juliana Freire, Alon Halevy, Zack Ives, Eduardo Kroth, Cristiano Leivas, Leonardo Murta, Dan Suciu and Yifeng

Zheng, thanks for the help on getting data for the evaluation of query trees in real world XML views (chapter 8).

To Instituto de Informática and PPGC, for providing this excellent environment for learning and researching. To all my teachers in the 8 years I spent at UFRGS, thanks for teaching me the passion for academia. To all the workers of Instituto de Informática and PPGC, who were always there when I needed. Specially to Luís Otávio, Lourdes, Silvania, Elisiane, Angela, Eliane, Júnior, Jorge, Ismael, Sula, Margareth, Ida and Bea.

To CNPq, who has been financially supporting me since undergraduation, and to Capes, for the sandwich scholarship (BEX 1123/02-5).

Finaly, to all of those who have contributed to this work in a way or another.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

BCNF	Boyce Codd Normal Form
BNF	Backus-Naur Form
CSP	Constraint Satisfaction Problem
DAD	Document Access Definition
DBA	Database Administrator
DBLP	Digital Bibliography & Library Project
DTD	Document Type Definition
EBNF	Extended BNF
FD	Functional Dependency
PATAXÓ	Permitting updates on relational databases through XML views
QGM	Query Graph Model
RDBMS	Relational Database Management System
UML	Unified Modeling Language
W3C	World Wide Web Consortium
XFD	XML Functional Dependency
XMark	XML Benchmark Project
XML	eXtensible Markup Language
XQGM	XML Query Graph Model
XSL	Extensible Stylesheet Language
XSLT	XSL Transformations

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

XML has become an important medium for data exchange, and is frequently used as an interface to - i.e. a view of - a relational database. Although lots of work have been done on querying relational databases through XML views, the problem of updating relational databases through XML views has not received much attention. In this work, we give the first steps towards solving this problem.

Using query trees to capture the notions of selection, projection, nesting, grouping, and heterogeneous sets found throughout most XML query languages, we show how XML views expressed using query trees can be mapped to a set of corresponding relational views. Thus, we transform the problem of updating relational databases through XML views into a classical problem of updating relational databases through relational views.

We then show how updates on the XML view are mapped to updates on the corresponding relational views. Existing work on updating relational views can then be leveraged to determine whether or not the relational views are updatable with respect to the relational updates, and if so, to translate the updates to the underlying relational database.

Since query trees are a formal characterization of view definition queries, they are not well suited for end-users. We then investigate how a subset of XQuery can be used as a top level language, and show how query trees can be used as an intermediate representation of view definitions expressed in this subset.

Keywords: Updates through views, XML, Relational databases.

De Atualizações sobre Visões XML para Atualizações sobre Visões Relacionais: aplicando soluções antigas a um novo problema

RESUMO

XML vem se tornando um importante meio para intercâmbio de dados, e é frequentemente usada com uma interface para - isto é, uma visão de - um banco de dados relacional. Apesar de existirem muitos trabalhos que tratam de consultas a bancos de dados através de visões XML, o problema de atualização de bancos de dados relacionais através de visões XML não tem recebido muita atenção. Neste trabalho, apresentam-se os primeiros passos para a solução deste problema.

Usando *query trees* para capturar noções de seleção, projeção, aninhamento, agrupamento e conjuntos heterogêneos, presentes na maioria das linguagens de consulta XML, demonstra-se como visões XML expressas através de *query trees* podem ser mapeadas para um conjunto de visões relacionais correspondentes. Consequentemente, esta tese transforma o problema de atualização de bancos de dados relacionais através de visões XML em um problema clássico de atualização de bancos de dados através de visões relacionais.

A partir daí, este trabalho mostra como atualizações na visão XML são mapeadas para atualizações sobre as visões relacionais correspondentes. Trabalhos existentes em atualização de visões relacionais podem então ser aplicados para determinar se as visões são atualizáveis com relação àquelas atualizações relacionais, e em caso afirmativo, traduzir as atualizações para o banco de dados relacional.

Como *query trees* são uma caracterização formal de consultas de definição de visões, elas não são adequadas para usuários finais. Diante disso, esta tese investiga como um subconjunto de XQuery pode ser usado como uma linguagem de definição das visões, e como as *query trees* podem ser usadas como uma representação intermediária para consultas definidas nesse subconjunto.

Palavras-chave: Atualização através de visões, XML, Bancos de dados relacionais.

1 INTRODUCTION

1.1 Motivation and Goals

XML is frequently used as an interface to relational databases. In this scenario, XML documents (or views) are exported from relational databases and published, exchanged, or used as the internal representation in user applications. This fact has stimulated much research in exporting and querying relational data as XML views (FERNÁNDEZ et al., 2002; SHANMUGASUNDARAM et al., 2000, 2001; CHAUDHURI; KAUSHIK; NAUGHTON, 2003). However, the problem of updating a relational database through an XML view has not received as much attention: Given an update on an XML view of a relational database, how should it be translated to updates on the relational database? To the best of our knowledge, this question remains unanswered until now.

Since the problem of updates through relational views has been studied for more than 20 years by the database community, it would be good to use all that work to solve the new arising problem of updates though XML views. Specifically, is there a way to leverage existing work on updating through relational views to map XML view updates to the underlying relational database?

The main goal of this thesis is to try to answer both of the above questions. To do so, we present an approach that maps an XML view to a set of relational views, and use existing work on updates through relational views to map the updates to the underlying relational database.

In the relational case, attention has focused on updates through select-project-join views since they represent a common form of view that can be easily reasoned about using primary key and foreign key information. Similarly, we focus on a common form of XML views that allows nesting, composed attributes, heterogeneous sets and repeated elements.

1.2 Running Example and Overview

An example of an XML view is shown in Figure 1.1. In this XML view, *book* and *dvd* nodes are nested under the *products* node, and the *address* node composes attributes in a nested record format.

In this example, and in every example of this thesis, we use the database shown in Figure 1.2. Its schema is composed of six tables: *Vendor*, *Deposit*, *Book*, *DVD*, *SellBook* and *SellDVD*. Table *SellBook* establishes a relationship between tables *Vendor* and *Book*, registering prices of books sold by a given vendor. The table *SellDVD* plays the same role for dvds and vendors. A vendor has several deposits,

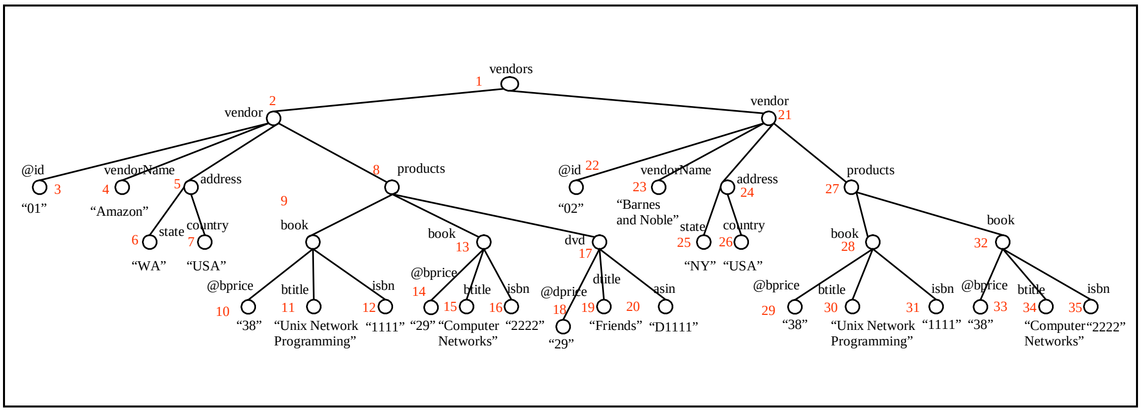


Figure 1.1: XML view containing vendors, books and dvds

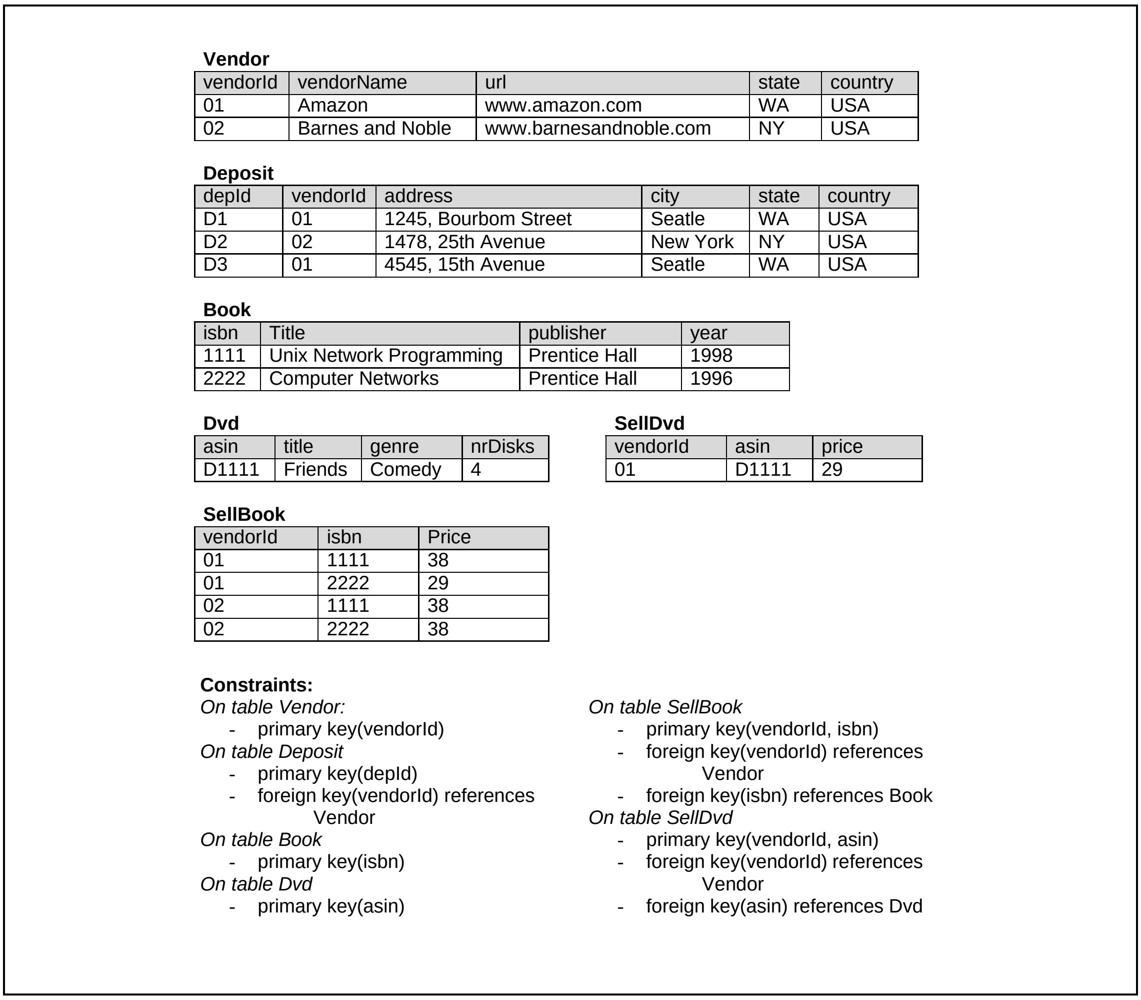
where he stores the goods he sells.

In the XML view of Figure 1.1, the data was extracted from tables *Vendor*, *Book*, *DVD*, *SellBook* and *SellDVD*. To specify how the XML view is constructed from the relational source, we represent XML view expressions as *query trees*. Query trees can be thought of as the intermediate representation of a query expressed by some high-level XML query language, and provide a language independent framework in which to study how to map updates to an underlying relational database. They are expressive enough to capture the XML views that we have encountered in practice, yet are simple to understand and manipulate. Their expressive power is equivalent to that of DB2 DAD files (CHENG; XU, 2000). Throughout the thesis, we will use the term "XML view" to mean those produced by query trees.

The strategy we adopt is to map an XML view to a set of underlying relational views. Similarly, we map an update against the XML view to a set of updates against the underlying relational views. It is then possible to use any existing technique on updates through relational views to both translate the updates to the underlying relational database and to answer the question of whether or not the XML view is updatable with respect to the update. An overview of our solution is shown in Figure 1.3.

In preliminary work (BRAGANHOLO; DAVIDSON; HEUSER, 2003a), we used the nested relational algebra (NRA) as the view definition language. In this approach, each XML view is mapped to a *single* relational view. However, NRA views are not capable of handling heterogeneity. Thus, the NRA is capable of representing the XML view of Figure 1.4 but not that of Figure 1.1. To make this clearer, in this context, heterogeneity means distinct DTD types. In the example of Figure 1.1, the node products has heterogeneous children – that is, it has children of types: *book* and *dvd*. In oppose to that, in Figure 1.4 there are no heterogeneous nodes.

Since views such as the one in Figure 1.1 are very common in practice, we have decided to adopt a more general view definition language – query trees. As mentioned before, a single XML view produced by a query tree can be mapped to a *set* of relational views. The cause of there being more than one underlying relational view for an XML view expressed by query trees is the presence of heterogeneous sets. For example, the XML view of Figure 1.1 is mapped to two corresponding relational views: one for vendors and books (*ViewBook*), and another one for vendors and DVDs (*ViewDVD*). We must then identify to which relational views an XML update should be mapped to.



Vendor

vendorId	vendorName	url	state	country
01	Amazon	www.amazon.com	WA	USA
02	Barnes and Noble	www.barnesandnoble.com	NY	USA

Deposit

depId	vendorId	address	city	state	country
D1	01	1245, Bourbom Street	Seatle	WA	USA
D2	02	1478, 25th Avenue	New York	NY	USA
D3	01	4545, 15th Avenue	Seatle	WA	USA

Book

isbn	Title	publisher	year
1111	Unix Network Programming	Prentice Hall	1998
2222	Computer Networks	Prentice Hall	1996

Dvd

asin	title	genre	nrDisks
D1111	Friends	Comedy	4

SellDvd

vendorId	asin	price
01	D1111	29

SellBook

vendorId	isbn	Price
01	1111	38
01	2222	29
02	1111	38
02	2222	38

Constraints:

On table Vendor:
- primary key(vendorId)

On table Deposit
- primary key(depId)
- foreign key(vendorId) references Vendor

On table Book
- primary key(isbn)

On table Dvd
- primary key(asin)

On table SellBook
- primary key(vendorId, isbn)
- foreign key(vendorId) references Vendor
- foreign key(isbn) references Book

On table SellDvd
- primary key(vendorId, asin)
- foreign key(vendorId) references Vendor
- foreign key(asin) references Dvd

Figure 1.2: Sample Database

As a concrete example, suppose we wish to insert a new book

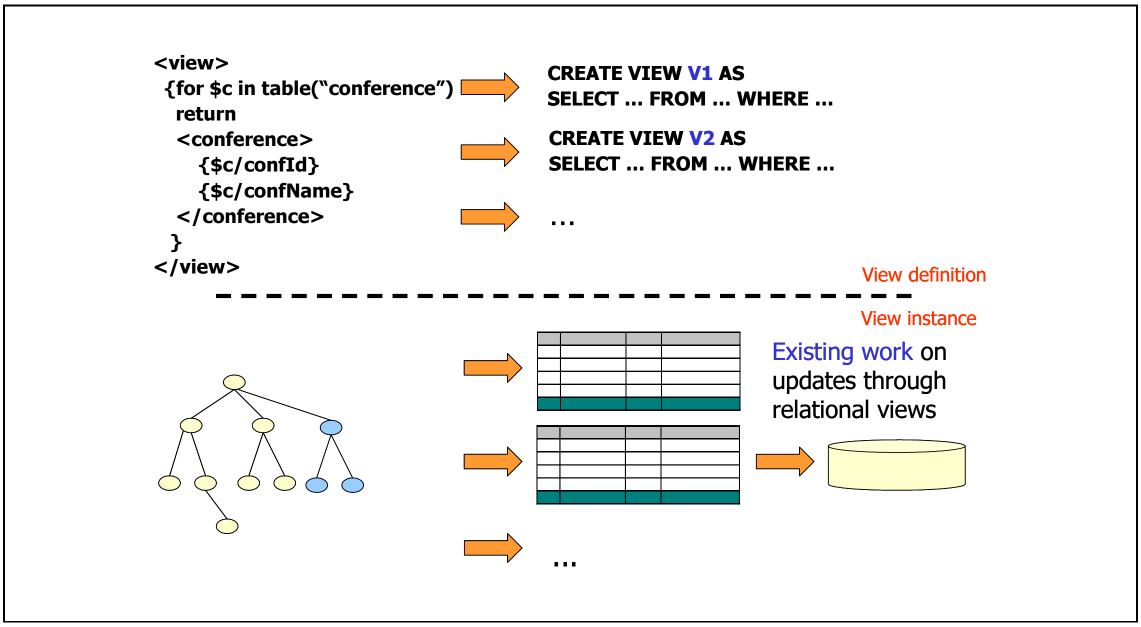
```
<book bprice="29">
  <btitle>Birding in North America</btitle>
  <isbn>5555</isbn>
</book>
```

at the point in the XML document specified by the following update path expression: `/vendors/vendor[@id="01"]/products` (node 8). Using the techniques of this thesis, this update to the XML view of Figure 1.1 would be mapped to the following SQL insert statement over *ViewBook*:

```
INSERT INTO VIEWBOOK (id, vendorName,  state, country, bprice, isbn, btitle)
VALUES ("01", "Amazon", "WA", "USA", 29, "5555", "Birding in North America");
```

Using existing relational techniques (in particular, that of (DAYAL; BERNSTEIN, 1982a)), we would then detect that the update is side-effect free and map the update on the relational view to a set of updates against the underlying relations.

We also address the problem of checking if an update respects the DTD of the XML view. As an example, the DTD of the XML view of Figure 1.4 is shown below:



```
<view>
 {for $c in table("conference")
   return
   <conference>
      {$c/confId}
      {$c/confName}
   </conference>
 }
</view>
```

```
CREATE VIEW V1 AS
SELECT ... FROM ... WHERE ...

CREATE VIEW V2 AS
SELECT ... FROM ... WHERE ...

...
```

Figure 1.3: Overview of the proposed solution

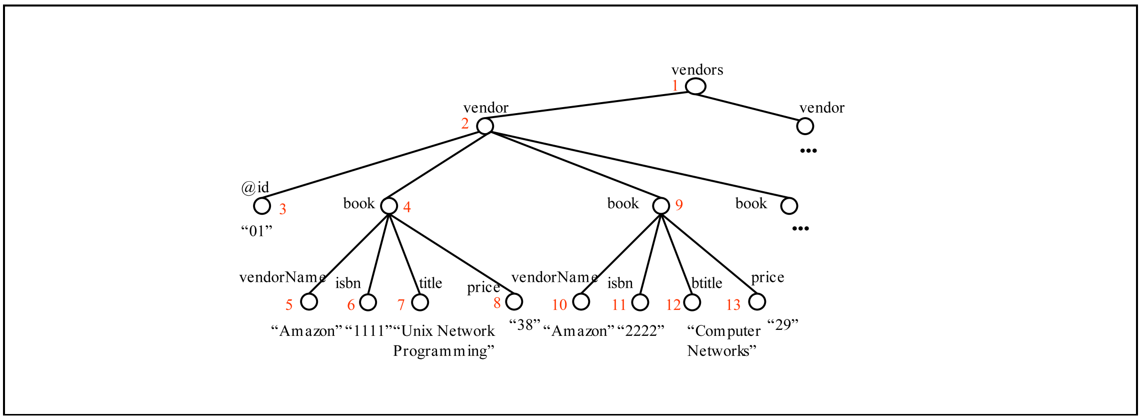


Figure 1.4: XML view showing books and vendors

```
<!ELEMENT vendors (vendor*)>
<!ELEMENT vendor (book*)>
<!ATTLIST vendor id CDATA #REQUIRED>
<!ELEMENT book (vendorName, isbn, title, price)>
<!ELEMENT vendorName (#PCDATA)>
<!ELEMENT isbn (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

If a user tries to insert the XML tree `<bike>Giant</bike>` using the update path `/vendors/vendor[@id="01"]`, we would determine that the update is not correct with respect to the DTD of the XML view, and would not attempt to map it to the underlying relational database.

Query trees are a formalism that is not well suited for end-users, since it has not a syntax users can use to specify XML views. In order to show how our approach can be used in practice, we have defined a subset of XQuery, which we call UXQuery, that users can use to specify XML views. We then map UXQuery view definitions into query trees and use the results of this thesis to translate the updates to relational views. We have implemented our approach in a system that we call PATAXÓ.

1.3 Contributions

In summary, the main contributions of this thesis are:

- A formalism to specify XML views over relational databases, which can be used as an intermediate representation of a top-level language.
- Mapping algorithms to map an XML view to a set of corresponding relational views, and also to map updates over the XML view to updates over the relational views. We thus transform the open problem of updating relational databases through XML views, into a well-studied problem, that of updating relational databases through relational views.
- An updatability study of XML views.
- A subset of XQuery, as an example of how a top-level language can be used in conjunction with query trees.

1.4 Organization of the text

The outline of this thesis is as follows:

- Chapters 2 and 3 present related work. In Chapter 2, we present work on updates through relational views, identifying the problems related to updates through views. In Chapter 3, we show approaches that build XML views over relational databases.
- Chapter 4 defines query trees, their abstract types, and the resulting XML view DTD, and shows how query trees are used to extract XML views from the relational database. It also presents a simple update language to update XML views and shows how to detect whether or not an update is correct with respect to the XML view DTD.
- Chapter 5 presents the algorithm for mapping an XML view to a set of underlying relational views, and proves its correctness. It also presents an algorithm for mapping insertions, modifications and deletions on XML views to updates on the underlying relational views, and proves its correctness.
- Chapter 6 presents an updatability study, based on the approach of (DAYAL; BERNSTEIN, 1982a). The study identifies classes of XML views for which updates are translatable without causing side-effects. The notion of side-effect free translations is defined by Dayal and Bernstein, and is explained in Chapter 2.
- On Chapter 7, we present a practical application of query trees. We define a subset of XQuery, which we call UXQuery, to define the XML views, and show how query trees can be used as its intermediate representation. We also show PATAXÓ, a prototype that implements the ideas of this thesis, using (DAYAL; BERNSTEIN, 1982a) to translate updates from the relational views to the underlying relational database.

- Chapter 8 discusses the expressive power of our language, and evaluates our technique with respect to existing proposals on extracting XML views of relational databases.
- We conclude in Chapter 9 with a summary of the thesis main contributions, a list of published papers, comparison with related work and a discussion of future work.

In Appendix A, we show how query trees can be easily extended to add grouping capabilities, and present an updatability study for these extended query trees. Appendix B presents an extension to query trees to support repeated node names. Finally, Appendix C shows three real world XML views we found, and how they can be expressed using query trees.

2 UPDATES THROUGH VIEWS

The problem of updates over XML is new and started to gain attention only recently. Despite of that, most of the work on updates over XML has focused on update languages for XML (ABITEBOUL et al., 1997; TATARINOV et al., 2001; BONIFATI et al., 2002; LAUX; MARTIN, 2000) and on schema conformance after updates (BOUCHOU; ALVES, 2003; PAPAKONSTANTINOU; VIANU, 2003). The problem of schema conformance consists on verifying if an updated XML document is still valid according to its original schema. The focus is on performing this verification without having to revalidate the entire document.

Both of these problems are also relevant when solving the problem of updating relational databases through XML views. In this specific field, there is also lots of research on extracting XML views over relational databases (we will study them on chapter 3). However, there is a lack of proposals in literature that deals with the core problem on updates through XML views: Given an XML view over a relational database, how are the updates over the view translated back to the underlying database?

To the best of our knowledge, the only work on literature that deals with updates through XML views is that of (WANG; MULCHANDANI; RUNDENSTEINER, 2003; WANG; RUNDENSTEINER, 2004). In (WANG; MULCHANDANI; RUNDENSTEINER, 2003), the XML views are XML documents stored in relational databases and reconstructed using XQuery. For this class of views, they prove that it is always possible to correctly translate the updates back to the database. However, they do not give details on how such translations are made. This approach differs from ours since we deal with XML views constructed over *legacy* databases. The approach in (WANG; RUNDENSTEINER, 2004) presents an extension to the notion of clean source of Dayal and Bernstein. They use this extended notion to study the updatability of XQuery views published over relational databases. Their results are analogous to the results of our WebDB 2003 paper (BRAGANHOLO; DAVIDSON; HEUSER, 2003a). It is important to state that both of these work do not deal with how the updates are translated to the underlying relational database.

Since there is this lack of related work on updates through XML views, we have studied work on updates through relational views. This helped us identifying the main problems in updates through views, and also helped us find the directions to solve the XML view update problem.

In the sections that follow, we overview work on updates through relational views, emphasizing problems and solutions.

V1 = $\sigma_{(\mathit{vendorId}=1)}(\mathit{Deposit})$
Deposits of vendor 1

V2 = $\sigma_{(\mathit{state}=\text{'}\mathit{WA}\text{'})}(\mathit{Vendor})$
Vendors of WA

V3 = $\pi_{(\mathit{vendorId},\ \mathit{vendorName})}(\mathit{Vendor})$
Projection of vendorId, vendorName on Vendor

V4 = $\pi_{(\mathit{city})}(\mathit{Deposit})$
Projection of city on Deposit

V5 = *Vendor * SellBook*
Natural join of Vendor and SellBook

V6 = $\sigma_{(\mathit{count}(\mathit{depId})\geq 2)}(\gamma_{(\mathit{vendorId})}(\mathit{Deposit}))$
Deposits grouped by vendorId and restricted to count(depId) ≥ 2

V7 = $\pi_{(\mathit{depId},\ \mathit{url})}(\mathit{Deposit} * \mathit{Vendor})$
Deposit joined with Vendor (on vendorId) and projected on depId and url

Figure 2.1: Sample views over database of Figure 1.2

2.1 Problem Statement

The problem of updates through views can be stated as follows: "Given a particular update on a particular view, what updates need to be applied to what underlying database tables in order to implement the original view update?" (DATE, 2000). Answering this question can be very complicated, especially because depending on the view structure, several problems may occur.

The main problems and implications of updates through views were enumerated by Furtado in (FURTADO; CASANOVA, 1985), and are listed below. For each of the identified points, we present an example based on the views of Figure 2.1, which were defined over the database of Figure 1.2[1]. The examples were adapted from (FURTADO; CASANOVA, 1985).

1. A constraint may be violated.

   Suppose the following insertion on V1:

   ```
   INSERT INTO V1 (depId, vendorId, address, city, state, country)
   VALUES ("D2", "01", "1478, 25th Avenue", "New York", "NY", "USA")
   ```

   The natural translation of this insertion would be to insert this tuple into the Deposit base table. However, this violates two constraints: (i) the functional dependency *DepId* → *VendorId*, since D2 is already assigned to Vendor 02 in the database; (ii) the primary key of table Deposit.

   Similarly, the translation of the deletion

[1] The view V6 on Figure 2.1 was defined using the extended algebra of (ULLMAN; WIDOM, 1997), where γ is the grouping operator.

```
DELETE FROM V2
WHERE vendorId="01" AND vendorName="Amazon" AND
url="www.amazon.com" AND state="WA" AND country="USA"
```

violates the constraint *Deposit[vendorId]* $\subseteq$ *Vendor[vendorId]*, because there are two deposits in table Deposit assigned to vendor Amazon. Similarly, this deletion violates the constraints *SellBook[vendorId]* $\subseteq$ *Vendor[vendorId]* and *SellDVD[vendorId]* $\subseteq$ *Vendor[vendorId]*.

2. A result that does not conform to the view definition may be produced.

   The translation of the insertion:

   ```
   INSERT INTO V2 (vendorId, vendorName, url, state, country)
   VALUES ("03", "XYZ", "www.xyz.com", "PA", "USA")
   ```

   would violate the view definition, since V2 is restricted to vendors whose state is "WA".

3. A result conforming to the view definition may be produced, but tuples lying outside the view may be involved.

   Consider the following modification:

   ```
   UPDATE VENDOR
   SET state = "WA"
   WHERE vendorId = "02"
   ```

   This example shows that updates made outside the view may affect the tuples of a given view. In this example, the update was made directly on the Vendor table, but it caused a new tuple ("02", "Barnes and Noble", "www.barnesandnoble. com", "WA", "USA") to appear on V2.

   Although this is one of the problems identified by (FURTADO; CASANOVA, 1985), it is not addressed in most of the work on updates through virtual relational views. This problem is more directly related to the maintenance of materialized views (AGRAWAL et al., 1997; SALEM et al., 2000; ZHUGE et al., 1995).

4. An effect different from that expected from the nature of the operation may be produced.

   The replace operation

   ```
   UPDATE V2
   SET state = "NY"
   WHERE vendorId="01" AND vendorName="Amazon"
   AND url="www.amazon.com" AND state="WA" AND country="USA"
   ```

   will cause the tuple to be deleted from V2, so the effect of the update (deletion) is not the one expected by the user (modification). Notice that the deletion from the view occurs because the modified tuple does not satisfy the view definition anymore. Despite of that, the tuple remains in the Vendor table.

5. It may be necessary to include undefined values (*null*s).

   An insertion over a view that projects out attributes of the base tables, such as V3, causes *null* values to be inserted on those attributes. As an example, the insertion

   ```
   INSERT INTO V3(vendorId, vendorName)
   VALUES ("03", "XYZ")
   ```

   would be translated to inserting ("03", "XYZ", *null*, *null*, *null*) into Vendors.

6. Attributes outside the view may be affected.

   Consider the following deletion:

   ```
   DELETE FROM V4
   WHERE city="Seattle"
   ```

   This would mean to delete all tuples which have "Seattle" in the city column of the Deposit table. In this case, information about deposits D1 and D3 would be completely lost.

7. Multiple effects on the stored relations may be produced.

   The translation of the replacement

   ```
   UPDATE V4
   SET city="Toronto"
   WHERE city="Seattle"
   ```

   would replace all tuples having "Seattle" as city in the table Deposit by another one containing "Toronto" in the city column. The implication here is that the update specification involves a single tuple from the view, but it causes several database tuples to be modified.

8. A multiple effect on the view may be visible.

   The modification

   ```
   UPDATE V5
   SET vendorName="Amazon.com"
   WHERE vendorId="01" AND vendorName="Amazon"
   AND state="WA" AND country="USA" AND isbn=1111 AND price=38
   ```

   makes that tuple ("01", "Amazon", "www.amazon.com", "WA", "USA") be replaced by ("01", "Amazon.COM", "www.amazon.com", "WA", "USA"). However, this translation would modify all tuples referring to the Amazon vendor in V5, and not only the desired tuple.

9. An entire group of tuples or none at all, may be changed.

   ```
   INSERT INTO V6 (depId, vendorId, address, city, state, country)
   VALUES ("D4", "02", "4444, 52nd Avenue", "New York", "NY", "USA")
   ```

This insertion would be translated as an insertion of a tuple in table Deposits. However, two tuples would be inserted in the view, and not only the desired one. This is because before the insertion, there was no tuple corresponding to vendor 02 in V6, because count(depId) of vendor 02 was equal to one (1). After the update, deposit D2 would also be inserted in the view, since now count(depId) of vendor 02 is greater or equal to two (2).

10. More than one translation may exist.

    Consider the following modification request over V7:

    ```
    UPDATE V7
    SET url="www.barnesandnoble.com"
    WHERE depId="D1" AND url="www.amazon.com"
    ```

    Since ("D1", x) is in the Deposit table, and (x, "www.amazon.com") and (y, "www.barnesandnoble.com") are in the Vendor table, there are two possible translations for this replacement. The first one is to replace ("D1", x) for ("D1", y) in Deposit, which associates D1 with the Barnes and Noble vendor. The second option is to replace (x, "www.amazon.com") by (x, "www.barnesandnoble.com") on table Vendor, which changes the URL of the Amazon vendor.

2.1.1 Minimality Requirements

Despite of taking into account the problems of the previous section, an update translator should also have a minimal effect on the base relations. Thinking on that, Keller (1985) has defined five rules that should be followed by update translators:

1. No database side effects. The only database tuples affected by the update translation should have keys that match their respective values in the tuples mentioned in the view update request. This is to solve problem 8 of previous section (*a multiple effect on the view may be visible*).

2. Only "one step" changes. Each database tuple is affected by at most one step of the translation for any single view update request. Specifically, a translation cannot replace an inserted tuple, or delete a replaced tuple, or replace a tuple twice in succession. This requirement serves to avoid problems in the view update translation, since the original tuple would have disappeared after the execution of the first update.

3. Minimal change: no unnecessary changes. Given two translations of an update request, if one is a proper subset of the other, then the minimal set of updates should always be used.

4. Minimal change: replacements cannot be simplified. Considering two (alternative) database replacement requests where both specify to replace the same tuple, a database replacement that does not involve changing the key is simpler than one where the key changes.

5. Minimal change: no delete-insert pairs. Candidate translations cannot include both deletions and insertions on any relation, since they may be converted into replacements.

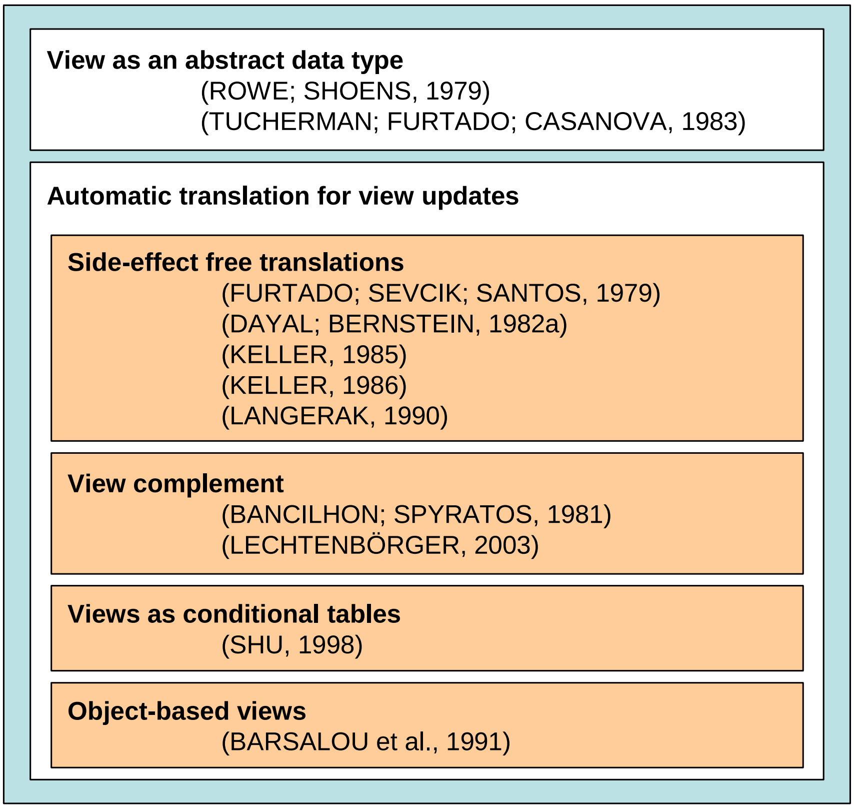

View as an abstract data type
(ROWE; SHOENS, 1979)
(TUCHERMAN; FURTADO; CASANOVA, 1983)
Automatic translation for view updates
Side-effect free translations
(FURTADO; SEVCIK; SANTOS, 1979)
(DAYAL; BERNSTEIN, 1982a)
(KELLER, 1985)
(KELLER, 1986)
(LANGERAK, 1990)
View complement
(BANCILHON; SPYRATOS, 1981)
(LECHTENBÖRGER, 2003)
Views as conditional tables
(SHU, 1998)
Object-based views
(BARSALOU et al., 1991)

Figure 2.2: Classification of work on updates through relational views

2.2 Updates through Relational Views

Although the problems of Section 2.1 were identified on relational views, they also hold for XML views. As a consequence, in order to be able to update relational databases through XML views, it is necessary to study how the above problems have been addressed in the literature of updates through relational views.

A large amount of work has been done on updates through relational views, and several different techniques have been proposed. These proposals can be grouped in two distinct categories. The first one treats a view as an abstract data type, and the second one tries to define automatic translators for view updates. Figure 2.2 presents the classification of the existing proposals into these two categories. The category of automatic translators is further divided into four classes of approaches, according to the correctness criteria they adopt for the translations. We summarize the existing approaches and classify them below:

View as an abstract data type: In this approach (ROWE; SHOENS, 1979; TUCHERMAN; FURTADO; CASANOVA, 1983), the DBA defines the view together with the updates it supports. The effect of updates on the base relations is explicitly defined. This means that the DBA must implement the code that will be executed every time an insertion, deletion or modification is issued against the view.

Automatic translations for view updates: Proposals that follow this approach try to define automatic translators for view updates. Some of them present algorithms for the update translations, while others just present guidelines and criteria to identify when a translation is correct.

There is an additional approach (MEDEIROS; TOMPA, 1985) that comprises both types of translators. The approach consists on an analysis algorithm that predicts whether the desired update really occurs in the view, and whether it causes side-effects. The inputs to the algorithm are the view definition, a set of constraints, as well as the update that is to be analyzed. The authors claim that the algorithm can be used both by general update translators, as well as by abstract data type approaches. Notice that the proposed algorithm does *not* automatically translate the update. In fact, the update translation is one of the algorithm's inputs. It only checks the effect of the update on the database and on the view.

As mentioned before, the proposals that follow the automatic translations approach can be further classified into four categories, depending on the criteria they adopt for correctness of the update translations. We now present these four categories and overview the approaches in each of them:

Side-effect free translations: An update translation is side-effect free as long as it corresponds exactly to the specified update and does not affect anything else in the view (KELLER, 1985). Finding side-effect free updates corresponds to solving problem 8 (*a multiple effect on the view may be visible*) of Section 2.1.

In (KELLER, 1985), Keller defines five criteria that the translations should respect in order to be correct and minimal, and present algorithm sketches that satisfy the five criteria. An exception is made to algorithms for views involving joins. For those, Keller allows side-effects to occur. In (KELLER, 1986), Keller uses the algorithms proposed in (KELLER, 1985) together with user (DBA) input to choose a view update translator. The DBA answers dialogs at view definition time to define a view update translator that better suits the semantics of the view.

Dayal and Bernstein (DAYAL; BERNSTEIN, 1982a) propose a translation mechanism that uses view graphs to decide if a given update translation is correct. The view graphs are constructed based on the syntax of the view definition and on the functional dependencies of base relations. Although Dayal and Bernstein use the term *exact translation* to denote correctness, this is equivalent to the notion of side-effect free translation of Keller (1985).

In (FURTADO; SEVCIK; SANTOS, 1979), a general approach for translation is discussed, depending on the relational algebra operations used in the view definition. General translations are presented for each combination of the relational operations. A more recent work is presented in (LANGERAK, 1990), but it does not consider views involving selections, which are very common in practice.

View complement: The view complement approach considers an update translation to be correct if it does not affect any part of the database that is outside the view (BANCILHON; SPYRATOS, 1981; LECHTENBÖRGER, 2003). This corresponds to solving problem 6 (*attributes outside the view may be affected*) of Section 2.1. The denomination "data outside the view" matches what Bancilhon and Spyratos in (1981) called the *view complement*. Years

Table 2.1: Check list regarding the problems of Section 2.1

	1	2	3	4	5	6	7	8	9	10
Furtado et al., 1979	√	√	×	√	×	×	×	√	√	×
Bancilhon et al., 1981	√	√	×	√	√	√	×	√	√	√
Dayal et al., 1982	√	√	×	√	×	×	×	√	√	×
Keller, 1985	√	√	×	√	×	×	×	P	√	×
Keller, 1986	√	√	×	√	×	×	×	P	√	×
Langerak, 1990	√	√	×	×	×	×	×	P	×	×
Barsalou et al., 1991	√	√	×	√	×	×	×	P	√	×
Shu, 1998	√	√	×	√	√	×	×	√	√	√
Lechtenbörger, 2003	√	√	×	√	√	√	×	√	√	√

Legend: √: solves the problem
×: does not address the problem
P: partially solves the problem
– : unknown

later, Cosmadakis and Papadimitriou (1984) proved that finding a view complement may be NP-complete even for very simple view definitions. A recent work (LECHTENBÖRGER, 2003) on view complements, however, states that it is not necessary to actually calculate the view complement in order to know if it was changed by the view update translation. It is enough to know if the view update can be undone. If so, then it is guaranteed that the view complement was not affected by the view update translation.

Views as conditional tables: A more recent technique consists in transforming a view update problem into a Constraint Satisfaction problem (CSP) (SHU, 1998). In this approach, views are represented as conditional tables and view updates are translated to a disjunction of CSPs. Each solution to the constraint satisfaction problem corresponds to a possible translation of the view update.

Object-based views: An extension of (KELLER, 1986) to deal with object-based views is proposed in (BARSALOU et al., 1991). In this work, Barsalou proposes algorithms for propagating updates in a hierarchical structure of objects. An implementation of this proposal is discussed in the Penguin Project (TAKAHASHI; KELLER, 1994; KELLER; WIEDERHOLD, 2001).

In Table 2.1 we analyze each of the existing proposals in literature to check if they solve the problems identified in Section 2.1. Each problem is identified by a number from 1 to 10 which corresponds to the numbers used in Section 2.1.

Based on Table 2.1, it is easy to see that the existing approaches do not address all the problems identified by Furtado and Casanova (1985). In fact, some of those problems are specifically connected to the notion of correctness in the existing proposals of automatic translators for view updates. With some exceptions, each author adopts some of those problems as the notion of correctness for update translations. An exception is the approach of (LANGERAK, 1990), which does not clearly define a correctness criterion. It presents translation algorithms for Project-Join views (they do not treat selections), and allow side-effects to occur both in insertions and in some types of modifications.

In our work, we are interested in automatically translating updates to the underlying base tables. We did not want to require the user to build a program to

Table 2.2: Comparison of work on updates through relational views

	algorithms for update translation	correctness criteria	correctness criteria for the algorithms	criteria of Section 2.1
Furtado et al., 1979	√	√	√	8
Bancilhon et al., 1981	×	√	×	6, 8
Dayal et al., 1982	√	√	√	8
Keller, 1985	√	√	×	8
Keller, 1986	√	√	×	8
Barsalou et al., 1991	√	√	×	8
Langerak, 1990	√	×	×	–
Shu, 1998	√	√	√	6
Lechtenbörger, 2003	×	√	×	6, 8

Legend: √: solves the problem
×: does not address the problem
– : does not apply

translate the updates for each XML view he specifies. Table 2.2 analyzes the existing approaches to check if they provide algorithms for update translations, and, if so, if they present the conditions under which the algorithms work (correctness criteria). In this table, we analyze only work on automatic translations.

Based on the above observations, in our work we could have taken three main paths in order to solve the problem of updates through XML views:

1. To define an automatic update translator based on one of the existing approaches for relational views;

2. To define a completely new update translator for XML views, together with the correctness criteria for the update translator (based on the problems of Section 2.1).

3. To transform the problem of updates through XML views into the problem of updates through relational views, so that all the work on updates through relational views could be applied;

We have chosen the third alternative mainly because it is more flexible. In this approach, one can use any update translator available in literature. Thus, we are not fixing a specific update translator, but we are creating conditions for existing ones to be applied in XML. In fact, this idea appeared before in literature in (DAYAL; BERNSTEIN, 1982b). In this work, Dayal and Bernstein map the database network model to the relational model, and then use their work on updates through relational views (DAYAL; BERNSTEIN, 1982a) to translate the updates and to check for correctness.

Also, we have avoided defining a fixed update translator to be able to deal with cases where the semantics of the view is not well suited for automatic translators. There is a classical example (KELLER, 1986; ABITEBOUL; HULL; VIANU, 1996) that illustrates this. It is a view that is formed by employees of a company who play in the company's soccer team (*play-soccer-team* = *yes*). This view has a special semantics regarding insertions and deletions: deleting an employee from the view means that the employee does not play in the soccer team anymore, but it *does not* mean that the employee was fired from the company. This deletion, as a consequence, could not be translated to a deletion on the employee table. Instead, it should be translated to modifying the attribute *play-soccer-team* to *no*. Since in our approach we are not specifying how the updates are translated to the underlying relational database, one can use any existing approach or, in the absence of one that comprises the semantics of that specific view, define a customized update translator to that view (by either using (KELLER, 1986) or the abstract data type approach).

Despite of being possible to use any of the existing work on updates through relational views to translate the updates, in our implementation (discussed in chapter 7) we have chosen one of them. Our choice was made among the approaches that present algorithms for update translations, namely (FURTADO; SEVCIK; SANTOS, 1979), (DAYAL; BERNSTEIN, 1982a), (KELLER, 1985), (BARSALOU et al., 1991), (LANGERAK, 1990) and (SHU, 1998). We discard (KELLER, 1986) and (BARSALOU et al., 1991) because they use user input to choose the update translator, while we are looking for a completely automatic process. Despite of that, at a first glance, one could argue that the approach of (BARSALOU et al., 1991) is a good option, since object-views are hierarchical in the same way as XML. It turns out that object views are a *subset* of the underlying database *structural model*. That is, given a relational database and its referential and integrity constraints, one obtains a model that Barsalou et al. calls *structural model*. A view object is then defined as a subset of this model, but no restructuring is allowed. This considerably limits the set of possible views one can construct, so we decided not to take this solution path.

Among the remaining options, only (DAYAL; BERNSTEIN, 1982a), (FURTADO; SEVCIK; SANTOS, 1979) and (SHU, 1998) fully present algorithms and conditions for the algorithms to produce correct translations (as shown in emphasized column of Table 2.2). From these three options, we have chosen the approach of Dayal and Bernstein because it provides general algorithms. Their algorithms comprise select-project-join views, and work for any combination of such algebra operators. As oppose to that, (FURTADO; SEVCIK; SANTOS, 1979) presents specific algorithms for each combination of the relational algebra operators. Although this is not a problem, this would require an additional analysis on the view structure, in order to identify which algorithm to apply to translate a given update. Besides, some of the conditions for the correctness of the translations are instance based conditions, while we would like to leave the reasoning at the schema level. As for the approach of (SHU, 1998), all the reasoning is done at instance level.

We would like to emphasize that we could have implemented our approach without choosing any update translator. In this way, the system would map an XML view to a set of relational views, and map updates over the XML view to updates over the relational views. The user could then use any translator he wanted. However, we have decided to include a translator in our implementation to better show the feasibility of our approach.

In the next section, we present the approach of Dayal and Bernstein in details.

2.2.1 The approach of Dayal and Bernstein

The relational views considered by Dayal and Bernstein (1982a) are select-project-join views, where the join conditions are specified as equalities. It is also possible to use selection conditions, comparing attributes with a constant value (c), as long as the comparison is an equality. Formally, we have: Let a relational view

$V(Z)$, with $Z = \{D_1, \ldots, D_n\}$ be defined as:

```
CREATE VIEW $V(D_1, \ldots, D_n)$
   AS SELECT $t_{i_1}.A_{i_1}, \ldots, t_{i_n}.A_{i_n}$
      FROM $R_1$ $t_1$, ..., $R_m$ $t_m$
      WHERE <qual>
```

where t_1, ..., t_m are aliases of relations R_1, ..., R_m, with $m \geq 1$. Each $t_i.A_{jk}$ (with $1 \leq i \leq m$ and $1 \leq k \leq n$) is an attribute of relation R_i, and $\{t_{i_1}, ..., t_{i_n}\} \subseteq \{t_1, ..., t_m\}$.

Also, the qualification <qual> contains only clauses of the form "$t.A = c$" or "$t.A = u.B$" (t and u may be the same). We say that the attribute A_{jk} *generates* the view attribute D_k, for each k in $[1, n]$. We call Rels(V) the set $\{R_1, \ldots, R_m\}$, which is the set of relations over which V is defined.

To be able to reason about the correctness of the translation algorithms (presented in Section 2.2.1.3), Dayal and Bernstein (1982a) use the structure of the view, as well as the functional dependencies of the source database schema, to build two auxiliary graphs: the *view trace graph* and the *view dependency graph*. We elaborate more on them in the sections that follow.

2.2.1.1 View Trace Graph

The structure of the view definition is captured by defining a labeled directed graph $G(V)$, called *View Trace Graph*, constructed as follows:

1. For every attribute A of each relation R_i occurring in the FROM clause, there is a node in the graph labeled "$R_i.A$";

2. For every view attribute D_i there is a node labeled $V.D_i$;

3. For each view attribute D_k there are arcs ($V.D_k$, $R_{i_k}.A_{i_{jk}}$) and ($R_{i_k}.A_{i_{jk}}$, $V.D_k$), if $A_{i_{jk}}$ generates D_k;

4. For every clause ($t_i.A = t_j.B$) in <qual> there are arcs ($R_i.A$, $R_j.B$) and ($R_j.B$, $R_i.A$);

5. For every clause ($t_i.A = c$) in <qual> introduce a node labeled c (called a constant node) and arcs (c, $R_i.A$) and ($R_i.A$, c).

If $G(V)$ has a path from node D to node $R_i.A$, then we say $R_i.A$ is *traceable* from V, and D is a *V-trace* for $R_i.A$.

Example 2.1 *As an example, consider the following relational view, defined over the database schema of Figure 1.2.*

```
CREATE VIEW VENDORBOOK (vendorId, vendorName, isbn, title, price)
  AS SELECT v.vendorId, v.vendorName, b.isbn, b.title, sb.price
     FROM Vendor v, SellBook sb, Book b
     WHERE v.vendorId = sb.vendorId AND b.isbn = sb.isbn
```

The view trace graph for this view is shown in Figure 2.3. In this figure, *VendorBook.isbn* is a *V-trace* for *Book.isbn* and *SellBook.isbn*.

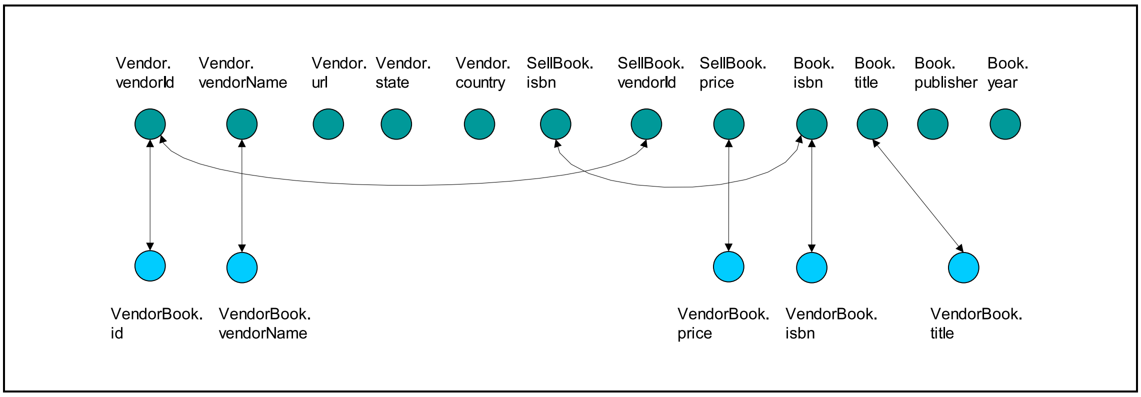


Figure 2.3: View trace graph for view *VendorBook*

2.2.1.2 View Dependency Graph

The *View Dependency Graph* ($F(V)$) is obtained by enriching $G(V)$ with the information provided by the functional dependencies in the database schema.

In the relational model, functional dependencies are not explicitly represented. The only information we have are the database schema and its constraints, such as foreign key constraints. However, assuming that the database is in the BCNF, we can infer the functional dependencies of a given table R, since in this normal form, all non-trivial dependencies in R are of type $\{A_1, ..., A_n\} \rightarrow B$, where B is an attribute of R and $\{A_1, ..., A_n\}$ is a superkey of R (ULLMAN; WIDOM, 1997). As a consequence, we can infer functional dependencies of type *primary key* → B, for each B in R that is not part of the primary key.

As an example, in the table *Vendor* we have the following functional dependencies:

$$\begin{array}{l} \textit{vendorId} \rightarrow \textit{vendorName} \\ \textit{vendorId} \rightarrow \textit{url} \\ \textit{vendorId} \rightarrow \textit{state} \\ \textit{vendorId} \rightarrow \textit{country} \end{array}$$

Notice that by the rules of BCNF, we can also have functional dependencies like *vendorId, vendorName* → *url*, since {*vendorId, vendorName*} is a superkey of Vendor. However, this functional dependency is not necessary. The reason for that will be explained after we show Dayal's definition of *paths*.

The view dependency graph is constructed as follows.

1. For every FD f: $B_1, \ldots, B_k \rightarrow_{R_i} A$, add the FD-node f and the arcs of Figure 2.4. If A and B are singletons, then for convenience we denote the FD f: $B \rightarrow_{R_i} A$ by a single arc ($R_i.B$, $R_i.A$).

2. If $R_i.A$ is traceable from a constant node c, then for every attribute B of every relation R_i in the FROM clause, draw arc ($R_j.B$, $R_i.A$).

As an example, Figure 2.5 shows the view dependency graph for the view *VendorBook* (example 2.1). To emphasize the differences between this graph and the view trace graph, we have chosen to show the added arcs in dashed lines. Notice

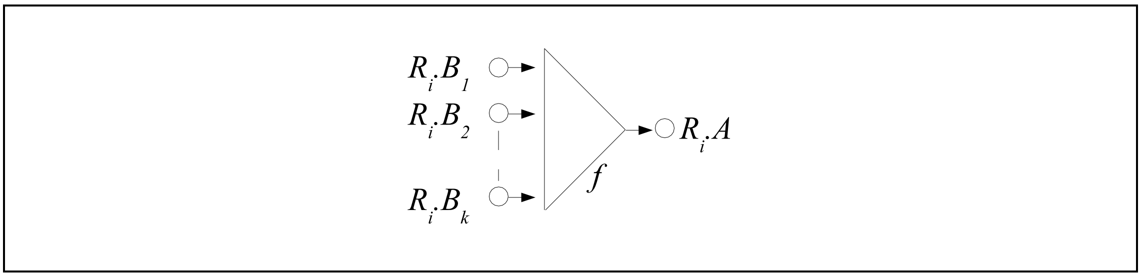


Figure 2.4: FD-node f corresponding to the FD f: $B_1, \ldots, B_k \to_{R_i} A$

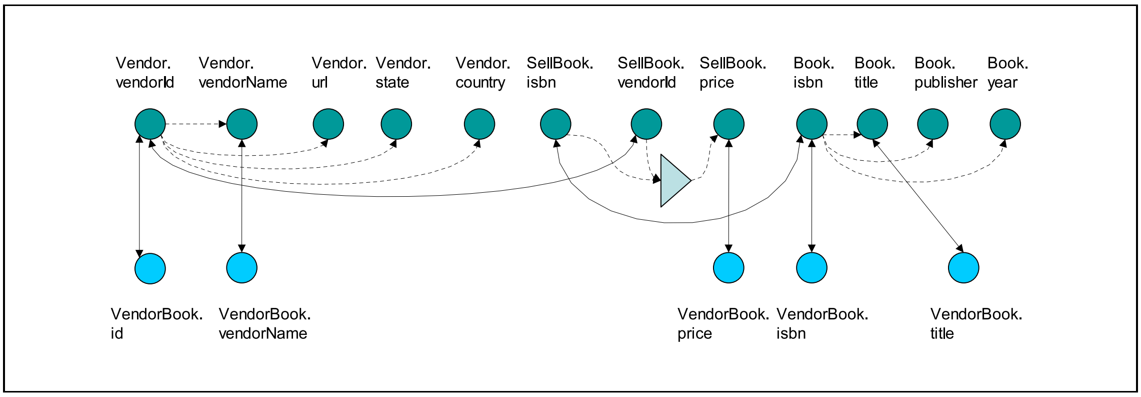


Figure 2.5: View Dependency Graph for view *VendorBook*

that there is an FD-node involving a non-singleton set from {*SellBook.vendorId, SellBook.isbn*} to *SellBook.price*.

Having defined view graphs, it is necessary to define paths on view graphs.

Let A, B, B_1, B_2, …, B_n be nodes and W, Y and Z be sets of nodes in $F(V)$. A *path* in $F(V)$ is defined as follows:

- There is a path from every node to itself;
- If there is an arc (B, A), then there is a path from B to A;
- Let f be an FD-node representing f: $B_1, \ldots, B_k \to_{R_i} A$. If there is a path from Y to every B_j, $1 \leq j \leq k$, then there is a path from Y to A (see Figure 2.6).
- If there is a path from a subset of Y to A, then there is a path from Y to A.
- If there is a path from Y to every node in Z, then there is a path from Y to Z.
- If there is a path from Y to Z, and a path from Z to W, then there is a path from Y to W.

The notation $Y \to_V Z$ says that there is a path in $F(V)$ from Y to Z.

As an example of path, in the view dependency graph of Figure 2.5, there is a path from *VendorBook.id* to *Vendor.country*, denoted *VendorBook.id* $\to_V$ *Vendor.country*, but there is no path from *Vendor.country* to *VendorBook.id*.

Now it is easy to see why we do not need to include functional dependencies considering all of the superkeys of a relation R in the view dependency graph. Consider the functional dependency *vendorId, vendorName* → *url* derived by the superkey

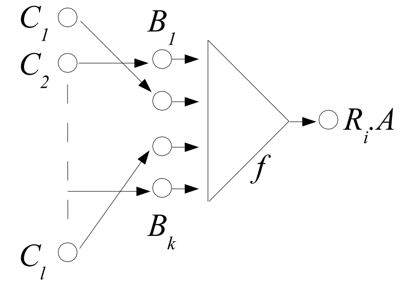


Figure 2.6: A path from $Y = \{C_1, \ldots, C_n\}$ to A through FD-node f

rule on table Vendor. If one needs to find a path from {*vendorId, vendorName*} to {*url*}, it is enough that there is a path from *vendorId* to *url*[2]. This shows that functional dependencies derived from superkeys are not necessary, since a path can always be found using only the functional dependencies derived from the primary key of the relation.

Before showing the translation algorithms, we want to give the intuition of *paths* in the View Dependency Graph. In (DAYAL; BERNSTEIN, 1982a), Dayal and Bernstein define *sources* and *clean sources*. Loosely speaking, a tuple t in the database is the *source* of a tuple v in the view if t was used to construct the view tuple v. Additionally, a tuple t in the database is a *clean source* of a tuple v in the view, if t is a *source* of v, but it is source of no other tuple in the view. This means that when v is updated, the update can be translated to affect t, and this will surely affect no other tuple in the view (no side-effects). In this way, traces in the View Trace Graph denote sources, and *paths* in the View Dependency Graph denote *clean sources*.

2.2.1.3 Translation Algorithms

Dayal and Bernstein consider simple[3] update operations over relational views. We now present the translation algorithms for insertions, deletions and modifications over a relational view V.

Deletions

Let V be a relational view. A *simple deletion* on V is a deletion $u(Y)$ of the form:

```
DELETE
FROM V $v$
WHERE $v.D_1 = c_1$ AND ... AND $v.D_k = c_k$
```

where Y is the set of the view columns D_i specified in the condition on the WHERE clause.

The translation procedure for deletions is:

[2] By using the rule "If there is a path from a subset of Y to A, then there is a path from Y to A".

[3] Simple update operations are the ones whose qualifications are conjunctions of equirestrictive clauses (DAYAL; BERNSTEIN, 1982a).

Step 1 Choose one relation R_i occurring in the FROM clause of V.

Step 2 The translation of the deletion will be

DELETE FROM R_i WHERE $R_i.K_i$ IN (SELECT $t_i.K_i$ FROM R_1 $t_1, ..., R_m$ t_m WHERE <qual> AND SOURCE(D_1) = c_1 AND ... AND SOURCE(D_k) = c_k)

where K_i is the primary key of R_i, and $SOURCE(D_j)$ denotes the expression $R_l.A_{m_j}$ if A_{m_j} generates D_j.

As an example, consider the following deletion over the view *VendorBook* of Example 2.1.

```
DELETE
FROM VENDORBOOK
WHERE title = "Computer Networks"
```

This would be translated to:

```
DELETE
FROM SellBook
WHERE (SellBook.vendorId, SellBook.isbn) IN
   (SELECT v.vendorId, b.isbn
    FROM Vendor v, SellBook sb, Book b
    WHERE v.vendorId = sb.vendorId AND b.isbn = sb.isbn
        AND b.title = "Computer Networks")
```

Correctness: Dayal proves that this procedure always exactly translates $u(Y)$ to the underlying relational database iff $X_i \rightarrow_V Y$, where $R_i(X_i)$ is the relation scheme chosen in Step 1 and Y is the set of the view columns D_i specified in the condition on the WHERE clause of the deletion over the view. The term *exact translation* means side-effect free, as explained in Section 2.2.

In the case of the example, the translation is exact because there is a path from the attributes of SellBook to the view attribute *title*, that is, {*SellBook.vendorId, SellBook.isbn, SellBook.price*} $\rightarrow_V$ {*VendorBook.title*} holds (see Figure 2.5).

Notice that there may be more than one translation for a given deletion. In the case of the example, we could have chosen any of the relations Vendor, SellBook or Book to translate the deletion. However, choosing Vendor would lead to a non exact translation, because there is no path from the attributes of Vendor to *VendorBook.title* in the view dependency graph of Figure 2.5. Additionally, choosing Book would exactly translate the deletion, but there would be a problem when executing the deletion if deletions do not cascade in the database. In our implementation, we order the tables according to foreign key constraints, so that SellBook will always be chosen before Book and Vendor to translate a deletion. In case using SellBook results in a non exact translation, we try the next base table until we find a table that satisfies the conditions for exact translation. There may be cases, however, where no base table satisfies the conditions. In such cases, it is not possible to translate the deletion without causing side-effects.

Insertions

In Dayal's approach, insertions can have a particular behavior when dealing with NULL values in view tuples. Dayal's translation mechanism works in a way that, if there is a tuple v in the view that has NULL values in some of its attributes, and the insert operation is trying to insert a tuple v' that is the same as v, but has some non-null values in some of the attributes that v has NULLs (that is, v' is more defined than v), then remove v and insert v'. In cases where v' contains less information than v, or is exactly equal to v, then the insertion is ignored. This procedure is called *reduce*. The procedure tries to determine whether or not the tuple that is being inserted is a *new* tuple. This notion of determining when two tuples represent the same tuple was first introduced in literature by (FURTADO; SEVCIK; SANTOS, 1979).

Let u be an insertion of a tuple v on a view V.

```
INSERT INTO $V$ ($D_1$, ..., $D_n$)
VALUES ($c_1$, ..., $c_n$)
```

This insertion is translated as follows.

For each relation $R_i \in \text{Rels}(V)$ (recall that $\text{Rels}(V)$ is the set of relations over which view V is defined), define a tuple t_i to be inserted into R_i as.

```
INSERT INTO $R_i$ ($A_1,\ldots,A_k$) VALUES (FindValue($A_1$), ..., FindValue($A_k$))
```

where function *FindValue(A)* is given in Algorithm 2.1.

```
function findValue($A$)

if $A$ has a V-trace $v.D$ AND $v[D]$ <> NULL then
  return $v[D]$
else
  if exists an FD $L \rightarrow A$ in $R_i$ AND exists $t_i$' in $R_i$ AND $t_i$'$[L] = t_i[L]$ AND $t_i$'$[A]$ <> NULL
  then
    return $t_i$'$[A]$
  else
    if exists $R_j.B$, such that there is a path from $R_j.B$ to $R_i.A$ in G(V) then
      return $t_j[B]$
    else
      return NULL
    end if
  end if
end if
```

Algorithm 2.1: Dayal and Bernstein's function *findValue*

As an example, consider the following insertion over view *VendorBook* shown in Example 2.1.

```
INSERT INTO VENDORBOOK (id, vendorName, isbn, title, price)
VALUES ("03", "Saraiva", 1111, "Unix Network Programming", 38)
```

This insertion would be translated as follows:

```
INSERT INTO VENDOR (vendorId, vendorName, url, state, country)
VALUES ("03", "Saraiva", null, null, null)

INSERT INTO SELLBOOK (vendorId, isbn, price)
VALUES ("03", 1111, 38)

INSERT INTO BOOK (isbn, title, publisher, year)
VALUES (1111, "Unix Network Programming", "Prentice Hall", 1998)
```

Notice that the insertion on table Book will be eliminated by the *reduce* algorithm, since there is a tuple in table Book exactly like the one that is trying to be inserted. Notice also that the function *findValue* found a value for attributes *publisher* and *year* that were neither in the original insertion nor in the view itself.

Correctness: Assuming that the insertion of t_i does not violate any FD in R_i, this procedure will produce exact translations for u if the following conditions are satisfied (DAYAL; BERNSTEIN, 1982b):

1. The primary key of each $R_i \in Rels(V)$ must be traceable from V; and
2. The definition of V can be expressed as a sequence of definitions of views $V_1, \ldots, V_k$, where each $V_i(Z_i)$ is defined over two base relations $R(X), S(Y)$, such that
   (a) $X \rightarrow_{V_i} Z_i$ AND
   (b) ($Y \rightarrow_{V_i} Z_i$ OR (R and S are equijoined on A, B respectively AND $R[A] \subseteq S[B]$ AND $B \rightarrow Y$ in S)).

In the example, we have an exact translation, since the above conditions are satisfied. Condition 1 is satisfied because the primary keys *Vendor.vendorId*, *Book.isbn* and *SellBook.vendorId, SellBook.isbn* are traceable from the view *VendorBook*. Condition 2 requires that the view can be specified as a sequence of view definitions where each new view is defined over two base relations. In the case of the example, we have two pairs of relations: (*Vendor*, *SellBook*) and (*SellBook*, *Book*). For each of these pairs, we need to check conditions 2a and 2b.

- For the pair (*Vendor*, *SellBook*), lets assume R = *SellBook* and S = *Vendor*. Consequently, X are the attributes of *SellBook* and Y are the attributes of *Vendor*.

  Condition 2a requires that there is a path in the view dependency graph from the attributes of *SellBook* to all attributes of the view originated from *SellBook* and *Vendor*. This condition holds, and can be easily verified in the graph of Figure 2.5. Condition 2b is an OR. The first condition of this OR does not hold, since there is no path from the attributes of *Vendor* to all the attributes of the view originated from *SellBook* and *Vendor*. However, the second condition of the OR does hold. The second condition is an AND of three other conditions: (i) *Vendor* and *SellBook* are equijoined on *SellBook.vendorId* and *Vendor.vendorId*; (ii) *SellBook.vendorId* $\subseteq$ *Vendor.vendorId*, by definition of foreign key; (iii) *Vendor.vendorId* $\rightarrow$ {*Vendor.vendorName*, *Vendor.url*, *Vendor.state*, *Vendor.country*} holds, since *vendorId* is the primary key of *Vendor*.

  So, all the conditions hold for this pair of base relations.

- For the pair (*SellBook*, *Book*), lets assume $R = SellBook$ and $S = Book$.

  Condition 2a requires that there is a path in the view dependency graph from the attributes of *SellBook* to all attributes of the view originated from *SellBook* and *Book*. This condition holds, as explained above. Condition 2b is an OR. The first condition of this OR does not hold, since there is no path from the attributes of *Book* to all the attributes of the view originated from *SellBook* and *Book*. However, the second condition of the OR does hold. The second condition is an AND of three other conditions: (i) *Book* and *SellBook* are equijoined on *SellBook.isbn* and *Book.isbn*; (ii) *SellBook.isbn* $\subseteq$ *Book.isbn*; (iii) *Book.isbn* $\rightarrow$ $\{$*Book.title*, *Book.publisher*, *Book.year*$\}$ holds.

  Consequently, all the conditions hold for the second pair of relations, which proves that the above update translation is exact.

Modifications

Let u be a replacement on a view V. Let W be the set of view attributes specified for replacement in u. Let Y be the set of attributes specified in the qualification of u.

```
UPDATE V
SET $W_i$ = $r_i$
WHERE $Y_i$ = $v_i$
```

Let TLRels(u) be the set of relations R_i which has some attribute with a V-trace $D \in W$.

The translation procedure is as follows:

Step 1 For each relation $R_i \in$ TLRels(u), do.

```
UPDATE Ri SET SOURCE(Wi) = ri
WHERE $R_i.K_i$ IN (SELECT $t_i.K_i$ FROM $R_1$ $t_1, ..., R_m$ $t_m$ WHERE <qual> AND
SOURCE(Yi) = vi)
```

where K_i is the primary key of R_i, and $SOURCE(D_j)$ denotes the expression $R_l.A_{m_j}$ if A_{m_j} generates D_j.

Consider the following modification over the view of Example 2.1.

```
UPDATE VENDORBOOK
SET title="New Title"
WHERE id="01" AND isbn="1111"
```

This modification is trying to change the title of book whose isbn is 1111 for vendor 1. This will be translated as follows:

```
UPDATE BOOK
SET BOOK.title="New Title"
WHERE BOOK.isbn IN
   (SELECT b.isbn
    FROM Vendor v, SellBook sb, Book b
    WHERE v.vendorId = sb.vendorId AND b.isbn = sb.isbn
    AND v.vendorId="01" AND b.isbn="1111")
```

Correctness: Dayal and Bernstein (1982a) prove that this procedure will always exactly translate the replacement u iff

1. For all $R_i(X_i) \in \text{TLRels}(u)$, there is a path $X_i \rightarrow_V Y$; and
2. For all $D \in W$, if D is a V-trace of some $R_i.A$ that appears in a join clause in the view qualification, then there is a path $X_i \rightarrow_V Z$, where Z is the set of attributes in the view V.

Also, u must not set to NULL a V-trace of any primary key attribute of any relation name R_i occurring in the FROM clause of V.

Checking the above conditions against the example, we conclude that the translation is not exact. This is because according to condition 1, there must be a path from the attributes of *Book* to {*VendorBook.id, VendorBook.isbn*}. However, there is no such path, because *VendorBook.id* is not reachable from any attribute of *Book*. Since condition 1 does not hold, condition 2 does not need to be checked.

It is easy to understand why the example translation is not exact. In the view, there is more than one occurrence of the book whose isbn is 1111. However, we are trying to modify just one of them – the one associated with vendor 01. Clearly, if we update the title of book 1111 in table Book, this would also affect the other occurrences of that book in the view, and not only the one associated with vendor 01. Thus, we have a side-effect, since the translated update does not affect only the tuples specified by the user update.

2.3 Chapter Remarks

The absence of a proposal that allows a user to update a relational database through XML views constructed over legacy relational databases motivated the development of this thesis. We have studied the requirements of view update translators, but decided against defining a specific translator for XML updates. Instead, we have decided to take advantage of the existing proposals on updates through relational views.

We do this by mapping XML views to relational views, and mapping updates on the XML view to updates on the corresponding relational views. In our implementation, we have chosen to use the approach of (DAYAL; BERNSTEIN, 1982a). In Chapter 6, we discuss how we use this approach to study the updatability of XML views, and on Chapter 7 we present the architecture of the Update Module, which uses the approach of Dayal and Bernstein to translate the updates to the underlying relational database.

3 XML EXTRACTION FROM RELATIONS

There has been a lot of work addressing the problem of building and querying XML views from relational databases; storing XML documents in relational databases and reconstructing them; and extracting XML documents from legacy databases.

In this chapter, we overview such approaches and present in details two approaches of building and querying XML views over relational databases, and one approach of extracting XML documents from relational databases. They are, respectively, SilkRoute (FERNÁNDEZ et al., 2002), XPERANTO (SHANMUGASUNDARAM et al., 2001) and DB2 XML Extender (CHENG; XU, 2000). To illustrate these approaches, we will show how they would construct the XML view of Figure 1.1. We have chosen these approaches since we consider them to have caused more impact in the database community. We do not study in details any of the approaches to store XML documents in relational databases, since this is not the goal of our work.

The remaining of this chapter is organized as follows: Section 3.1 overviews approaches to build and query XML views over relational databases. Particularly, it presents SilkRoute and XPERANTO in details. Section 3.2 overviews the approaches that extract XML documents over relational databases, giving special attention to DB2 XML Extender. Section 3.3 overviews the approaches to store XML documents in relational databases. In Section 3.3.1, we present approaches that are capable of updating XML documents stored in relational databases. Finally, Section 3.4 concludes with final remarks.

3.1 Building and Querying XML views over Relational Databases

Several approaches in literature explore the subject of building and querying XML views over relational databases (FERNÁNDEZ et al., 2002; SHANMUGASUNDARAM et al., 2001; BOHANNON et al., 2002; CHAUDHURI; KAUSHIK; NAUGHTON, 2003; SHANMUGASUNDARAM et al., 2000). Most of them approach the problem by building a *default* XML view from the relational source and then using an XML query language to query the default view (FERNÁNDEZ et al., 2002; SHANMUGASUNDARAM et al., 2001; BOHANNON et al., 2002; CHAUDHURI; KAUSHIK; NAUGHTON, 2003). They usually allow the definition of views over views and their concern is basically the following:

1. How to compose queries with view definitions, in order to get a *single* resulting

expression that represents both the query and the view;

2. How to use the relational engine to execute the query resulting from Step 1;

3. How to use the relational tuples resulting from Step 2 to build the resulting XML document.

Each of the existing approaches uses a different technique to construct the XML view using the relational engine to retrieve data. Some transform the XML view definition into extended SQL (SHANMUGASUNDARAM et al., 2000, 2001; CHAUDHURI; KAUSHIK; NAUGHTON, 2003), other use internal representations to map the XML view to several SQL queries (FERNÁNDEZ et al., 2002; BOHANNON et al., 2002). We now present SilkRoute and XPERANTO in details.

3.1.1 SilkRoute

SilkRoute was first proposed in 2000 (FERNÁNDEZ; TAN; SUCIU, 2000). At that time, XML-QL (DEUTSCH et al., 1999) was the query language used to query the XML views. A different language (RXL) was used to define the public view[1]. SilkRoute has not stopped in time, and it evolved in a very nice way through the past years (FERNÁNDEZ et al., 2001; FERNÁNDEZ; MORISHIMA; SUCIU, 2001). The current proposal (FERNÁNDEZ et al., 2002) uses XQuery to both define the public view and to query it.

In SilkRoute, initially there is an XML view called *canonical XML view.* It consists of a default representation of the underlying relational tables. For example, the relational table *Book* of Figure 1.2 is represented in the canonical view as follows:

```
<Book>
  <Tuple>
     <isbn>1111</isbn>
     <title>Unix Network Programming</title>
     <publisher>Prentice Hall</publisher>
     <year>1998</year>
  </Tuple>
  <Tuple>
     <isbn>2222</isbn>
     <title>Computer Networks</title>
     <publisher>Prentice Hall</publisher>
     <year>1996</year>
  </Tuple>
</Book>
```

From this canonical view, the DBA defines a *public view.* This is what corresponds to the default view, and this is the view that will be queried by users. The public view is defined in XQuery. To access data from the canonical view, SilkRoute offers a special variable called `$CanonicalView`. In the same way, users write queries over the public view using XQuery, and refer to the public view by a special variable called `$PublicView`.

To exemplify, suppose the DBA wants to construct the XML view of Figure 1.1 to play the role of the public view. The corresponding XQuery expression is shown on Figure 3.1. Notice that tuples of table *Vendor* are referenced in the query by `$CanonicalView/Vendor/Tuple`.

```
<vendors>
  {for $v in $CanonicalView/Vendor/Tuple
   return
  <vendor id='{$v/vendorId/text()}'>
     <vendorName>{$v/vendorName/text()}</vendorName>
     <address>
        <state>{$v/state/text()}</state>
        <country>{$v/country/text()}</country>
     </address>
      <products>
         {for $sb in $CanonicalView/SellBook/Tuple
          for $b in $CanonicalView/Book/Tuple
          where $sb/vendorId = $v/vendorId and
                $b/isbn = $sb/isbn
          return
          <book bprice='{$sb/price/text()}'>
             <btitle>{$b/title}</btitle>
             <isbn>{$b/isbn}</isbn>
          </book>
         }
         {for $sd in $CanonicalView/SellDVD/Tuple
          for $d in $CanonicalView/DVD/Tuple
          where $sd/vendorId = $v/vendorId and
                $d/asin = $sd/asin
          return
          <dvd dprice='{$sd/price/text()}'>
             <dtitle>{$d/title}</dtitle>
             <asin>{$d/asin}</asin>
          </dvd>
         }
      </products>
   </vendor>
  }
</vendors>
```

Figure 3.1: Definition of the Public View

```
<books>
   {for $v in $PublicView//vendor
    let $sb := $v//book
    let $c := count($sb)
    return
    <vendor vendorId="{$v/@id}">
       {$v/vendorName}
       <sellsBooks>{$c}</sellsBooks>
    </vendor>
   }
</books>
```

```
<books>
  <vendor vendorId="01">
    <vendorName>Amazon</vendorName>
    <sellsBooks>2</sellsBooks>
  </vendor>
  <vendor vendorId="02">
    <vendorName>Barnes and Noble</vendorName>
    <sellsBooks>2</sellsBooks>
  </vendor>
</books
```

Figure 3.2: Definition of the Application query and its result

The result of this query applied over the database of Figure 1.2 is exactly the XML view shown in Figure 1.1. SilkRoute, however, does not compute this view. It assumes the user is aware of its schema, and allows him to formulate queries over it. An example of a user query would be to select vendors and the quantity of books they sell. Such query is called an *application query*, and is shown in Figure 3.2. The result of the query is also shown on Figure 3.2.

In order to execute such query, SilkRoute composes the definition of the public

[1] The public view is the view that is visible to users. We give more details on it later.

view with the application query, obtaining a single resulting query. This composition, however, is done on an intermediate representation data structure called *view forest*. Each query in XQuery is mapped to a view forest, and then the view forests are composed into a single view forest. This view forest is then translated to one or more SQL queries over the relational database and to an XML template that will be used to obtain the resulting XML document.

A view forest is a forest (of trees) where each node is labeled with an XML label and an SQL fragment over the base relations. Internal nodes have SQL fragments consisting of a FROM clause and an optional WHERE clause. Leaf nodes have a required SELECT clause and optional FROM and WHERE clauses. To exemplify, Figure 3.3 shows the view forest corresponding to the query of Figure 3.1 (the public view).

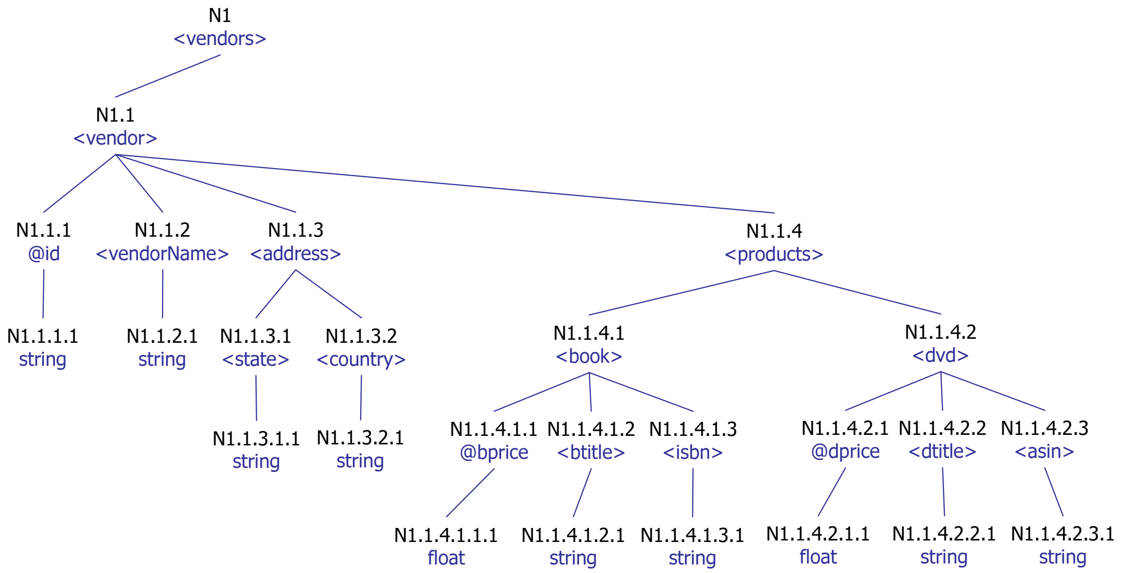
The view forest is constructed from the XQuery query, taking variable bindings and using them in the FROM clauses, and taking leaf element constructors and using them in the SELECT clauses. WHERE clauses are constructed from the where clause in XQuery. As an example, node `N1.1(<vendor>)` has an SQL FROM clause `FROM Vendor v` because it is being returned by an XQuery `for` clause that binds variable `$v` to the tuples of table Vendor (line 2 of Figure 3.1). Notice that the variable `$v` is being used as an alias in the SQL FROM clause. In the same way, node `N1.1.4.1(<book>)` has a FROM and a WHERE SQL clause corresponding to lines 11-14 of Figure 3.1. Leaf nodes have a SELECT clause that simply selects the value that was used to construct that node. As an example, node `N1.1.1.1(string)` has a SELECT clause `SELECT v.vendorId` since the attribute *vendorId* was used to construct the attribute `@id` in the XML view (line 4 of Figure 3.1).

This view forest is then composed with the view forest that corresponds to the application query. The resulting view forest is then used to generate SQL queries. We will not give details on how view forests are composed. For more information, please refer to (FERNÁNDEZ et al., 2002).

To exemplify how the resulting XML document is constructed, we will use the view forest of Figure 3.3 (supposing that this view forest is a result of a query composition - which in fact is not, but it serves to our purpose of exemplifying the generation of the XML instance). For each node in the view forest, a complete SQL query is generated. The query is obtained as follows: the FROM clause of a node n is the concatenation of the FROM clause of n and all of its ancestors. The WHERE clause is computed in the same way. The SELECT clause is the SELECT clause of n, or `SELECT *` if n is not a leaf node. As an example, the complete SQL queries for the nodes of the view forest of Figure 3.3 are shown on Figure 3.4.

Each of these SQL queries are then executed and associated to its corresponding node n. For each tuple associated with n, one XML element is generated. These elements are then glued together in a tree (or forest). A node d is connected below a node a if d is a child of a in the view forest, and if their corresponding tuples have the same variable bindings for the variables that are common to d and a. As an example, node `N1.1.4.1(<book>)` may have multiple children, as long as they agree in the bindings of variables `v`, `b` and `sb`. In the same way, a node `N1.1.4.1(<book>)` is connected to `N1.1.4(<products>)` if they agree on the binding of variable `v`. The XML instance resulting from this process is that of Figure 1.1.

As we could see from the example, SilkRoute does not extract the relational tables in XML, nor does it execute XQuery queries. Instead, it translates XQuery



(a)

```
N1(<vendors>)           := FROM()
N1.1(<vendor>)          := FROM Vendor v
N1.1.1(@id)             := FROM()
N1.1.1.1(string)        := SELECT v.vendorId
N1.1.2(<vendorName>)    := FROM()
N1.1.2.1(string)        := SELECT v.vendorName
N1.1.3(<address>        := FROM()
N1.1.3.1(<state>)       := FROM()
N1.1.3.1.1(string)      := SELECT v.state
N1.1.3.2(<country>)     := FROM()
N1.1.3.2.1(string)      := SELECT v.country
N1.1.4(<products>)      := FROM()
N1.1.4.1(<book>)        := FROM Book b, SellBook sb
                           WHERE sb.vendorId = v.vendorId AND b.isbn = sb.isbn
N1.1.4.1.1(@bprice)     := FROM()
N1.1.4.1.1.1(float)     := SELECT sb.price
N1.1.4.1.2(<btitle>)    := FROM()
N1.1.4.1.2.1(string)    := SELECT b.title
N1.1.4.1.3(<isbn>)      := FROM()
N1.1.4.1.3.1(<string>)  := SELECT b.isbn
N1.1.4.2(<dvd>)         := FROM DVD d, SellDVD sd
                           WHERE sd.vendorId = v.vendorId AND d.asin = sd.asin
N1.1.4.2.1(@dprice)     := FROM()
N1.1.4.2.1.1(float)     := SELECT sd.price
N1.1.4.2.2(<dtitle>)    := FROM()
N1.1.4.2.2.1(string)    := SELECT d.title
N1.1.4.2.3(<asin>)      := FROM()
N1.1.4.2.3.1(<string>)  := SELECT d.asin
```

(b)

Figure 3.3: View forest corresponding to the view definition of Figure 3.1: (a) tree representation (b) internal representation

```
N1.1(<vendor>)          := SELECT * FROM Vendor v
N1.1.1(@id)             := SELECT * FROM Vendor v
N1.1.1.1(string)        := SELECT v.vendorId FROM Vendor v
N1.1.2(<vendorName>)    := SELECT * FROM Vendor v
N1.1.2.1(string)        := SELECT v.vendorName FROM Vendor v
N1.1.3(<address>        := SELECT * FROM Vendor v
N1.1.3.1(<state>)       := SELECT * FROM Vendor v
N1.1.3.1.1(string)      := SELECT v.state FROM Vendor v
N1.1.3.2(<country>)     := SELECT * FROM Vendor v
N1.1.3.2.1(string)      := SELECT v.country FROM Vendor v
N1.1.4(<products>)      := SELECT * FROM Vendor v
N1.1.4.1(<book>)        := SELECT * FROM Vendor v, Book b, SellBook sb
                           WHERE sb.vendorId = v.vendorId AND b.isbn = sb.isbn
N1.1.4.1.1(@bprice)     := SELECT * FROM Vendor v, Book b, SellBook sb
                           WHERE sb.vendorId = v.vendorId AND b.isbn = sb.isbn
N1.1.4.1.1.1(float)     := SELECT sb.price FROM Vendor v, Book b, SellBook sb
                           WHERE sb.vendorId = v.vendorId AND b.isbn = sb.isbn
N1.1.4.1.2(<btitle>)    := SELECT * FROM Vendor v, Book b, SellBook sb
                           WHERE sb.vendorId = v.vendorId AND b.isbn = sb.isbn
N1.1.4.1.2.1(string)    := SELECT b.title FROM Vendor v, Book b, SellBook sb
                           WHERE sb.vendorId = v.vendorId AND b.isbn = sb.isbn
N1.1.4.1.3(<isbn>)      := SELECT * FROM Vendor v, Book b, SellBook sb
                           WHERE sb.vendorId = v.vendorId AND b.isbn = sb.isbn
N1.1.4.1.3.1(<string>) := SELECT b.isbn FROM Vendor v, Book b, SellBook sb
                           WHERE sb.vendorId = v.vendorId AND b.isbn = sb.isbn
N1.1.4.2(<dvd>)         := SELECT * FROM Vendor v, DVD d, SellDVD sd
                           WHERE sd.vendorId = v.vendorId AND d.asin = sd.asin
N1.1.4.2.1(@dprice)     := SELECT * FROM Vendor v, DVD d, SellDVD sd
                           WHERE sd.vendorId = v.vendorId AND d.asin = sd.asin
N1.1.4.2.1.1(float)     := SELECT sd.price FROM Vendor v, DVD d, SellDVD sd
                           WHERE sd.vendorId = v.vendorId AND d.asin = sd.asin
N1.1.4.2.2(<dtitle>)    := SELECT * FROM Vendor v, DVD d, SellDVD sd
                           WHERE sd.vendorId = v.vendorId AND d.asin = sd.asin
N1.1.4.2.2.1(string)    := SELECT d.title FROM Vendor v, DVD d, SellDVD sd
                           WHERE sd.vendorId = v.vendorId AND d.asin = sd.asin
N1.1.4.2.3(<asin>)      := SELECT * FROM Vendor v, DVD d, SellDVD sd
                           WHERE sd.vendorId = v.vendorId AND d.asin = sd.asin
N1.1.4.2.3.1(<string>) := SELECT d.asin FROM Vendor v, DVD d, SellDVD sd
                           WHERE sd.vendorId = v.vendorId AND d.asin = sd.asin
```

Figure 3.4: SQL queries for the nodes of the view forest of Figure 3.3

to SQL and uses the relational engine to execute the query. An important remark here is that the large amount of SQL queries generated by this approach may lead to low performance in the generation of the XML view.

3.1.2 XPERANTO

XPERANTO (SHANMUGASUNDARAM et al., 2001) is a proposal of IBM Almaden Research Group, so it uses resources of DB2 to process queries. In its first proposals (CAREY; KIERNAN; SHANMUGASUNDARAM; SHEKITA; SUBRAMANIAN, 2000; CAREY; FLORESCU; IVES; LU; SHANMUGASUNDARAM; SHEKITA; SUBRAMANIAN, 2000), XPERANTO also used XML-QL to define XML views. In the current proposal, views are defined using XQuery. A default XML view that represents the underlying relational database is used as source data in the user's view definitions.

The default view referring to the database of Figure 1.2 is as follows:

```
<db>
  <book>
    <row>
      <isbn>1111</isbn>
      <title>Unix Network Programming</title>
      <publisher>Prentice Hall</publisher>
      <year>1998</year>
    </row>
    <row>
      <isbn>2222</isbn>
      <title>Computer Networks</title>
      <publisher>Prentice Hall</publisher>
      <year>1996</year>
    </row>
  </book>
  <vendor>
    <row>...</row>
    ...
  </vendor>
  ...
</db>
```

In the user-defined view, the user accesses the default view by using a new input function called `view`. There is also an statement of `create view`, through which users assign names to views. As an example, the XML view of Figure 1.1 is generated as shown in Figure 3.5. Notice that the input function `view` is used to get relational data directly from the default view.

XPERANTO accepts views to be defined over views. As an example, if now one wants to query the view of Figure 3.5 to extract vendors and the number of books they sell, this could be done as shown in Figure 3.6. Another way of doing this would be a normal query instead of a new view definition. Both are acceptable by XPERANTO.

To process queries, XPERANTO also uses the relational engine as much as possible. First, it processes the query and converts it to an intermediate representation called XML Query Graph Model (XQGM). XQGM is very similar to QGM (Query Graph Model) (HAAS et al., 1989), which is the internal representation of SQL queries in DB2. As a consequence, this step facilitates using the relational engine to execute the query.

Differently from view forests, XQGM is a low level representation of queries. It is composed of a series of functions and operations that captures the semantics of XML queries. These functions and operators comprise both data retrieval and XML construction. In fact, the XML tagging is done outside the relational engine, but it uses the information on the XQGM together with data retrieved using the data retrieval functions to produce the final result.

Table 3.1 (taken from (SHANMUGASUNDARAM et al., 2001)) shows the operators allowed in a XQGM graph. As one can see, they are basically the relational algebra operators incremented by three additional operators: *table*, *view* and *unnest*. The operators *table* and *view* are used to access relational data, and *unnest* is used to unnest XML lists. Additionally, the *project* operator is used to call functions as well as to project attributes in the output.

The XML functions that can appear in an XQGM are shown in Table 3.2[2].

[2] This table was also taken from (SHANMUGASUNDARAM et al., 2001).

```
create view vendors as (
<vendors>
  {for $v in view("default")/vendor/row
   return
  <vendor id='{$v/vendorId/text()}'>
     <vendorName>{$v/vendorName/text()}</vendorName>
     <address>
        <state>{$v/state/text()}</state>
        <country>{$v/country/text()}</country>
     </address>
      <products>
         {for $sb in view("default")/sellbook/row
          for $b in view("default")/book/row
          where $sb/vendorId = $v/vendorId and
                $b/isbn = $sb/isbn
          return
          <book bprice='{$sb/price/text()}'>
             <btitle>{$b/title}</btitle>
             <isbn>{$b/isbn}</isbn>
          </book>
         }
         {for $sd in view("default")/selldvd/row
          for $d in view("default")/dvd/row
          where $sd/vendorId = $v/vendorId and
                $d/asin = $sd/asin
          return
          <dvd dprice='{$sd/price/text()}'>
             <dtitle>{$d/title}</dtitle>
             <asin>{$d/asin}</asin>
          </dvd>
         }
      </products>
   </vendor>
  }
</vendors>)
```

Figure 3.5: User defined view in XPERANTO

```
create view books as (
<books>
   {for $v in view("vendor")//vendor
    let $sb := $v//book
    let $c := count($sb)
    return
    <vendor vendorId="{$v/@id}">
       {$v/vendorName}
       <sellsBooks>{$c}</sellsBooks>
    </vendor>
   }
</books>)
```

```
<books>
  <vendor vendorId="01">
    <vendorName>Amazon</vendorName>
    <sellsBooks>2</sellsBooks>
  </vendor>
  <vendor vendorId="02">
    <vendorName>Barnes and Noble</vendorName>
    <sellsBooks>2</sellsBooks>
  </vendor>
</books
```

Figure 3.6: Definition of a new XML view defined over the view of Figure 3.5

Table 3.1: XQGM Operators

Operator	Description
Table	Represents a table in a relational database
Project	Computes results based on its input
Select	Restricts the input
Join	Joins two or more inputs
Groupby	Applies aggregate functions and grouping
Orderby	Sorts input based on column values
Union	Unions two or more inputs
Unnest	Unnests an XML list
View	Represents a view
Function	Represents an XQuery function

Table 3.2: XML Functions and the operators in which they can appear

XML Function	Description	Operator
cr8Elem(*Tag*, *Atts*, *Clist*)	Creates an element with tag name *Tag*, attribute list *Atts*, and contents *Clist*	Project
cr8AttList(A_1, ..., A_n)	Creates a list of attributes from the attributes passed as parameters	Project
cr8Att($Name$, Val)	Creates an attribute with name $Name$ and value Val	Project
cr8XMLFragList(C_1, ..., C_n)	Creates an XML fragment list from the content (element/text) parameters	Project
aggXMLFrags(C)	Aggregate function that creates an XML fragment list from content inputs	Groupby
getTagName(*Elem*)	Returns the element name of *Elem*	Project, Select
getAttributes(*Elem*)	Returns the list of attributes of *Elem*	Project, Select
getContents(*Elem*)	Returns the XML fragment list of contents (elements/text) of *Elem*	Project, Select
getAttName(*Att*)	Returns the name of attribute *Att*	Project, Select
getAttValue(*Att*)	Returns the value of attribute *Att*	Project, Select
isElement(E)	Returns true if E is an element, returns false otherwise	Select
isText(T)	Returns true if T is text, returns false otherwise	Select
unnest(*List*)	Superscalar function that unnests a list	Unnest

The process of translating an XQuery view definition to XQGM is very complex. In (SHANMUGASUNDARAM et al., 2001), this is shown through an example, but general translation rules are not given (probably because this project is associated with a commercial DBMS). In general lines, a query in XQuery is translated to a single SQL query that contains several sub-queries that are put together by a *correlated join* operation. This operation, however, is not efficient, so they transform the query (by a *decorrelation process*) into a query that uses joins to compute intermediate results, and outer joins to relate nested elements with their ancestors.

An example of an XQGM is shown in Figure 3.7. It represents the view definition of Figure 3.5. It is important to notice that this is only a try of showing an XQGM graph, since the paper does not provide enough details on how the graph is constructed.

This figure represents the XQGM before the decorrelation process. Each box corresponds to one of the operations of Table 3.1. Notice that first (box 1), the data in table Vendor is retrieved. Each element *book* and *dvd* is constructed by a subquery (boxes 2-7 and 8-13, respectively) that is later joined with vendor (box 14).

The XML templates of this figure (boxes 6, 12, 15 and 17) are shown just for a better understanding. In fact, they are XML function calls that construct XML elements using the functions of Table 3.2. As an example, box 6 of Figure 3.7 corresponds to the function applications shown in Figure 3.8.

A view definition in XQGM is composed with the view definitions referenced in the original query to produce a single XQGM that will be executed. More details on how the composition is done are available in (SHANMUGASUNDARAM et al., 2001).

3.2 Extracting XML Documents from Relational Databases

The goal of the approaches that extract XML documents from relational databases is very similar to the approaches that build XML views from relational databases, shown in the previous section. The difference is that they do not explicitly consider the resulting XML document as a *view*. That is, the resulting XML document is usually directly used by an application, and there is no support for queries over the resulting documents. This is in fact a very small difference, since once we have the XML document, we can use XQuery to query it without having to use the relational engine to do so. Summarizing, the approaches we present in this section do not comprise queries over the views, and as a consequence, they do not perform query composition. From now on, we call the XML documents extracted from the relational databases, XML Views.

There are several tools that are capable of doing so. Some of them apply a default mapping over an SQL query specified by the user (TURAU, 2001). Others allow the user to specify an XML template, together with SQL instructions that will be used to generate the resulting XML document (INTELLIGENT SYSTEM RESEARCH, 2001). Still others require the user to explicitly identify how database columns and tables are mapped to XML elements and attributes (MERGEN; KELLER; KROTH, 2002). As for updates, the AXIS tool (MERGEN; KELLER; KROTH, 2002) allows the insertion of XML documents in the database, as well as some kinds of updates and deletions. The limitation of this approach resides in the absence of

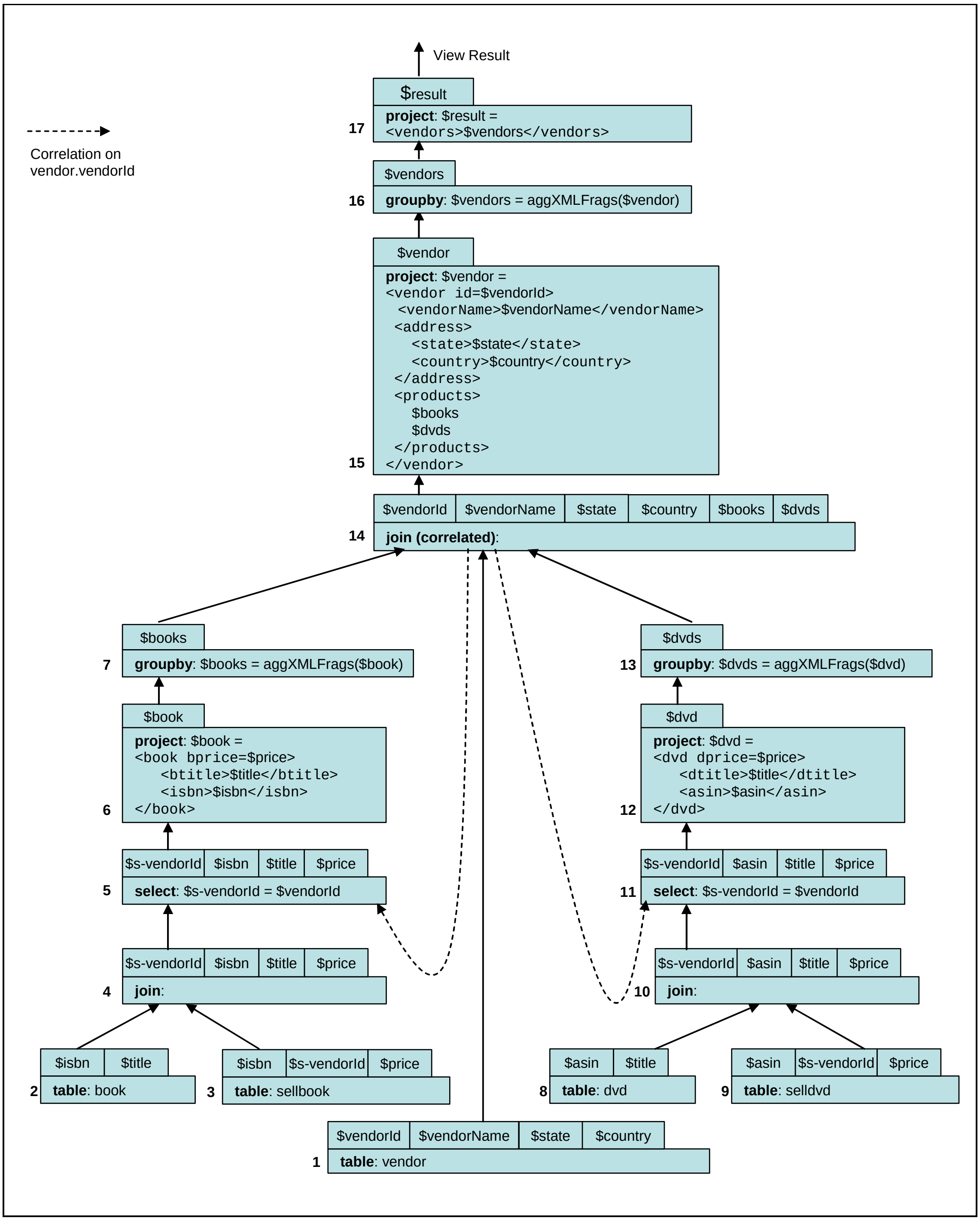


Figure 3.7: XQGM corresponding to view definition of Figure 3.5

```
cr8Elem(book,
    cr8AttList(cr8Att(bprice, $price)),
    cr8XMLFragList(cr8Elem(btitle,
                            cr8AttList(),
                            cr8XMLFragList($title)),
                   cr8Elem(isbn,
                            cr8AttList(),
                            cr8XMLFragList($isbn)),
                   )
          )
```

Figure 3.8: Expansion of box 6 in Figure 3.7

a view definition language. The user specifies the mapping database/XML using a graphical tool. Also, update operations are associated with database tables, and not XML documents or explicit update operations. In this way, when an XML document is submitted to the tool, some of the elements may be inserted, some may be deleted, and some may be updated, depending on to which relational table they correspond. We think that this may lead to confusions in some cases. It is important to state that the goal of this tool is to provide ways of permitting data exchange among companies with heterogeneous databases.

The commercial relational databases also offer support for extracting XML views over relational databases. IBM DB2 XML Extender (CHENG; XU, 2000) uses a mapping file called DAD (*Data Access Definition*) to specify how a given SQL query is mapped to XML. This mapping file is very complex, and is generally built using a wizard. Oracle 9i release 2 uses SQL/XML (EISENBERG; MELTON, 2002). SQL Server extends SQL with a directive called `FOR XML` (CONRAD, 2001a). It also provides an alternative way of expressing XML views, which can be done by an annotated XML Schema. The schema reflects the view structure the user wants to construct, and it is incremented by annotations that tells where (database table and column) that element must come from. The XML instance is not generated from the schema. The user needs to specify an XPath query over the view in order to get the instance (or portions of it).

As we can see, most commercial databases have their own way of dealing with XML, which makes it difficult to use them for accessing legacy databases. As for updates, DB2, which allows the creation of XML documents from relational tables, requires that updates be issued directly to the relational tables. In SQL Server an XML view generated by an annotated XML Schema can be modified using *updategrams*. Instead of using INSERT, UPDATE or DELETE statements, the user provides a before image of what the XML view looks like and an after image of the view (CONRAD, 2001b). The system computes the difference between these images and generates corresponding SQL statements to reflect changes on the relational database. Oracle offers the option of specifying an annotated XML Schema, but the only possible update operation is to insert XML documents that agree with an annotated XML Schema.

Since SQL Server appears to be the most complete solution regarding updates, let's take a look on their solution more carefully. Their idea of *updategrams* is very similar to our fist proposal of updates through views (BRAGANHOLO; HEUSER; VITTORI, 2001). However, at that time, we did not had an explicit command to provide the before and after images of the updated view. Instead, we would simply compare the original view with the updated one, detect what had changed and map the updates to the underlying database. In SQL Server, the user explicitly writes an XML code to inform the change. As an example, let's assume that the `vendors.XSD` is the annotated XML Schema that generates the view of Figure 1.1. To modify the address of a vendor, the user specifies:

```
<ROOT xmlns:updg="urn:schemas-microsoft-com:xml-updategram">
<updg:sync mapping-schema="vendors.XSD" >
  <updg:before>
     <vendor id="01">
       <vendorName>Amazon</vendorName>
       <address>
          <state>WA</state>
          <country>US</country>
       </address>
     </vendor>
  </updg:before>

  <updg:after>
    <vendor id="01">
       <vendorName>Amazon</vendorName>
       <address>
          <state>PA</state>
          <country>US</country>
       </address>
     </vendor>
  </updg:after>
</updg:sync>
</ROOT>
```

The problem with this approach is that they do not provide details on how the updates are mapped to the database. They also do not provide details on the power of expression of these annotated XML Schemas. Besides, it is an specific solution for SQL Server, so database instances stored in a different RDMS can not benefit from this approach unless we migrate it to SQL Server. This option, however, is not the best one, since SQL Server is a proprietary solution.

Native XML databases like XIndice (APACHE SOFTWARE FOUNDATION, 2002), Timber (JAGADISH et al., 2002) and Tamino (SOFTWARE AG, 2002; SCHÖNING, 2001) also support updates. However, the goal of all these systems differs from ours since they do not update through views, nor the source data is relational.

We now analyze DB2 XML Extender in details.

3.2.1 DB2 XML Extender

DB2 XML Extender (CHENG; XU, 2000) provides ways of both storing XML documents in relational databases, as well as extracting XML documents from pre-existing relations. In this section, we focus on this latter feature.

In order to be able to create XML documents from stored relational data, it is necessary to provide a mapping file called DAD (Document Access Definition). DAD files can express the mapping XML-relations in two distinct ways: using the method `SQL_stmt` or using the method `RDB_node`. The `SQL_stmt` method restricts the view to be built from a single SQL statement. As a consequence, views such as the one in Figure 1.1 can not be expressed.

On the other hand, the `RDB_node` method allows the specification of the table and attributes that are used to construct a given element, so it allows more flexible views. As an example, Figure 3.9 shows how the XML view of Figure 1.1 would be specified in DB2 XML Extender, assuming the database of Figure 1.2.

The mapping file requires that the user specifies all the relational tables that will be used in the generation of the XML file, right under the element root (lines 7-14). It is also necessary to specify the join conditions for those tables (lines 12-13). Additional selection conditions can also be specified (such as *sellbook.price* > *30*, for instance). After that, each text XML element has to specify the relational source data, using the elements `table` and `column`. As an example, the *id* attribute is taken from table *Vendor* and column *vendorId* (lines 16-18).

This approach has some limitations. First, the mapping file is rather complex. Second, it only works with DB2, thus it can not be used to extract XML documents of other RDBMSs. Finally, we have no access to more detailed information. That is, it is not clear how the DAD file is used to generate the XML document. This sometimes makes it difficult to predict the result of a given mapping file.

3.3 Storing and Querying XML documents in Relational Databases

(ABITEBOUL; CLUET; MILO, 1993) was one of the first proposals to suggest the use of relational database engines to query data in files. For this purpose, the file contents should be stored in the relational database using some mapping schema. For this to be possible, the file should have a "strict inner structure". This requirement resembles XML documents which have a DTD or schema.

Based on this observation, lots of work arose with the goal of storing (FLORESCU; KOSSMANN, 1999; DEUTSCH; FERNANDEZ; SUCIU, 1999; SHANMUGASUNDARAM et al., 1999; LEE; CHU, 2000; CHEN; DAVIDSON; ZHENG, 2002, 2003) and querying (SHANMUGASUNDARAM et al., 1999; MANOLESCU; FLORESCU; KOSSMANN, 2001; SHANMUGASUNDARAM et al., 2001; TATARINOV et al., 2002; DEHAAN et al., 2003) XML documents using relational databases. The interesting point is that some of the solutions explore the XML structure (parent-child) to be able to store documents that do not have DTDs (FLORESCU; KOSSMANN, 1999; DEHAAN et al., 2003).

These approaches are different from ours because they consider a different question: they query XML documents stored in relational databases, while we query relational databases to extract XML views. Therefore, the underlying assumptions used are different. For example, querying XML documents stored in relational databases must preserve document order, while in our case, order is not important, since the relational model is unordered. Another difference, pointed by (TATARINOV et al., 2001), is that a relational repository for XML is usually automatically generated based on the schema of the XML instances, while XML views are constructed over pre-existing relational tables, together with a view definition.

On the other hand, the flat nature of relational databases may cause redundancy when translated to XML views, which may cause problems regarding updates. That is, a well designed relational database does not imply a redundancy-free XML view.

```
<DAD>
 <dtdid>c:\dxx\dtd\vendors.dtd</dtdid>
 <validation>NO</validation>
 <Xcollection>
  <prolog>?xml version="1.0"?</prolog>
  <doctype>!DOCTYPE vendor SYSTEM "c:\dxx\dtd\vendors.dtd"</doctype>
  <root_node>
   <element_node name="vendors">
    <RDB_node>
     <table name="vendor"/><table name="book"/><table name="sellbook"/>
     <table name="dvd"/><table name="selldvd"/>
     <condition>vendor.vendorId=sellbook.vendorId AND sellbook.isbn=book.isbn AND
        vendor.vendorId=selldvd.vendorId AND selldvd.asin=dvd.asin</condition>
    </RDB_node>
    <element_node name="vendor" multi_occurrence="YES">
     <attribute_node name="id">
      <RDB_node><table name="vendor"/><column name="vendorId"/></RDB_node>
     </attribute_node>
     <element_node name="vendorName">
      <text_node>
       <RDB_node><table name="vendor"/><column name="vendorName"/></RDB_node>
      </text_node>
     </element_node>
     <element_node name="address">
      <element_node name="state">
       <text_node>
        <RDB_node><table name="vendor"/><column name="state"/></RDB_node>
       </text_node>
      </element_node>
      <element_node name="country">
       <text_node>
        <RDB_node><table name="vendor"/><column name="country"/></RDB_node>
       </text_node>
      </element_node>
     </element_node>
     <element_node name="products">
      <element_node name="book" multi_occurrence="YES">
       <attribute_node name="bprice">
        <RDB_node><table name="sellbook"/><column name="price"/></RDB_node>
       </attribute_node>
       <element_node name="btitle">
        <text_node>
         <RDB_node><table name="book"/><column name="title"/></RDB_node>
        </text_node>
       </element_node>
       <element_node name="isbn">
        <text_node>
         <RDB_node><table name="sellbook"/><column name="isbn"/></RDB_node>
        </text_node>
       </element_node>
      </element_node>
      <element_node name="dvd" multi_occurrence="YES">
       <attribute_node name="dprice">
        <RDB_node><table name="selldvd"/><column name="price"/></RDB_node>
       </attribute_node>
       <element_node name="dtitle">
        <text_node>
         <RDB_node><table name="dvd"/><column name="title"/></RDB_node>
        </text_node>
       </element_node>
       <element_node name="asin">
        <text_node>
         <RDB_node><table name="selldvd"/><column name="asin"/></RDB_node>
        </text_node>
       </element_node>
      </element_node>
     </element_node>
    </element_node>
   </element_node>
  </root_node>
 </Xcollection>
</DAD>
```

Figure 3.9: View definition in DB2 XML Extender

This problem is not critical for XML documents stored in relational databases since well designed XML documents (EMBLEY; MOK, 2001; ARENAS; LIBKIN, 2002) tend not to be redundant. Additionally, existing proposals for updating XML documents stored in relational databases do not consider updates through views.

We now overview the approaches:

Edge The Edge approach (FLORESCU; KOSSMANN, 1999) uses a single relation to store the XML document. Each element or attribute of the XML document is stored in a tuple of this relation.

Attribute This approach requires one relational table for each element or attribute in the XML instance. Elements with the same name are stored in the same relation.

Shared Inlining The Shared Inlining method (SHANMUGASUNDARAM et al., 1999) explores the DTD to reduce the number of joins required to reconstruct the document. It does so by storing children together with their parents every time this is possible. As an example, in an XML document that stores information about vendors and the books they sell, information about vendor such as name and id would be stored in a single relation.

Hybrid Inlining This approach (SHANMUGASUNDARAM et al., 1999) is similar to the Shared Inlining approach. The difference is that this method reduces even more the number of generated relations by exploring additional features of the DTD.

Dynamic Interval The dynamic interval approach (DEHAAN et al., 2003) proposes to store the XML instance in a single relation. The relation stores the tag name (or atomic value), and an interval *(start, end)*. The interval can be calculated by a depth-first traversal in the document, assigning consecutive numbers to each node. Every time a node is reached, the *start* attribute receives the current number. Then when all the descendants of that node were visited, the *end* attribute is closed with the current counting value. In this way, the interval of children nodes is always contained in the interval of the parent node. The approach is called *dynamic* because it is possible to leave gaps in the intervals to accommodate insertion of new nodes.

Constraints Some of the approaches construct the relational schema based on constraints of the XML documents. (LEE; CHU, 2000) explores DTD constraints such as cardinalities and referential integrity. On the other hand, (CHEN; DAVIDSON; ZHENG, 2002) uses XML key and keyref, while (CHEN; DAVIDSON; ZHENG, 2003) explores XML functional dependencies (XFDs) to generate a relational schema that has as few redundancy as possible.

The querying techniques depend on the method adopted to store the XML in relations. The edge and attribute approaches require lots of joins to recompute the document. The dynamic interval uses sub-queries and the *union all* operator to reproduce the original XML instance.

3.3.1 Updating XML documents stored in Relational Databases

There are three proposals for updating XML documents stored in relational databases (TATARINOV et al., 2001, 2002; WANG; MULCHANDANI; RUNDENSTEINER, 2003). All of them assume a default mapping of the XML document to relations, such as the *shared inlining method* (SHANMUGASUNDARAM et al., 1999), or the approach of (LEE; CHU, 2000).

Tatarinov et al. (2001) propose an extension to XQuery to support updates. This extension comprises primitives such as `UPDATE`, `DELETE`, `INSERT (AFTER|BEFORE)`, `REPLACE` and `RENAME`. They then discuss how those updates could be translated to the underlying relational database, in order to update XML documents stored there using the shared inlining method. Notice that these translations are straightforward, since this is a direct update, not an update through views.

In (TATARINOV et al., 2002), Tatarinov et al. uses the XQuery extension proposed in (TATARINOV et al., 2001) to deal with ordered XML documents. They propose an storage method for ordered XML documents in relations, and show how updates that affect order can be translated.

Finally, (WANG; MULCHANDANI; RUNDENSTEINER, 2003) provides an updatability study for XML documents stored in relational databases. First, they require that the storage mapping be both a lossless data loading and a lossless constraints loading. That is, the storage mapping preserves all XML data, and also all the XML constraints such as cardinalities, hierarchy, etc. In this work, Wang, Mulchandani and Rudensteiner use the storage mapping of (LEE; CHU, 2000). They consider the following scenario: given an XML document stored in a relational database, and given an XQuery expression over the database that reconstructs exactly the original XML document, then using an update language similar to that of (TATARINOV et al., 2001), it is possible to translate all the updates to the database without changing the view complement (they adopt the view complement approach (BANCILHON; SPYRATOS, 1981) as their correctness criteria). In this work, however, the authors do not specify what is exactly the update language that is being used, and how updates are translated to the database.

3.4 Chapter Remarks

Most of the existing proposals to extract XML from relational databases translate the view definition to SQL in order to be able to use the relational engine to execute the query. As will be clear in Chapter 7, in our approach we have not taken this direction. We have decided to extract the relevant portions of relational tables to XML, and use an existing XQuery processor to generate the view. Notice, however, that this was done in order to simplify the view generation, since our focus was on updates, and not on the view generation.

We claim that we could have used an approach similar to the ones presented in this chapter to generate the view, in order to make it more efficiently. Specifically, we believe that an approach similar to SilkRoute could be very easily adapted to work with query trees.

The most important aspect taken from this bibliographical study is that there is no proposal in literature that is capable of updating the underlying relational database through XML views. In this aspect, we believe this thesis gives a significant contribution. We explain our approach in details in the next chapters.

4 BUILDING AND UPDATING XML VIEWS

In this chapter, we present the basis of our solution. We present a formalism – *query trees* – which we use to extract XML views from a relational database, and an update language that we use to update the XML view. In the next chapter, we show how XML views constructed by query trees can be mapped to a set of corresponding relational views, and how updates to the XML view can be mapped to updates over this set of relational views. We thus transform a new problem - updating relational databases through XML views - into an existing problem - updating relational databases through relational views.

A paper that presents an overview of *query trees* and of our mapping to relational views was published at the *International Conference on Very Large Data Bases* (BRAGANHOLO; DAVIDSON; HEUSER, 2004a).

4.1 Query Trees

Query trees are used as a representation of the XML view extraction query. We use this abstract representation rather than an XML query language syntax for several reasons:

- First, reasoning about updates and the updatability of an XML view is performed at this level. The characteristics considered in these reasonings are the structure of the XML view and the source of each XML element/attribute. These are syntax independent features, which allow us to work on a syntax independent level.
- Second, they are easy to understand yet expressive enough to capture several important aspects of XQuery such as nesting, composed attributes, and heterogeneous sets.[1]

Query trees can therefore be thought of as the intermediate processing form for a subset of many different XML query languages. For example, we have developed an implementation of our technique that uses a subset of XQuery (BRAGANHOLO; DAVIDSON; HEUSER, 2003b) as the top-level language. This will be shown in chapter 7.

After defining query trees, we introduce a notion which will be used to describe the mapping to relational queries: the abstract type of a query tree node. We use this notion of typing to define the semantics of query trees, and then present their result type DTD.

[1] They can also capture grouping, and we present such extension in Appendix A.1.

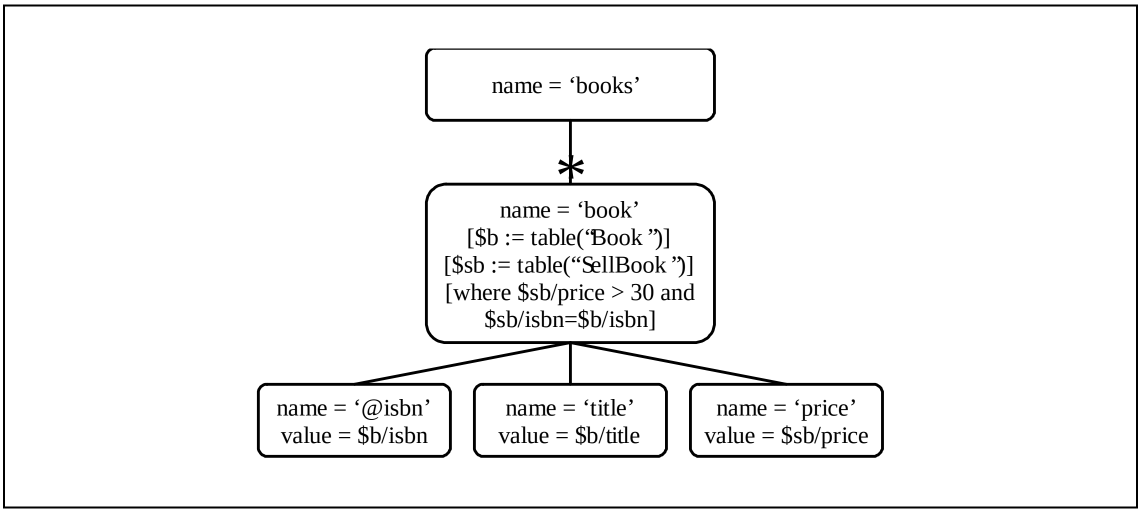


Figure 4.1: Example of query tree

4.1.1 Query Trees Defined

An example of a query tree can be found in Figure 4.1, which retrieves books that are sold for prices greater than $30. The query tree resembles the structure of the resulting XML view. The root of the tree corresponds to the root element of the result. Leaf nodes correspond to attributes of relational tables, and interior nodes whose incoming edges are starred capture repeating elements. The result of this query is as follows:

```
<books>
  <book @isbn="1111">
    <title>Unix Network Programming</title>
    <price>38</price>
  </book>
  <book @isbn="2222">
    <title>Computer Networks</title>
    <price>29</price>
  </book>
  ...
</books>
```

Query trees are very similar to the *view forests* of (FERNÁNDEZ et al., 2002) and *schema-tree queries* presented in (BOHANNON et al., 2002). The difference is that, instead of annotating all nodes with the relational queries that are used to build the content model of a given node, we annotate interior nodes in the tree using only the selection criteria (not the entire relational query).

An annotation can be a *source* annotation or a *where* annotation. Source annotations bind variables to relational tables, and *where* annotations impose restrictions on the relational tables making use of the variables that were bound to the tables.

In the definitions that follow, we assume that $\mathcal{D}$ is a relational database over which the XML view is being defined. $\mathbb{T}$ is the set of table names of $\mathcal{D}$. $\mathbb{A}_T$ is the set of attributes of a given table $T \in \mathbb{T}$.

Definition 4.1 (Query Tree) *A query tree defined over a database $\mathcal{D}$ is a tree with a set of nodes $\mathbb{N}$ and a set of edges $\mathbb{E}$ in which:* ***Edges*** *are simple or starred ("*-edge"). An edge is* simple *if, in the corresponding XML instance, the child node* *appears exactly once in the context of the parent node, and* starred *otherwise.* ***Nodes*** *are as follows:*

1. *All nodes have a name that represents the tag name of the XML element associated with this node in the resulting XML view. Names of leaf nodes that start with "@" are considered to be XML attributes.*
2. *Leaf nodes have, and only leaf nodes, a value (to be defined).*
3. Starred nodes *(nodes whose incoming edge is starred) may have one or more source annotations and zero or more where annotations (to be defined). A leaf node may be a starred node.*

Since we map XML views to relational views, nodes with the same name in the query tree may cause ambiguities in the mapping (a relation can not have two attributes with the same name (ULLMAN; WIDOM, 1997)). For simplicity, in this thesis we will ignore this problem and use unique names for nodes in the query trees. We deal with such problem in Appendix B.

Returning to the example in Figure 4.1, there is a *-edge from the root (named *books*) to its child named *book*, indicating that in the corresponding XML instance there may be several *book* subelements of *books*. There is a simple edge from the node named *book* to the node named *title*, indicating that there is a single *title* subelement of *book*. The node named *@isbn* will be mapped to an XML attribute instead of an element.

Before giving an example of how values are associated with nodes, we define *source* and *where* annotations on nodes of a query tree.

Definition 4.2 (Source Annotation) *A* source annotation *s within a starred node n is of the form* [$\$x :=$ table(T)], *where $\$x$ denotes a variable and $T \in \mathbb{T}$ is a relational table. We say that $\$x$ is bound to T by s.*

Definition 4.3 (Where Annotation) *A* where annotation *on a starred node n is of the form* [where $\$x_1/A_1$ op Z_1 AND ... AND $\$x_k/A_k$ op Z_k], *$k \geqslant 1$, where $A_i \in \mathbb{A}_{T_i}$ and $\$x_i$ is bound to T_i by a source annotation on n or some ancestor of n. The operator* op *is a comparison operator $\{=, \neq, >, <, \leqslant, \geqslant\}$. Z_l is either a literal (integer, string, etc.) or an expression of the form $\$y/B$, where $B \in \mathbb{A}_T$ and $\$y$ is bound to T by a source annotation on n or some ancestor of n.*

Definition 4.4 (Node Value) *The value of a leaf node n is of form $\$x/A$, where $A \in \mathbb{A}_T$ and $\$x$ is bound to table T by a source annotation on n or some ancestor of n.*

In Figure 4.1, the node *book* has source annotations and where annotations. The source annotations bind variable *$b* to the relational table *Book*, and variable $*sb* to the relational table *SellBook*. The where annotations restrict the books that appear in the view to those with price greater than $30, and specify the join condition of tables *Book* and *SellBook*. The value of the node *@isbn* is specified as *$b/isbn*, indicating that the content of the XML view attribute *isbn* will be generated using column *isbn* of the table *Book*.

A more complex example of a query tree can be found in Figure 4.2 (ignore for now the types τ associated with nodes). This query tree retrieves *vendors*, and

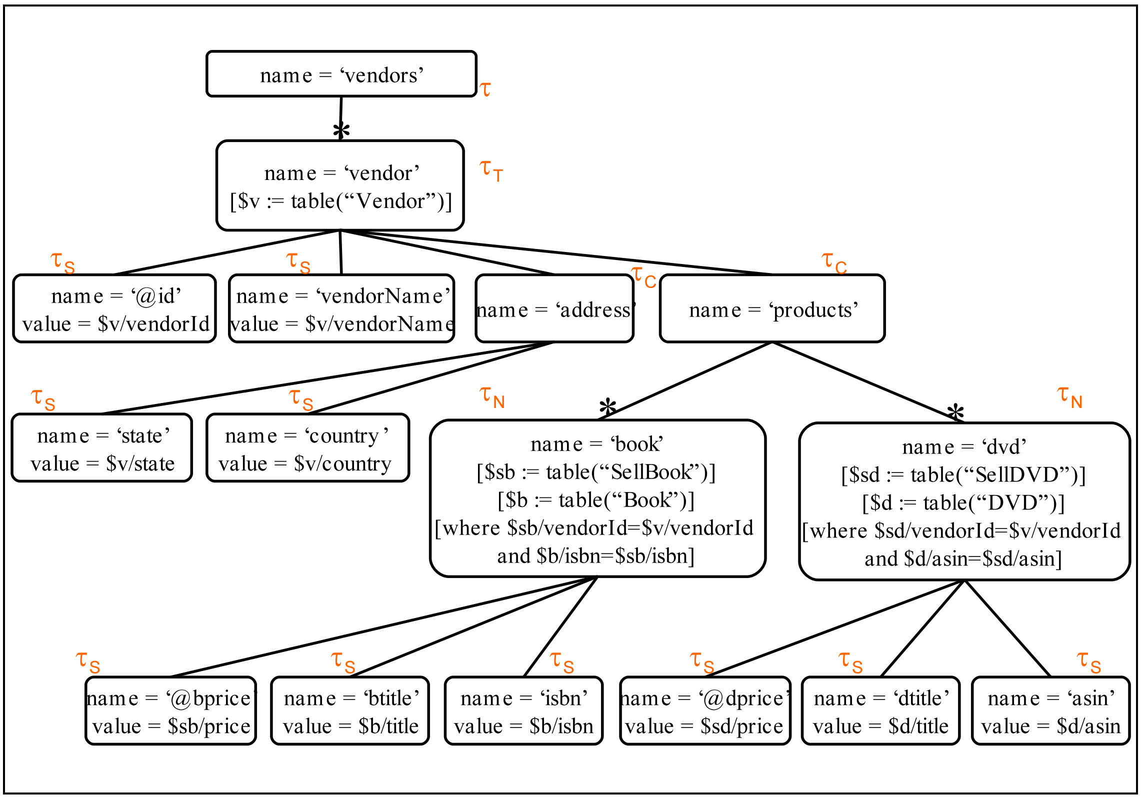


Figure 4.2: Query tree for the XML view of Figure 1.1

for each *vendor*, its *@id*, *vendorName*, *address* and a set of *book*s and *dvd*s within *products*. The root *vendors* has a set of *vendor* child nodes (*-edge). The *vendor* node is annotated with a binding for *$v* (to table Vendor), and has several children at the end of simple edges (*@id*, *vendorName*, and *address*). The value of its *id* attribute is specified by the path *$v/vendorId*, and that of *vendorName* is specified by the path *$v/vendorName*. The node *address* is more complex, and is composed of *state* and *country* subelements.

The node *products* has two *-edge children, *book* and *dvd*. Source annotations of the *book* node include bindings for *$b* (Book) and *$sb* (SellBook) and its where annotations connect tuples in SellBook to tuples in Book, and tuples in SellBook with tuples in Vendor (join conditions). Node *dvd* has source annotations for *$d* (DVD) and *$sd* (SellDVD). Its where annotation connects tuples in SellDVD to tuples in DVD and tuples in SellDVD with tuples in Vendor. The result of this query tree is shown in Figure 1.1.

From now on, we assume that a query tree is *non-empty*, i.e. that it consists of more than a root node.

4.1.2 Abstract Types

In our mapping strategy, it will be important to recognize nodes that play certain roles in a query tree. In particular, we identify five abstract types of nodes: τ, τ_T, τ_N, τ_C and τ_S. We call them *abstract types* to distinguish them from the type or DTD of the XML view elements.

Nodes in the query tree are assigned abstract types as follows:

1. The root has abstract type τ.

2. Each leaf has abstract type τ_S (Simple).
3. Each non-leaf node with an incoming simple edge has abstract type τ_C (Complex).
4. Each starred node which is either a leaf node or whose subtree has only simple edges has an abstract type of τ_N (Nested).
5. All other starred nodes have abstract type τ_T (Tree).

Note that each node has exactly one type unless it is a starred leaf node, in which case it has types τ_S and τ_N.

As an example of this abstract typing, consider the query tree in Figure 4.2, which shows the type of each of its nodes. Since *book* and *dvd* are repeating nodes whose descendants are non-repeating nodes, their types are τ_N rather than τ_T.

The motivation behind abstract types is as follows. To map updates in the XML view to updates in the underlying relational database, we must be able to identify a mapping from the column of a tuple in the relational database to an element or attribute in the XML view. Ideally, this mapping is 1:1, i.e. each attribute of a tuple occurs at most once in the XML view and can therefore be updated without introducing side-effects into the view. In general, however, it may be a 1:n mapping. The class of views allowed by our query trees and its associated abstract type views captures this mapping intrinsically.

Specifically:

- τ_T/τ_N identifies potential tuples in the underlying relational database. Nodes of type τ_T/τ_N are mapped to tuples, and the node itself serves as a tuple delimiter. A node of type τ_T may have children of type τ_T, i.e. nesting is allowed.
- τ_S identifies relational attributes (columns). A node of type τ_S must have a node of type τ_T or τ_N as its ancestor. Starred leaf nodes are an exception to this rule: they need not to have such ancestor.
- τ_C identifies complex XML elements. Since they do not carry a value, this type of node is not mapped to anything in the relational model. Nodes of type τ_C are present in our model to allow more flexible XML views, but are not important in the mapping process.

We call the XML views produced by query trees and their associated abstract types *well-behaved* because, as we will show in the next section, they can be easily mapped to a set of corresponding relational views. However, before turning to the mapping we prove two facts about query trees that will be used throughout the thesis.

PROPOSITION 4.1 *There is at least one τ_N node in the abstract type of a query tree qt.*

Proof: Since query trees are assumed to be non-empty, *qt* must have at least one leaf. This means that *qt* must have at least one starred node n, since the leaf node has a value which involves at least one variable which must be defined in some source annotation attached to a starred node. Since the tree is finite, at least one of these starred nodes is either a leaf node or has a subtree of simple edges, i.e. the starred node is a τ_N node. ■

PROPOSITION 4.2 *There is at most one τ_N node along any path from a leaf to the root in the abstract type of a query tree qt.*

Proof: Suppose there are two τ_N nodes, n_1 and n_2, along the path from some leaf to the root of qt. Without loss of generality, assume that n_1 is the ancestor of n_2. By definition of τ_N, n_2 must be a starred node. Therefore n_1 has a *-edge in its subtree, a contradiction. ■

We will refer to the abstract type of an element by the abstract type that was used to generate it followed by the element name. As an example, the abstract type of the element *dvd* in Figure 4.2 is referred to as τ_N*(dvd)*, and its type (DTD) is *<!ELEMENT dvd (dtitle, asin)>*.

4.1.3 Semantics of Query Trees

The semantics of a query tree follows the abstract type of its nodes, and can be found in Algorithm 4.1. The algorithm constructs the XML view resulting from a query tree *qt* recursively, and starts with n being the root of the query tree. The basic idea is that the source and where annotations in each starred node n are evaluated in the database instance d, producing a set of tuples. The algorithm then iterates over these tuples, generating one element corresponding to n in the output for each of these tuples and evaluating the children of n once for each tuple.

The *bindings{}* hash array keeps the values of variables, taken from the underlying relational database. We assume that values in *bindings{}* are represented as *$x/A = 1, $x/B = 2*, where $x is a variable bound to a relational table T, A and B are the attributes of T and 1 and 2 are the values of attributes A and B in the current tuple of T.

4.1.4 DTD of a Query Tree

Query tree views defined over a relational database have a well-defined schema (DTD) that is easily derived from the tree. Given a query tree, its DTD is generated as follows:

1. For each attribute leaf node named @A with parent named E, create an attribute declaration
<!ATTLIST E @A CDATA #REQUIRED>
2. For each non-attribute leaf node named E, create an element declaration <!ELEMENT E (#PCDATA)>
3. For each non-leaf node named E, create an element declaration
<!ELEMENT E ($E_1, \ldots, E_k$)>, where $E_1, ..., E_k$ are non-attribute child nodes of E connected by a simple or starred edge. In case E_i is connected to E by a starred edge, add a "*" after E_i. In case $k = 0$, then create an element declaration <!ELEMENT E EMPTY>

As an example, the DTD of the view produced by the query tree shown in Figure 4.2 is:

```
<!ELEMENT vendors (vendor*)>
<!ELEMENT vendor (vendorName, address, products)>
<!ATTLIST vendor id CDATA #REQUIRED>
<!ELEMENT vendorName (#PCDATA)>
```

```
eval($qt$, $d$)
{$qt$ is the query tree and $d$ is the database instance}
evaluate(root($qt$), $d$)

evaluate($n$ ,$d$)
{Assume a node type has functions abstract_type($n$), name($n$), value($n$), children($n$), sources($n$), and where($n$)
(with the obvious meanings).}
Let bindings{} be a hash array of bindings of variable attributes to values, initially empty.
case abstract_type($n$)
  $\tau|\tau_C$: buildElement($n$, $d$)
  $\tau_T|\tau_N$: table($n$)
  $\tau_S$: print "<name($n$)>value($n$)</name($n$)>"
end case

buildElement($n$, $d$)
let $tag$ = "name($n$)"
for each attribute $c$ in children($n$) do
  add "name($c$) = value($c$)"to $tag$
end for
print "< $tag$ >"
for each non-attribute $c$ in children($n$) do
  evaluate($c$, $d$)
end for
print "</name($n$)>"

table($n$)
let $w$ be a list of conditions in where($n$)
for each $w[i]$ do
  if $w[i]$ involves a variable $v$ in bindings{} then
    substitute the value binding{$v$} for $v$
  end if
end for
calculate the set $B$ of all bindings for variables in sources($n$) that makes the conjunction of the modified $w[i]$'s
true, using $d$
for each $b$ in $B$ do
  add $b$ to bindings{}
  buildElement($n$)
  remove $b$ from bindings{}
end for
```

Algorithm 4.1: The *eval* algorithm

```
<!ELEMENT address (state, country)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT products (book*,dvd*)>
<!ELEMENT book (btitle, isbn)>
<!ATTLIST book bprice CDATA #REQUIRED>
<!ELEMENT btitle (#PCDATA)>
<!ELEMENT isbn (#PCDATA)>
<!ELEMENT dvd (dtitle, asin)>
<!ATTLIST dvd dprice CDATA #REQUIRED>
<!ELEMENT asin (#PCDATA)>
<!ELEMENT dtitle (#PCDATA)>
```

Note that all (#PCDATA) elements are required. When the value of a relational attribute is *null*, we produce an element with a distinguished *null* value. This makes it easier for the user to distinguish between a value which is not known (*null*) and a value which is known to be the empty string.

We could also have chosen to omit the element tag when the value of that element is *null*. However, we feel our choice of using a distinguished *null* value has several advantages over this option. First, it facilitates modifying a *null* value to some other value: if tag t is omitted from the view, the user must understand the view definition to know whether or not an element with tag t can be added at a particular point in the XML view. Second, it makes our update translation easier: If modifying a *null* value required inserting a new tag in the view, then this insertion in the view would translate to a modification in the underlying relational database. Similarly, changing from some known value to *null* would require a deletion in the view but would be mapped to a modification in the underlying relational database. Our strategy is to map an update (e.g. insertion, deletion or modification) in the XML view to the same type of update in the underlying relational database.

4.2 Update Language

In this section, we present a simple update language for XML views, and describe how we check for schema conformance after updates.

Although no standard has been established for an XML update language, several proposals have appeared (ABITEBOUL et al., 1997; TATARINOV et al., 2001; BONIFATI et al., 2002; LAUX; MARTIN, 2000). The language described in this section is much simpler than any of these proposals and in some sense can be thought of as an internal form for one of these richer languages (assuming a static[2] translation of updates (BONIFATI; FLESCA; PUGLIESE, 2003)).The simplicity of the language allows us to focus on the key problem we are addressing.

Updates are specified using path expressions to point to a set of target nodes in the XML tree at which the update is to be performed. For insertions and modifications, the update must also specify a Δ containing the new values.

DEFINITION 4.5 (UPDATE OPERATION) *An update operation u is a triple <t, Δ,* ref*>, where t is the type of operation (insert, delete, modify); Δ is the XML tree to be inserted, or (in case of a modification) an atomic value; and* ref *is a simple path expression in XPath (CLARK; DEROSE, 1999) which indicates where the update is to occur.*

DEFINITION 4.6 (UPDATE PATH) *An update path* ref *is of the form $p_1/p_2/.../p_n$ where p_i is either a label l_i or a qualified label $l_i[c_1$ and c_2 and ... $c_m]$. Each p_i is called a* step *of P. Each c_i is a qualification of the form $A = x$, where A is a label and x is an atomic value (string, integer, etc).*

The path expression *ref* is evaluated from the root of the tree and may yield a set of nodes which we call *update points*. In the case of modify, it must evaluate to a set of leaf nodes. We restrict the filters used in *ref* to conjunctions of comparisons of attributes or leaf elements with atomic values, and call the expression resulting from removing filters in *ref* the *unqualified portion* of *ref*. For example, the unqualified portion of `/vendors/vendor[@id="01"]` is `/vendors/vendor`.

DEFINITION 4.7 (VALID UPDATE PATH) *An update path* ref *is* valid *with respect to a query tree qt iff the unqualified portion of* ref *is non-empty when evaluated on qt.*

For example, `/vendors/vendor[@id="01"]/vendorName` is a valid path expression with respect to the query tree of Figure 4.2, since the unqualified path `/vendors/vendor/vendorName` is non-empty when evaluated on that query tree.

[2] Static updates are complex update operations that do not "see"the changes produced by previously applied updates (those that are part of the same complex update).

The semantics of insert is that Δ is inserted as a child of the nodes indicated by *ref*; the semantics of modify is that the atomic value Δ overwrites the values of the leaf nodes indicated by *ref*; and the semantics of a delete is that the subtrees rooted at nodes indicated by *ref* are deleted.

The following examples refer to Figure 1.1:

EXAMPLE 4.1 *To insert a new book selling for $38 under the vendor with id="01" we specify:*
t = insert,
ref = `/vendors/vendor[@id="01"]/products`,

```
Δ = {<book bprice = "38">
         <btitle>New Book</btitle><isbn>9999</isbn>
      </book>}.
```

EXAMPLE 4.2 *To change the* vendorName *of the vendor with id = "01" to* Amazon.com *we specify:*
t = modify,
ref = `/vendors/vendor[@id="01"]/vendorName`,
Δ = *{Amazon.com}.*

EXAMPLE 4.3 *To delete all books with title "Computer Networks" we specify:*
t= delete,
ref = `/vendors/vendor/products/book[btitle="Computer Networks"]`.

4.2.1 Schema conformance

Note that not all insertions and deletions make sense since the resulting XML view may not conform to the DTD of the query tree (for details, see Section 4.1.4). For example, the deletion specified by the path `/vendors/vendor/vendorName` would not conform to the DTD of Figure 4.2 since *vendorName* is a required subelement of *vendor*. We must also check that Δ's inserted and subtrees deleted are correct.

DEFINITION 4.8 (CORRECTNESS OF UPDATE OPERATION) *An update* <t,Δ,ref> *against an XML view specified by a query tree qt is* correct *iff*

- ref *is valid with respect to qt, according to definition 4.7;*
- *if t is a modification, then the unqualified portion of* ref *evaluated on qt arrives at a node whose abstract type is* τ_S*;*
- *if t is an insertion, then the unqualified portion of* ref + *the root of* Δ *evaluated on qt arrives at a node whose incoming edge is starred (equivalently, its abstract type is* τ_T *or* τ_N*);*
- *if t is a deletion, then the unqualified portion of* ref *evaluated on qt arrives at a node whose incoming edge is starred;*
- *if nonempty, then* Δ *conforms to the DTD of the element arrived at by* ref.

For example, the deletion of example 4.3 is correct since *book* is a starred subelement of *products*. However, the deletion specified by the update path `/vendors/vendor/vendorName` is not correct since *vendorName* is of abstract type τ_S. Additionally, the deletion specified by the invalid update path `/vendors/vendor/dvd` is also incorrect.

As another example, the insertion of example 4.1 is correct since *book* (arrived at by `/vendors/vendor/products`) is a starred subelement of *products*, the DTD for *book* is <!ELEMENT book (btitle, isbn)>, and Δ conforms to this DTD. However, the following insertion would not be correct for the update path `/vendors/vendor[@id="01"]/products` and

Δ = `{<book bprice="38"><rating>Children</rating></book>}`

since the *isbn* and *btitle* subelements are missing, and *book* does not have a *rating* subelement.

Definition 4.6 excludes descendant traversal (`//`) (CLARK; DEROSE, 1999). The reason for excluding such axis traversal is that query trees can have distinct nodes with the same name (although we are only considering query trees with distinct node names in this thesis, the Appendix B shows how to deal with such cases – the solution is to include artificial numbers in the node names). Thus, a path expression containing `//` would possibly lead to ambiguity. The main problem associated with this ambiguity is not identifying the update points, but to check for schema conformance. Suppose a node *book* is being inserted in the XML view, and the update point is specified as `//`. Suppose also that there are two nodes *book* in the query tree, each of them with a different schema. It would not be possible to know which schema portion to use to check for schema conformance in a case like this. Notice that in this case, using a numbering schema to differentiate the nodes would not solve the problem, since the numbering schema is used internally and the user is not aware of that. That is, the user specifies update paths over the original XML view, which does not contain any number associated to the nodes.

4.3 Chapter Remarks

In this chapter we have presented a formalism – *query trees* – to specify XML views over relational databases, together with algorithms that show, given a query tree and a database instance, how the XML view can be constructed. We have chosen to keep query trees as simple as possible, to make them easier to manipulate and understand. However, query trees can be extended to deal with more complicated features such as grouping, aggregations, etc. In Appendix A.1, we show how query trees can be extended to deal with grouping. With this extension, it is possible to cluster tuples that agree with a given value under a single XML element. Please refer to the Appendix A.1 for further details and examples.

Using query trees to define XML views provides us a series of advantages:

- It makes our proposal language independent - any language capable of constructing XML views from relational databases can be used to implement our approach. All that needs to be done is to define how a given view definition query in that language can be mapped to a query tree. We have done this for a subset of XQuery, which we show in details in Chapter 7 [3].

[3] Our implementation considers extended query trees (Appendix A.1).

- Being language independent allows us to disregard specific syntaxes and to focus on the solution of the problem of how to update the underlying database through the views constructed by query trees.

- Query trees provide a formal framework which makes it easier to prove properties about XML views and to reason about updatability (Chapter 6).

We also have shown how XML views can be updated using a simple update language. This update language can also be thought of as an intermediate representation of existing update languages. We have chosen to use this language to simplify the presentation of our ideas. Besides, despite the efforts of the standardization committees (W3C, 2004; ISO, 2004), no standard XML update language is available yet.

The next chapter shows how XML views constructed by query trees can be mapped to a set of corresponding relational views, and how updates over the XML view can be mapped to updates over this set of relational views.

5 FROM XML VIEWS TO RELATIONAL VIEWS

In our approach, updates over an XML view are translated to SQL update statements on a set of corresponding relational view expressions. Existing techniques such as (DAYAL; BERNSTEIN, 1982a; KELLER, 1985; LECHTENBÖRGER, 2003; BANCILHON; SPYRATOS, 1981; TUCHERMAN; FURTADO; CASANOVA, 1983) can then be used to accept, reject or modify the proposed SQL updates.

In order to do so, it is first necessary to map an XML view to a relational view. As we will show later in this chapter, there are cases where a single XML view must be mapped to a *set* of relational views.

5.1 Mapping XML views to Relational Views

In this section, we discuss how an XML view constructed by a query tree is mapped to a set of corresponding relational view expressions. There are two main steps in the mapping process: *map* and *split*. The *map* process maps a query tree with a single τ_N node to a relational view, and the *split* process deals with query trees that have more than one node of type τ_N. It splits the query tree in several *split trees*, so that each of them has a single node of type τ_N. Then, the *map* process can be applied.

We start by showing the *map* process, and then we discuss the *split* process in details.

5.1.1 Map

Given a query tree *qt* with only one τ_N node, the corresponding SQL view statement is generated as follows:

- Join together all tables found in source annotations (called *source tables*) in a given node *n* in *qt*, using the where annotations that correspond to joins on source tables in *n* as inner join conditions. If no such join condition is found then use "true" (e.g. 1=1) as the join condition, resulting in a cartesian product. Call these expressions *source join expressions.*

- Use the hierarchy implied by the query tree to left outer join source join expressions in an ancestor-descendant direction, so that ancestors with no children still appear in the view. The conditions for the outer joins are captured as follows: If node *a* is an ancestor of *n* and a where annotation in *n* specifies a join condition on a table in *n* with a table in *a*, then use this annotation as the join condition for the outer join. Similar to inner joins, if no condition for the outer join is found, then use “true” as the join condition so that if the inner relation is empty, the tuples of the outer will still appear.

- Use the remaining where annotations (the ones that were not used as inner or outer join conditions) in an SQL where-clause and project the values of leaf nodes. The resulting SQL view statement represents an unnested version of the XML view.

According to the above procedure, *source join expressions* are as follows:

```
<source table> AS <source variable> INNER JOIN
<source table> AS <source variable> INNER JOIN ...
ON <inner joincond>
```

The complete SQL expression resulting from the mapping process is:

```
SELECT <leaf value> AS <leaf name>, ...,
       <leaf value> AS <leaf name>
FROM (<source join expression> LEFT JOIN
  <source join expression> ON <outer joincond>) LEFT JOIN ...
WHERE <remaining "where" annotation>  AND ...
     AND <remaining "where" annotation>
```

For example, the relational view corresponding to the query tree in Figure 4.1 is:

```
SELECT b.isbn AS isbn, b.title AS title, sb.price AS price
FROM (Book AS b INNER JOIN SellBook AS sb ON sb.isbn=b.isbn)
WHERE sb.price > 30
```

The mapping algorithm is presented in details by algorithm 5.1. The auxiliary functions used in this algorithm have obvious meanings. The one that is not so obvious is function *variable(n)*. It returns the variable that was used in the value of a leaf node, without the $ symbol. For example, if the value of node n is $\$x/A$, then *variable(n)* returns x. When the parameter is a source annotation s, then the function returns the variable referenced in this source annotation, without the $ (e.g. with $s = \$x$ *in table("X ")*, function *variable(s)* returns x). Function *attribute(n)* returns the relational attribute that was used to specify the value of a leaf node. Using the example of value of leaf node n above, *attribute(n)* returns A.

5.1.2 Split

For a query tree with more than one τ_N node, this process is incorrect. As an example, consider the query tree of Figure 4.2 which has two τ_N nodes (*book* and *dvd*). If we follow the mapping process described above, the tables DVD and Book will be joined, resulting in a cartesian product. In this expression, a book is repeated for each DVD, violating the semantics of the query tree. We must therefore split a query tree into sub-query trees containing exactly one τ_N node each before generating the corresponding relational views. After the splitting process, each sub-query tree produced is mapped to a relational view as explained above.

The splitting process consists in isolating a node n of type τ_N in the query tree qt, and taking its subtree as well as its ancestors and their non-repeating descendants

```
map($qt$[])
Let $sql$[] be an array of strings, initially empty; Let $numberqt$ be the number of split trees in $qt$[]
for $k$ from 1 to $numberqt$ do
  Let $n$ be the node of type $\tau_N$ in $qt[k]$
  $sql[k]$ = "CREATE VIEW "+ "VIEW"+ name($n$) + "AS "
  $sql[k] = sql[k]$ + "SELECT "
  Let $N$ be the list of leaf nodes in $qt[k]$
  for $i$ from 1 to size($N$) do
    get next $n$ in $N$
    if $i > 1$ then
      $sql[k] = sql[k]$ + ","+ variable($n$) + "."+ attribute($n$) + "AS "+ name($n$)
    else
      $sql[k] = sql[k]$ + variable($n$) + "."+ attribute($n$) + "AS "+ name($n$)
    end if
    $i = i + 1$
  end for
  $sql[k] = sql[k]$ + "FROM "; Let $from$ = ""; Let $N$ be the set of starred nodes in $qt[k]$
  Let $tabs$= ""
  for each $n$ in $N$ do
    Let $join$ = ""; Let $S$ be the list of source annotations in $n$; Let $W$ be the list of where annotations in $n$
    while there is an $s$ in $S$ do
      get next $s$ in $S$
      $join = join$ + table($s$) + "AS "+ variable($s$)
      if $i \geq 2$ then
        $join = join$ + "INNER JOIN "
        tabs = tabs + "$"+ variable($s$)
      end if
      delete $s$ from $S$
      Let $count = 0$
      Let $i = 0$
      while i < size($W$) do
        get next $w$ in $W$
        if $w$ is of the form $\$x/A\ op\ \$y/B$ AND $\$x$ is bound to table $X$ by a source annotation $s \in S$ AND $\$y$ is bound to table
        $Y$ by a source annotation $s' \in S$ AND $x$ is in $tabs$ and $y$ is in $tabs$ then
          if $count = 0$ then
            $join = join$ + "ON "+ $x.Aopy.B$
          else
            $join = join$ + "AND "+ $x.Aopy.B$
          end if
          $count = count + 1$
        end if
        delete $w$ from $W$
        $i = i + 1$
      end while
      if $count = 0$ then
        $join = join$ + "ON (1=1) "
      end if$join$ = "("+ $join$ + ")"
    end while
    Let $A$ be the set of starred ancestors of $n$; Let $count = 0$
    if $n$ has a starred ancestor then
      $join$ = "LEFT JOIN "+ $join$
      for $i = 1$ to size($W$) do
        get next $w$ in $W$
        if $w$ is of the form $\$x/B\ op\ \$y/C$ AND (($\$x$ is bound to table $X$ on node $n$ AND $\$y$ is bound to table $Y$ on a node $a$ in
        $A$) OR ($\$x$ is bound to table $X$ on a node $a$ in $A$ AND $\$y$ is bound to table $Y$ on node $n$)) then
          if $count = 0$ then
            $join = join$ + "ON "+ $x.B\ op\ y.C$
          else
            $join = join$ + "AND "+ $x.B\ op\ y.C$
          end if
        end if
        $i = i + 1$; $count = count + 1$
      end for
      if $count = 0$ then
        $join = join$ + "ON (1=1) "
      end if
      $from$ = "("+ $from$ + $join$ + ")"
    end if
  end for
  $sql[k] = sql[k] + from$; Let $W'$ be the set of all where annotations on nodes of $qt[k]$. Let $count = 0$
  for each $w'$ in $W'$ do
    if $w'$ is of the form $\$x/A\ op\ Z$ AND $Z$ is an atomic value then
      if $count = 0$ then
        $sql[k] = sql[k]$ + "WHERE "+ $x.A\ op\ Z$
      else
        $sql[k] = sql[k]$ + "AND "+ $x.A\ op\ Z$
      end if
    end if
  end for
end for
return $sql$[]
```

Algorithm 5.1: The *map* algorithm

```
split($qt$)
Let $qt[]$ be an array of query trees, initially empty
Let $i = 0$
Let $N$ be the set of nodes of type $\tau_N$ in $qt$
for each node $n$ in $N$ do
  inc $i$
  {initialize $t[i]$ with $qt$}
  $qt[i] = qt$
  repeat
    delete from $qt[i]$ all subtrees rooted at a node $z$ of type $\tau_N$, where $z \neq n$
    retype the ancestors of the deleted nodes
  until $n$ is the only node of type $\tau_N$ in $qt[i]$
end for
return $qt[]$
```

Algorithm 5.2: The *split* algorithm

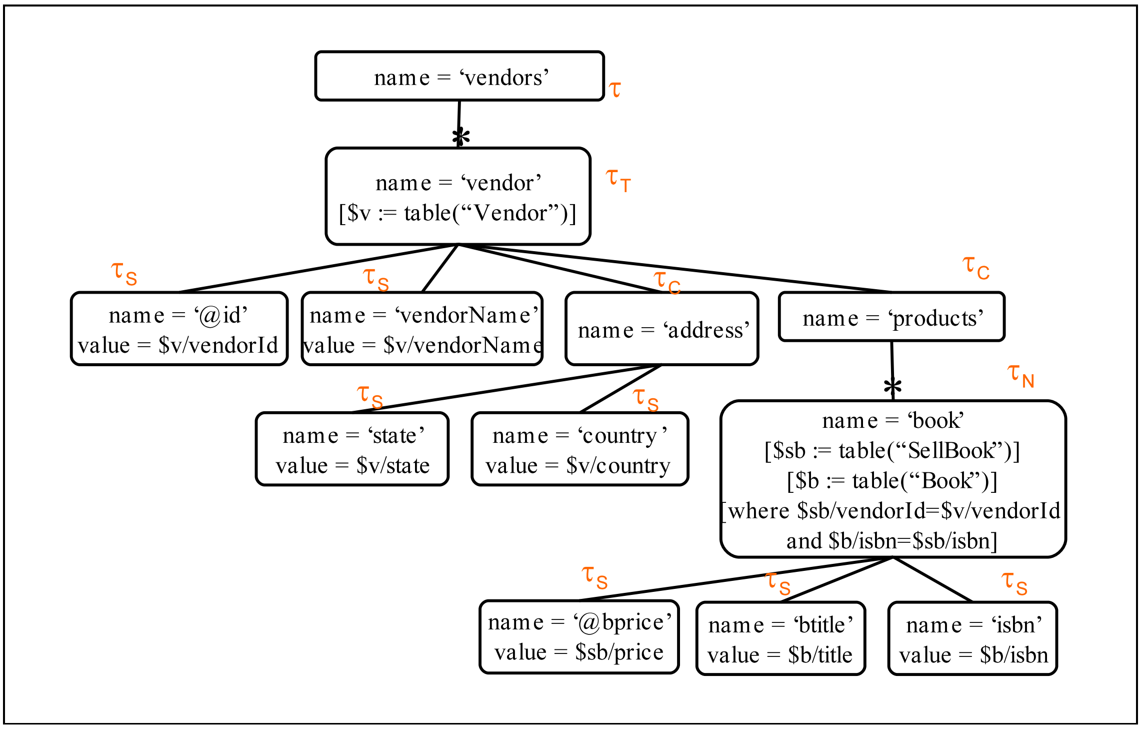


Figure 5.1: Partitioned query tree for $\tau_N(book)$

(types τ_C and τ_S) to form a new tree qt_i. Recall that qt must have at least one τ_N node by Proposition 4.1.

The first step to generate qt_i is to copy qt to qt_i. Then, delete from qt_i all subtrees rooted at nodes of type τ_N, except for the subtree rooted at n. Observe that deleting a subtree r may change the abstract type of the ancestors of r. Specifically, if r has an ancestor a with type τ_T, and r is a's only starred descendant, then the type of a becomes τ_N after the deletion of r. Continue to delete subtrees rooted at nodes of type τ_N in qt_i and retype ancestors until n is the only node of type τ_N in qt_i. The process is repeated for every node of type τ_N in qt and results in exactly one τ_N node per split tree.

Formally, the *split* algorithm (algorithm 5.2) splits a query tree qt, producing one split tree qt_i for each node of type τ_N in qt.

The result of this process for the query tree of Figure 4.2 is shown in Figures 5.1 and 5.2. Using these split trees, the corresponding relational views *ViewBook* and *ViewDVD* are:

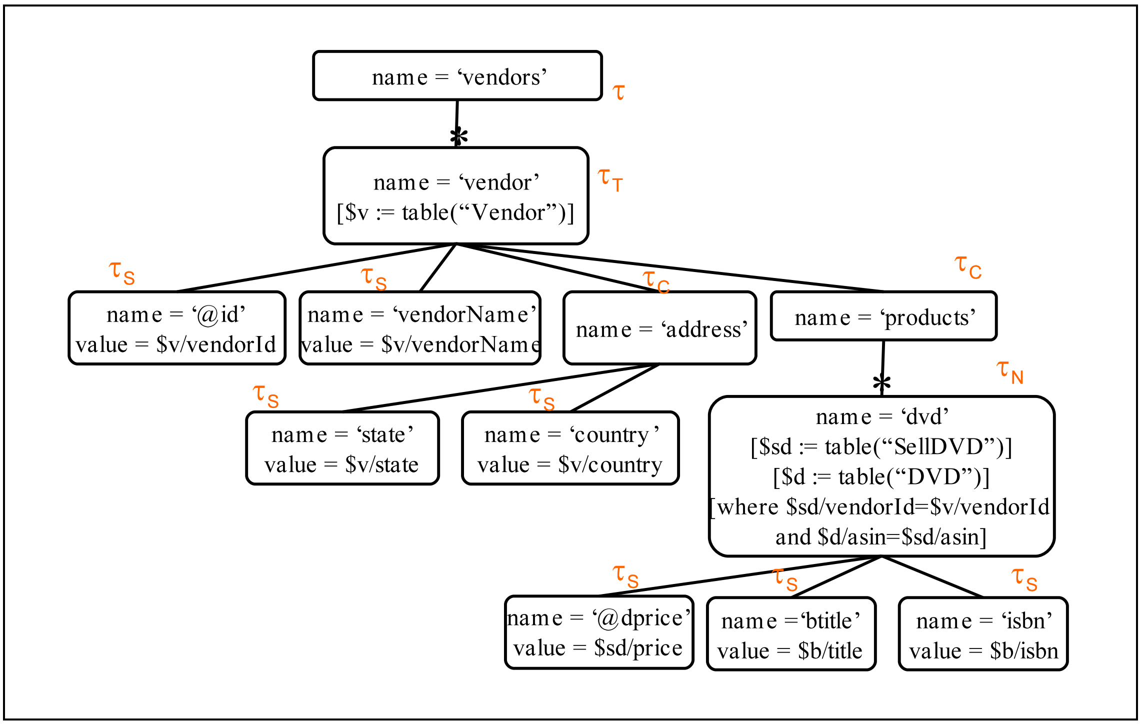


Figure 5.2: Partitioned query tree for $\tau_N(dvd)$

```
CREATE VIEW VIEWBOOK AS
SELECT v.vendorId AS id, v.vendorName AS vendorName,
 v.state AS state, v.country AS country,
 sb.price AS bprice, b.isbn AS isbn, b.title AS btitle
FROM (Vendor AS v LEFT JOIN (SellBook AS sb INNER JOIN
Book AS B ON b.isbn=sb.isbn) ON v.vendorId=sb.vendorId);

CREATE VIEW VIEWDVD AS
SELECT v.vendorId AS id, v.vendorName AS vendorName,
 v.state AS state, v.country AS country,
 sd.price AS dprice, d.asin AS asin, d.title AS dtitle
FROM (Vendor AS v LEFT JOIN (SellDVD AS sd INNER JOIN
DVD AS d ON d.asin=sd.asin) ON v.vendorId=sd.vendorId)
```

As described above, *split* takes as input the original query tree qt and produces as output a set of query trees $\{qt_1, ..., qt_n\}$, each of which has one τ_N node; *map* takes $\{qt_1, ..., qt_n\}$ as input and produces a set of relational view expressions $\{V_1, ..., V_n\}$, where each V_i is produced from qt_i as described above. It follows directly from these algorithms that:

Proposition 5.1 *The number of relational view expressions in* map(split(qt)) *is the number of τ_N nodes in qt.*

In Appendix A.2 we show how the *map* and *split* algorithms can be modified to support extended query trees with grouping capabilities.

5.1.3 Correctness

The correctness of the set of relational view expressions resulting from *map* and *split* can be understood in the following sense: Each tuple in the bindings relations

	id	vendorName	state	country	bprice	btitle	isbn	dprice	dtitle	asin
t_1	1	Amazon	WA	US	38	Unix Network Programming	1111	NULL	NULL	NULL
t_2	1	Amazon	WA	US	29	Computer Networks	2222	NULL	NULL	NULL
t_3	1	Amazon	WA	US	NULL	NULL	NULL	29	Friends	D1111
t_4	2	Barnes and Noble	NY	US	38	Unix Network Programming	1111	NULL	NULL	NULL
t_5	2	Barnes and Noble	NY	US	38	Computer Networks	2222	NULL	NULL	NULL

Figure 5.3: Tuples resulting from *evalRel(eval(qt, d))* for the query tree of Figure 4.2

for the XML view is in one or more instances of the corresponding relational views. To be more precise, we define the following:

Definition 5.1 (Evaluation Schema) *The* evaluation schema *S of a query tree qt is the set of all names of leaf nodes in qt.*

As an example, the *evaluation schema* of the query tree of Figure 4.2 is $S = (id$, $vendorName$, $state$, $country$, $bprice$, $btitle$, $isbn$, $dprice$, $dtitle$, $asin)$.

Definition 5.2 (Evaluation Relation) *Let x be an XML instance of a query tree qt with evaluation schema S, in which the instance nodes are annotated by the query tree type from which they were generated. Let n be the deepest τ_N or τ_T instance nodes for some root to leaf path in x. Let p be the set of nodes in the path from n to the root of x. An* evaluation tuple *of x is created from n by associating the value of each leaf node l that is a descendant of n or of some node in p with the attribute in S corresponding to the name of l, and leaving the value of all other attributes in S null.*

The multi-set[1] *of all evaluation tuples of x is called its* evaluation relation *and is denoted* evalRel(x).

For example, Figure 5.3 shows the result of *evalRel(x)* for the query tree of Figure 4.2 and the XML view of Figure 1.1.

Definition 5.3 (relOuterUnion) *Let $\{V_1, ..., V_n\}$ be defined over a relational schema $\mathcal{D}$, and d be an instance of $\mathcal{D}$. Then* relOuterUnion($\{V_1, ..., V_n\}, d$) *denotes the set of relational instances that result from taking the outer union of the evaluation of each V_i over d:* relOuterUnion($\{V_1, ..., V_n\}$, d) = evalV(V_1, d) $\bigcup$ ... $\bigcup$ evalV(V_n, d)*, where $\bigcup$ denotes outer union, and* evalV(V,d) *instantiates V over d.*

For example, *relOuterUnion*($\{$ *ViewBook*, *ViewDVD*$\}$, d) is the result of the outer union of *evalV*(*ViewBook*, d) and *evalV*(*ViewDVD*, d), which is shown on Figure 5.4. The evaluations *evalV*(*ViewBook*, d) and *evalV*(*ViewDVD*, d) are shown in Figures 5.5 and 5.6, respectively.

The correctness of the set of relational views resulting from *map* and *split* can now be understood in the following sense:

[1] Note that SQL queries may return repeated tuples. Because of that, we can have repeated evaluation tuples in *evalRel*. Thus, *evalRel* is a multi-set instead of a set.

	id	vendorName	state	country	bprice	btitle	isbn	dprice	dtitle	asin
t_1	1	Amazon	WA	US	38	Unix Network Programming	1111	NULL	NULL	NULL
t_2	1	Amazon	WA	US	29	Computer Networks	2222	NULL	NULL	NULL
t_3	2	Barnes and Noble	NY	US	38	Unix Network Programming	1111	NULL	NULL	NULL
t_4	2	Barnes and Noble	NY	US	38	Computer Networks	2222	NULL	NULL	NULL
t_5	1	Amazon	WA	US	NULL	NULL	NULL	29	Friends	D1111
t_6	2	Barnes and Noble	NY	US	NULL	NULL	NULL	NULL	NULL	NULL

Figure 5.4: Tuples resulting from *relOuterUnion({ViewBook, ViewDVD}, d)*

	id	vendorName	state	country	bprice	btitle	isbn
t_1	1	Amazon	WA	US	38	Unix Network Programming	1111
t_2	1	Amazon	WA	US	29	Computer Networks	2222
t_3	2	Barnes and Noble	NY	US	38	Unix Network Programming	1111
t_4	2	Barnes and Noble	NY	US	38	Computer Networks	2222

Figure 5.5: Tuples on *ViewBook*

Theorem 5.1 (Correctness of the mapping process) *Given a query tree qt defined over a database $\mathcal{D}$ and an instance d of $\mathcal{D}$, then* evalRel(eval(qt, d)) $\subseteq$ relOuterUnion(map(split(qt)), d).

Proof: The $\subseteq$ operation needs the two multi-sets being compared to be union compatible. By definition, the schema of *evalRel* is the *evaluation schema S*, which is composed of all leaf node names in qt. The execution of *map(split(qt), d)* results in a set of relational views $\{V_1, ..., V_n\}$. Each view V_i is a schema composed of names of leaf nodes in qt_i (which is produced by *split(qt)*). By definition of *split*, each split tree qt_i contains a single τ_N node n_i: the subtrees rooted at τ_N nodes different from n_i are deleted from qt_i. However, nodes deleted in qt_i are preserved in qt_j, so that each node n in qt is in at least one of the $qt_1, ...qt_n$. Consequently, the schema of $V_1 \bigcup ... \bigcup V_n$ equals S.

Assume t is in *evalRel(eval(qt, d))*, but not in *relOuterUnion(map(split(qt)), d)*. Let x be the XML view resulting from *eval(qt, d)*. Since t is in *evalRel(eval(qt, d))*, it was constructed by taking values from the leaf nodes in a given path p. The path p starts in a node n which is the deepest node of type τ_N or τ_T in a given subtree and goes up to the root of x. If n is of type τ_N, and V_i is the view corresponding to n, then t is in *evalV(V_i,d)*, and consequently, t is in *relOuterUnion(map(split(qt)), d)*, a contradiction. If n is of type τ_T, then the node that originated n in the query tree has at least one node of type τ_N in its subtree. Assume $V_j, ..., V_k$ are the relational views corresponding to those τ_N nodes. Consequently, t is in $V_j \bigcup ... \bigcup V_k$, and thus in *relOuterUnion(map(split(qt)), d)*, a contradiction. ■

Furthermore, the tuples in *relOuterUnion(map(split(qt)), d) − evalRel(eval(qt, d))* represent starred nodes with an empty evaluation (which we call "stubbed" nodes). More precisely:

Definition 5.4 (Stubbed Tuple) *Let x be an XML instance of a query tree qt*

	id	vendorName	state	country	dprice	dtitle	asin
t_1	1	Amazon	WA	US	29	Friends	D1111
t_2	2	Barnes and Noble	NY	US	NULL	NULL	NULL

Figure 5.6: Tuples on *ViewDVD*

	id	vendorName	state	country	bprice	btitle	isbn	dprice	dtitle	asin
t_1	1	Amazon	WA	US	NULL	NULL	NULL	NULL	NULL	NULL
t_2	2	Barnes and Noble	NY	US	NULL	NULL	NULL	NULL	NULL	NULL

Figure 5.7: The *stubs(x)* relation for the XML view x of Figure 1.1

with evaluation schema S, and n be a τ_N or τ_T instance node in x. A stubbed tuple *of x is created from n by associating the value of each leaf node l that is an ancestor of n with the attribute in S corresponding to the name of l, and leaving the value of all other attributes in S null. The set of all stubbed tuples of x is denoted* stubs(x)*.*

As an illustration of a stubbed tuple, consider tuple t_6 in Figure 5.4. Since the XML instance of Figure 1.1 does not have any dvd sold by vendor *Barnes and Noble*, there is a tuple [*2, Barnes and Noble, NY, US, null, null, null*] in *ViewDVD* which was added by the LEFT join. This is correct, since *vendor* is in a common part of the view, so its information appears both in *ViewBook* and *ViewDVD*. However, t_6 is not in Figure 5.3, since when the entire view is evaluated, this vendor joins with a book.

The set *stubs(x)* for the XML view of Figure 1.1 is shown in Figure 5.7.

Theorem 5.2 (Stubs) *Given a query tree qt defined over a database $\mathcal{D}$ and an instance d of $\mathcal{D}$, then every tuple t in* relOuterUnion(map(split(qt)), d) − evalRel(eval (qt, d)) $\subseteq$ stubs(x).

Proof: Tuples in *relOuterUnion(map(split(qt)), d)* that are not in *evalRel(eval(qt, d))* are those resulting from left outer joins with no match in a given relational view $V_i \in$ *map(split(qt), d)*. Since *stubs(x)* contains tuples that has *null*s in attributes related to descendant nodes, and a LEFT JOIN always keeps information of the ancestor, then *relOuterUnion(map(split(qt)), d) − evalRel(eval(qt, d))* $\subseteq$ *stubs(x)*. ■

Note that the statement of correctness is *not* that the XML view can be constructed from instances of the underlying relational views. The reason is that we do not know whether or not keys of relations along the path from τ_N nodes to the root are preserved, and therefore do not have enough information to group tuples from different relational view instances together to reconstruct the XML view. When keys at all levels *are* preserved, then the query tree can be modified to a form in which the variables iterate over the underlying relational views instead of base tables, and used to reconstruct the XML view. We call this algorithm *replace*.

Replace. For query trees that preserve the keys of each source table in the resulting XML view, we can use the corresponding relational views to reconstruct the view. Assuming that *map(qt)* = $\{V_1,...,V_n\}$, the algorithm *replace* replaces references to relational tables in the source and where annotations of *qt* by references to the set of

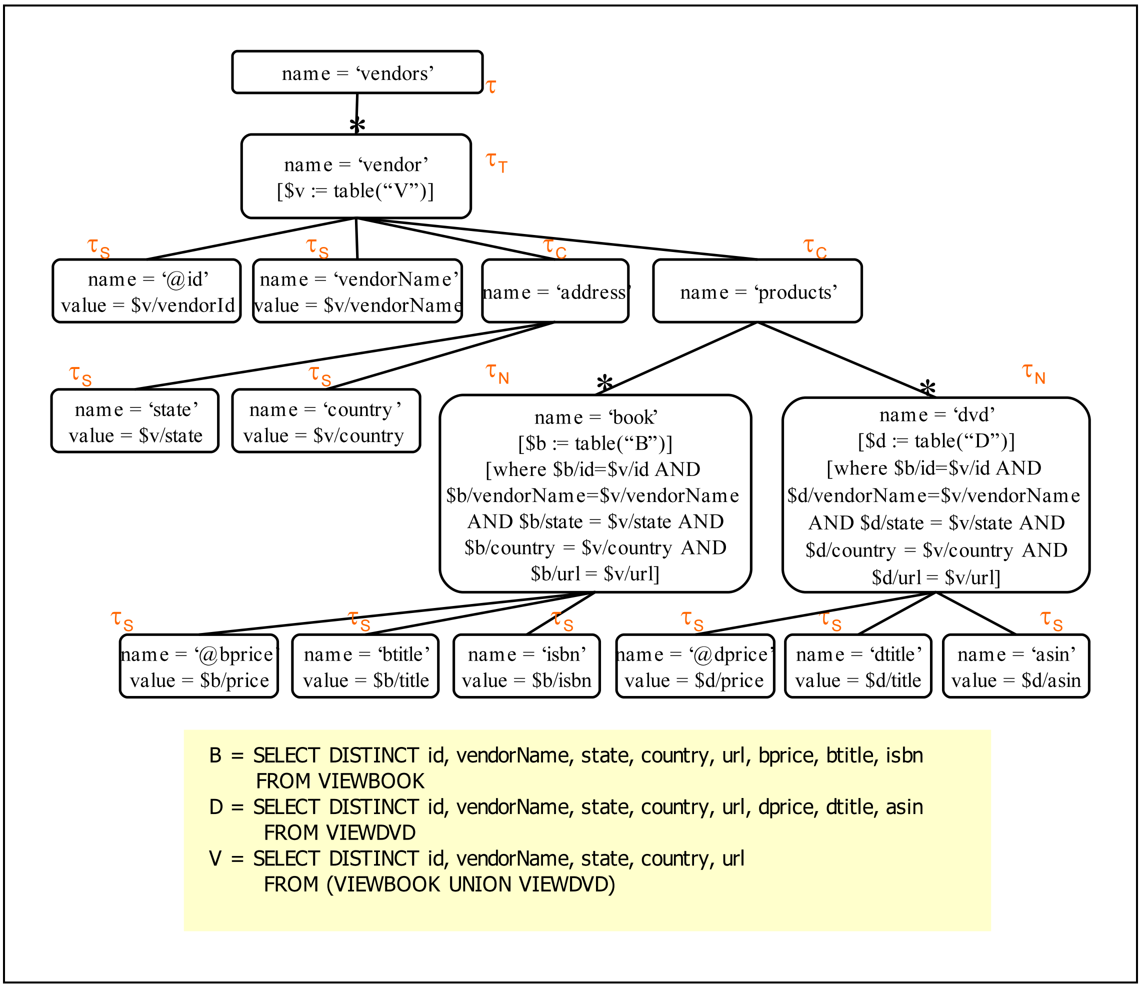
```
replace($qt$)
Let $qt_r = qt$
for each node $n$ of type $\tau_N$ or $\tau_T$ in $qt_r$ do
  remove all source and where annotations from $n$
  if abstract_type($n$) = $\tau_N$ then
    $def$ = createSourceDef($n$,view($n$))
  else
    Let $n_1, ..., n_m$ be the set of nodes with abstract type $\tau_N$ in the subtree rooted at $n$
    $def$ = createSourceDef($n$,(view($n_1$) UNION ... UNION view($n_m$)))
  end if
  Let $s$ be a source annotation
  $s$ = [variable($n$) := Table($X$)], where $X$ is defined as $def$
  Annotate $n$ with $s$
  if $n$ has a starred ancestor $a$ then
    Let $w$ be a where annotation
    Let $W$ be a string
    for each $el$ in $AttsAncestrals(n)$ do
      $W = W + var(a)/el = var(n)/el$
      if $el$ is not the last element in $AttsAncestrals(n)$ then
        $W = W +$ "AND "
      end if
    end for
    Annotate $n$ with $w$ = [where $W$]
  end if
end for
for each node $n$ of type $\tau_S$ in $qt_r$ do
  Let the value of $n$ be of the form $\$x/A$
  Let $a$ be the starred ancestor of $n$ (if $n$ is starred, then $a = n$)
  Replace the value of $n$ by $var(a)/A$
end for
return $qt_r$

createSourceDef($n$, $viewName$)
$def$ = "SELECT DISTINCT " + $AttsAncestrals(n)$ + "," + $Atts(n)$ + "FROM " + $viewName$
return $def$
```

Algorithm 5.3: The *replace* algorithm

relational views V_1,...,V_n. This must be done in a way such that for any instance d of the underlying relational database D, evaluating qt over d is the same as applying *eval* over *replace(qt)* using the evaluation of the relational views V_1,...,V_n produced by $map(split(qt))$.

Before presenting the algorithm, we present some definitions that will be used within this section. We say $view(n)$ is the relational view corresponding to node n, if n is of type τ_N. *Atts(n)* is the set of leaf node names whose values are specified using the variables declared on node n. For example, in Figure 4.2, *Atts(*book*)* = {*bprice, btitle, isbn*}. Notice that we exclude the "@" from attribute names. In the same way, the function *AttsAncestrals(n)* returns a set of leaf nodes whose values are specified using variables declared in the ancestors of node n. As an example, *AttsAncestrals(*book*)* = {*id, vendorName, state, country, url*}. The function *var(n)* returns a unique variable name to be used in node n. Note that every call of *var(n)* for the same node n returns the same variable name. The function *var(*$\$x$*)* finds the node n in which variable $\$x$ was defined, and returns the result of *var(n)*.

The *replace(qt)* algorithm (algorithm 5.3) takes each starred node n in qt and analyzes it. It first removes all source and where annotations from n. Then, it creates a single source annotation that bounds a new unused variable $\$x$ to X, where X is defined over the relational views that carries values for node n. The complete algorithm is shown on algorithm 5.3. It returns a modified query tree qt_r, which references V_1,...,V_n instead of base tables.



```
B = SELECT DISTINCT id, vendorName, state, country, url, bprice, btitle, isbn
      FROM VIEWBOOK
D = SELECT DISTINCT id, vendorName, state, country, url, dprice, dtitle, asin
       FROM VIEWDVD
V = SELECT DISTINCT id, vendorName, state, country, url
       FROM (VIEWBOOK UNION VIEWDVD)
```

Figure 5.8: Modified query tree, resulting of the execution of the *replace* algorithm over the query tree of Figure 4.2

As an example of the result of the *replace* algorithm, the query tree of Figure 4.2 is modified as shown in Figure 5.8. Notice that now the source annotations refers only to the views *ViewBook* and *ViewDVD*. No table from the underlying relational database is used. The result of the evaluation of this query tree is the same of that of Figure 4.2, which is shown in Figure 1.1. Notice also that the values of the leaf nodes now reference the new variables created by the replace algorithm.

5.2 Mapping Updates over XML views to updates over Relational Views

We now discuss how correct updates to an XML view are translated to SQL updates on the corresponding relational views produced in the previous section.

Throughout this section, we will use the XML view of Figure 1.1, produced by the query tree of Figure 4.2 as an example. The relational views *ViewBook* and *ViewDVD* corresponding to this XML view were presented in Section 5.1.2.

The translation algorithm for insertions, deletions and modifications, *translateUpdate*, is given in algorithm 5.4. What it does is to check the type of update operation and call the corresponding algorithm to translate the update. All the three algorithms (*translateInsert*, *translateDelete* and *translateModify*) assume that

```
translateUpdate($x$, $qt$, $u$)
case $u.t$
  $insert$: translateInsert($x$, $qt$, $u.ref$, $u.\Delta$)
  $delete$: translateDelete($x$, $qt$, $u.ref$)
  $modify$: translateModify($x$, $qt$, $u.ref$, $u.\Delta$)
end case
```

Algorithm 5.4: The *translateUpdate* algorithm

```
translateInsert($V$, $qt$, $ref$, $\Delta$)
{Insert $\Delta$ in the XML view $V$ using $ref$ as insertion point. $\Delta$ must be inserted under every node resulting
from the evaluation of $ref$ in $V$. $qt$ is the query tree.} {Assume that view(n) returns the name of the rel. view
associated with node $n$}
Let $p$ be the unqualified portion of $ref$ concatenated with the root of $\Delta$
Let $m$ be the node resulting from the evaluation of $p$ against $qt$
Let $N$ be the set of nodes resulting from the evaluation of $ref$ in $V$
for each $n$ in $N$ do
  if abstract_type($m$) = $\tau_N$ then
    generateInsertSQL(view(m), root($\Delta$), n, V)
  else
    Let $X$ be the set of nodes of abstract type $\tau_N$ in $\Delta$
    for each $x$ in $X$ do
      generateInsertSQL(view(x), x, n, V)
    end for
  end if
end for

generateInsertSQL($RelView$, $r$, $InsertionPoint$, $V$)
{Inserts the subtree rooted at $r$ into $RelView$}
$sql$ = "INSERT INTO"+ $RelView$ + getAttributes($RelView$)
$sql$ = $sql$ + "VALUES ("
for i = 0 to getTotalNumberAttributes($RelView$) - 1 do
  $att$ = getAttribute($RelView$, $i$)
  if $att$ is a child $n$ of $r$ then
    $sql$ = $sql$ + getValue($n$)
  else
    Find $att$ in $V$, starting from $InsertionPoint$ examining the leaf nodes until $V$'s root is found
    Let the node found be $m$
    $sql$ = $sql$ + getValue($m$)
  end if
  if i < getTotalNumberAttributes($RelView$) - 1 then
    $sql$ = $sql$ + ", "
  else
    $sql$ = $sql$ + ")"
  end if
end for
```

Algorithm 5.5: The *tranlateInsert* algorithm

the update specification u was already checked for schema conformance (see Section 4.2.1 for details on how update operations are checked against the view schema).

5.2.1 Insertions

To translate an insert operation on the XML view to the underlying relational views we do the following: First, the unqualified portion of the update path *ref* is used to locate the node in the query tree under which the insertion is to take place. Together with Δ, this will be used to determine which underlying relational views are affected. Second, *ref* is used to query the XML instance and identify the update points. Third, SQL insert statements are generated for each underlying relational view affected using information in Δ as well as information about the labels and values in subtrees rooted along the path from each update point to the root of the XML instance.

Observe that by proposition 4.2, there is at most one node of type τ_N along the path from any node to the root of the query tree and that insertions can never occur below a τ_N node, since all nodes below a τ_N node are of type τ_S or τ_C by definition.

The algorithm *translateInsert* is presented in algorithm 5.5.

For example, to translate the insertion of Example 4.1, we use the unqualified update path `/vendors/vendor/products` on the query tree of Figure 4.2, and find that the type of the update point is $\tau_C(\mathit{products})$. Continuing from $\tau_C(\mathit{products})$ using the structure of Δ, we discover that the only τ_N node in Δ is its root, which is of type $\tau_N(\mathit{book})$. The underlying view affected will therefore be *ViewBook*. We then use the update path *ref*= `/vendors/vendor[@id="01"]/products` to identify update points in the XML document. In this case, there is one node (8). Therefore, a single SQL insert statement against view *ViewBook* will be generated.

To generate the SQL insert statement, we must find values for all attributes in the view. Some of these attribute-value pairs are found in Δ, and others must be taken from the XML instance by traversing the path from each update point to the root and collecting attribute-value pairs from the leaves of trees rooted along this path. In Example 4.1, Δ specifies *bprice="38"*, *btitle="New Book"* and *isbn="9999"*. Along the path from the node 8 to the root in the XML instance of Figure 1.1, we find *id="01"*, *vendorName="Amazon"*, *state="WA"* and *country="US"*. Combining this information, we generate the following SQL insert statement:

```
INSERT INTO VIEWBOOK (id, vendorName, state, country,
  bprice, isbn, btitle)
VALUES ("01","Amazon","WA","US",38,"9999","New Book")
```

As another example, consider the following insertion against the view of Figure 1.1: t = *insert*, *ref* = `/vendors`,

```
Δ={<vendor id="03">
     <vendorName>New Vendor</vendorName>
     <address>
        <state>PA</state>
        <country>US</country>
     </address>
       <products>
        <book bprice="30">
         <btitle>Book 1</btitle><isbn>9111</isbn></book>
        <book bprice="30">
         <btitle>Book 2</btitle><isbn>9222</isbn></book>
        <dvd dprice="30">
         <dtitle>DVD 1</dtitle><asin>D9333</asin></dvd>
       </products>
    </vendor>}.
```

The unqualified update path *ref* evaluated against the query tree of Figure 4.2 yields a node *τ(vendors)*, which is the root. Continuing from here using labels in Δ, we discover two nodes of type τ_N: *τ_N(book)* and *τ_N(dvd)*. We will therefore generate SQL insert statements to *ViewBook* as well as *ViewDVD*.

Evaluating *ref* against the XML instance of Figure 1.1 yields one update point, node 1. Traversing the path from this update point to the root yields no label-value pairs (since the update point is the root itself). We then identify each node of type τ_N in Δ, and generate one insertion for each of them. As an example, traversing the path from the first $\tau_N(\mathit{book})$ node in Δ yields label-value pairs *bprice = "30"*, *btitle = "Book 1"*, and *isbn = "9111"*. Going up to the root of Δ, we have *id =*

```
translateModify($V$, $qt$,$ref$,$\Delta$)
Let $p$ be the unqualified portion of $ref$
Let $m$ be the node resulting from the evaluation of $p$ against $qt$
if abstract_type($m$) = $\tau_N$ then
  $r = m$
else
  Let $r$ be the ancestor of $m$ whose abstract type is $\tau_T$, $\tau_G$ or $\tau_N$
end if
if abstract_type($r$) = $\tau_N$ then
  generateModifySQL(view(r), $\Delta$, ref)
else
  Let $X$ be the set of nodes with abstract type $\tau_N$ under $r$
  for each $x$ in $X$ do
    generateModifySQL(view(x), $\Delta$, ref)
  end for
end if

generateModifySQL($RelView$, $\Delta$, $ref$)
$sql$ = "UPDATE "+ $RelView$ + "SET "
Let $t$ be the terminal node in $ref$
$sql = sql + t$ + "="+ $\Delta$
for each filter $f$ in $ref$ do
  if $f$ is the first filter in $ref$ then
    $sql = sql$ + "WHERE "+ $f$
  else
    $sql = sql$ + "AND "+ $f$
  end if
end for
```

Algorithm 5.6: The *translateModify* algorithm

"03", *vendorName* = *"New Vendor"*, *state* = *"PA"* and *country* = *"US"*. This information is therefore combined to generate the following SQL insert statement:

```
INSERT INTO VIEWBOOK (id, vendorName, state, country,
  bprice, isbn, btitle)
VALUES ("03","New Vendor","PA","US",30,"9111","Book 1");
```

In a similar way, information is collected from the remaining two τ_N nodes in Δ to generate:

```
INSERT INTO VIEWBOOK (id, vendorName,  state, country,
   bprice, isbn, btitle)
VALUES ("03","New Vendor","PA","US",30,"9222","Book 2");
INSERT INTO VIEWDVD (id, vendorName, state, country,
   dprice, asin, dtitle)
VALUES ("03","New Vendor","PA","US",30,"D9333","DVD 1");
```

5.2.2 Modifications

By definition, modifications can only occur at leaf nodes. To process a modification, we do the following: First, we use the unqualified *ref* against the query tree to determine which relational views are to be updated. This is done by looking at the first ancestor of the node specified by *ref* which has type τ_T or τ_N, and finding all nodes of type τ_N in its subtree. (At least one τ_N node must exist, by definition.) If the leaf node that is being modified is of type τ_N itself, then it is guaranteed that the update will be mapped only to the relational view corresponding to this node.

Second, we generate the SQL modify statements. The qualifications in *ref* are combined with the terminal label of *ref* and value specified by Δ to generate an SQL update statement against the view. The corresponding algorithm is presented in algorithm 5.6.

```
translateDelete($V$, $qt$,$ref$)
{Deletes the subtree rooted at $ref$ from $V$}
Let $p$ be the unqualified portion of $ref$ concatenated with the root of $\Delta$
Let $m$ be the node resulting from the evaluation of $p$ against $qt$
if abstract_type($m$) = $\tau_N$ then
  generateDeleteSQL(view(m), ref)
else
  Let $X$ be the set of nodes of abstract type $\tau_N$ under $m$
  for each $x$ in $X$ do
    generateDeleteSQL(view(x), ref)
  end for
end if

generateDeleteSQL($RelView$, $ref$)
sql = "DELETE FROM "+ RelView
for each filter $f$ in $ref$ do
  if $f$ is the first filter in $ref$ then
    sql = sql + + "WHERE "+ f
  else
    sql = sql + "AND "+ f
  end if
end for
```

Algorithm 5.7: The *translateDelete* algorithm

For example, consider the update in Example 4.2. The unqualified *ref* is `/vendors/vendor/vendorName`. The τ_N nodes in the subtree rooted at *vendor* (the first τ_T or τ_N ancestor of *vendorName*) are τ_N(*book*) and τ_N(*dvd*), and we will therefore generate SQL update statements for both *ViewBook* and *ViewDVD*. We then use the qualification *id = "01"* from *ref* = `/vendors/vendor[@id="01"]/vendorName` together with the new value in Δ, to yield the following SQL modify statements:

```
UPDATE VIEWBOOK SET vendorName="Amazon.com" WHERE id="01";
UPDATE VIEWDVD SET vendorName="Amazon.com" WHERE id="01"
```

5.2.3 Deletions

Deletions are very simple to process. First, the unqualified portion of the update path *ref* is used to locate the node in the query tree at which the deletion is to be performed. This is then used to determine which underlying relational views are affected by finding all τ_N nodes in its subtree. Second, SQL delete statements are generated for each underlying relational view affected using the qualifications in *ref*. The corresponding algorithm is presented in algorithm 5.7.

As an example, consider the deletion in Example 4.3. The unqualified update path expression is `/vendors/vendor/products/book`. The only τ_N node in the subtree indicated by this path in the query tree is τ_N*(book)*. This means that the deletion will be performed in *ViewBook*. Examining the update path `/vendors/vendor/products/book[btitle="Computer Networks"]` yields the label-value pair *btitle="Computer Networks"*. Thus the deletion on the XML view is translated to an SQL delete statement as:

```
DELETE FROM VIEWBOOK WHERE btitle="Computer Networks"
```

It is important to notice that if a tuple t in one relation "owns" a set of tuples in another relation via a foreign key constraint (e.g. a vendor "owns" a set of books), then deletions must cascade in the underlying relational schema in order for the deletion of t specified through the XML view to be allowed by the underlying relational system.

5.2.4 Correctness

Since we are not focusing on how updates over relational views are mapped to the underlying relational database, our notion of correctness of the update mappings is their effect on each relational view *treated as a base table.*

Let $x = eval(qt,\ d)$ be the initial XML instance, u be the update as specified in Definition 4.5, and *apply(x, u)* be the updated XML instance resulting from applying u to x. The function *translateUpdate(x, qt, u)* (shown in Section 5.2) translates u to a set of SQL update statements $\{U_{11},\ ...,\ U_{1m_1},\ ...,\ U_{n1},\ ...,\ U_{nm_n}\}$, where each U_{ij} is an update on the underlying view instance $v_i = evalV(V_i{,}d)$ generated by *map(split(qt))*.

We use the notation $v_i' = applyR(v_i,\ \{U_{i1}, ..., U_{im_i}\})$ to denote the application of $\{U_{i1}, ..., U_{im_i}\}$ to v_i, resulting in the updated view v_i'. If the set of updates for a given v_i is empty, then $v_i' = v_i$.

Theorem 5.3 (Correctness of Update Mapping) *Given a query tree qt defined over database $\mathcal{D}$, then for any instance d of $\mathcal{D}$ and correct update u over qt,* evalRel(apply(x, u)) $\subseteq v_1' \bigcup ... \bigcup v_n'$*, where* $\bigcup$ *denotes outer union.*

Proof: Since the update u does not change the view schema, and the application of an update U_{ij} over view v_i also does not change v_i's schema, by theorem 5.1 we have that *evalRel(apply(x, u))* and $v_1' \bigcup ... \bigcup v_n'$ have the same schema (are union compatible).

Insertions Suppose t is a tuple in *evalRel(apply(x, u))*, resulting from a insertion of a subtree in x. Assume t is not in $v_1' \bigcup ... \bigcup v_n'$. Assume update U_{ij} is the translation of u.

Consider a tuple t' which was inserted by update U_{ij} in v_i. Since U_{ij} is the translation of u, t' has the values of one of the subtrees that were inserted in x by u, and also the values of x that were above the update point ref of u. As a consequence, $t = t'$, and t is in $v_1' \bigcup ... \bigcup v_n'$, a contradiction.

The same applies for the insertion of a more complex subtree. It will generate several tuples t_1, ..., t_n to appear in *evalRel(apply(x, u)).* Each of these tuples will be inserted in the relational views by a set of updates $U_{ij}, ..., U_{kl}$. So we have that *evalRel(apply(x, u))* $\subseteq v_1' \bigcup ... \bigcup v_n'$ holds for insertions.

Modifications Suppose t is a tuple in *evalRel(apply(x, u))*, resulting from a modification of a leaf value in x. Assume t is not in $v_1' \bigcup ... \bigcup v_n'$. Assume update U_{ij} is the translation of u.

Consider a tuple t' which was modified by update U_{ij} in v_i. Since U_{ij} is the translation of u, t' had a single attribute modified - the one that was updated in x. As a consequence, $t = t'$, and t is in $v_1' \bigcup ... \bigcup v_n'$, a contradiction.

The same applies for modifications that affect more than one leaf in x, that is, when ref in u evaluates to more than one update point. For every node affected by the modification, will be generated one modification U_{ij}. Since by theorem 5.1 all tuples in *evalRel(x)* are in $v_1 \bigcup ... \bigcup v_n$, then *evalRel(apply(x, u))* $\subseteq v_1' \bigcup ... \bigcup v_n'$.

Deletions Following the inverse reasoning of insertions, every subtree deleted from x makes a tuple to disappear from *evalRel(apply(x, u))*s. Analogously, the translation U_{ij} of u will make that tuple to disappear from $v_1' \bigcup ... \bigcup v_n'$, so *evalRel(apply(x, u))* $\subseteq v_1' \bigcup ... \bigcup v_n'$ holds. ■

Theorem 5.4 (Stubs for Updated View) *Given a query tree qt defined over a database $\mathcal{D}$ and an instance d of $\mathcal{D}$, then* $v'_1 \bigcup ... \bigcup v'_n$ − evalRel(apply(x, u)) ⊆ stubs(apply(x, u)).

Proof: Insertions of uncomplete subtrees or deletions of uncomplete subtrees may cause tuples to be filled in with nulls because of the LEFT JOINS in some v'_i. These tuples, however, will be in *stubs*. The reasoning is the same as in proof of theorem 5.2. ■

Note that a correctness definition like *apply(eval(qt,d), u)* ≡ *eval(qt, d′)*, where d' is the updated relational database state resulting from the application of the translated view updates $\{U_{11}, ..., U_{1m_1}, ..., U_{n1}, ..., U_{nm_n}\}$ to updates on d, does not make sense due to the fact that we do not control the translation of view updates. Therefore we cannot claim that they are side-effect free.

In the next chapter, we discuss a scenario in which this claim can be made.

5.3 Chapter Remarks

In this chapter, we have shown how an XML view can be mapped to a *set* of corresponding relational views, and how updates over the XML views are mapped to this set of relational views. We have also demonstrated the correctness of our mapping process, both for the view and for the updates over the view. The mapping process is published in (BRAGANHOLO; DAVIDSON; HEUSER, 2004a).

We consider this to be the most important contribution of this thesis. We have shown how an open problem – that of updating XML views over relational databases – can be mapped into a well studied problem – that of updating relational views over relational databases. The mapping presented in this chapter allows the use of existing techniques to update relational views (DAYAL; BERNSTEIN, 1982a; KELLER, 1985; LECHTENBÖRGER, 2003; BANCILHON; SPYRATOS, 1981; TUCHERMAN; FURTADO; CASANOVA, 1983) to solve the XML view update problem. Notice that these work can be used both to translate the updates to the underlying relational database and to reason about the updatability of XML views. In the next chapter, we show an updatability study using the approach of Dayal and Bernstein (DAYAL; BERNSTEIN, 1982a).

As mentioned before, the mapping presented in this chapter does not consider query trees which have nodes with the same names. An extension to deal with such cases is shown in Appendix B. We have chosen not to include this extension in the core of the thesis, because it would complicate the understanding of our approach. Now that we have presented the mapping for the trivial case, the reader can refer to the appendix to get more details in how to use XML numbering schemas to avoid problems in the mapping.

6 ON THE UPDATABILITY OF XML VIEWS

In our approach, we have chosen to use the approach of Dayal and Bernstein (DAYAL; BERNSTEIN, 1982a,b, 1978) to translate updates from the relational views to the underlying relational database. Despite of the arguing presented in Chapter 2, we now reinforce our choice:

Besides being one of the few proposals that fully presents translation algorithms, (DAYAL; BERNSTEIN, 1982a) also present clear conditions for their algorithms to work. Specifically, they prove that if certain conditions are satisfied, then the translation found by their algorithm will never cause side-effects on the database [1].

Definition 6.1 (Side-effect Free Update) *Given an XML view defined as a function V over a relational database with schema S, an update u over the view and a translation U of u over S, u is side-effect free if for any instance I of S, $u(V(I)) = V(U(I))$ (KELLER, 1985).*

Finally, their notion of correctness is less restrictive than other existing proposals. Although most of the other existing approaches also adopt the side-effect free criteria, others add more restrictions on the correctness notion. In the view complement approach (BANCILHON; SPYRATOS, 1981), a translation of an update is correct iff it does not touch any portion of data outside the view. We chose not to adopt this definition of correctness because we think it may be hard to the user to understand why a given update translation was incorrect, if he had not seen any erroneous effect on the view. A more recent work (LECHTENBÖRGER, 2003) gives an easier interpretation to this notion: a translation of an update over the view is correct if it can be undone by a user action, that is – if the user can issue another update over the view that makes the database return to its previous state. As an example, consider a view that is defined as a projection over a single base table R. Suppose R has 3 attributes (A, B, C), but only (A, B) are projected in the view. If a user removes a tuple from the view, he cannot undo this action by performing an insertion. This is because the value of C was permanently lost (the user was unaware of its value). We think it is too restrictive to consider such deletion as incorrect. In fact, other authors have also come to the same conclusion (KELLER, 1987; LANGERAK, 1990).

Having reinforced our choice, we now proceed to an updatability study. In Chapter 2, we have shown how Dayal and Bernstein translate updates over the view to updates over the base tables, and how they assure that these translations

[1] In their work, they use the term *exact translation* to denote this situation.

vendorId	vendorName	Books		
1	Amazon	**isbn**	**title**	**price**
		1111	Unix Network Programming	38
		2222	Computer Networks	29
2	Barnes and Noble	**isbn**	**title**	**price**
		1111	Unix Network Programming	38
		2222	Computer Networks	38

Figure 6.1: An instance of NRA1

are side-effect free. In this chapter, we use those results to reason about updatability of XML views.

Our first updatability study was made using the *Nested Relational Algebra* (NRA) (JAESCHKE; SCHEK, 1982; THOMAS; FISCHER, 1986) as the view definition language (BRAGANHOLO; DAVIDSON; HEUSER, 2003a). We then extended the results of this study to reason about the updatability of views constructed by query trees (BRAGANHOLO; DAVIDSON; HEUSER, 2004a). We now summarize the results for the nested relational algebra, and then present the results for query trees.

6.1 Updatability of NRA Views

The nested relational algebra contains the classical relational algebra operators (σ, π, $\cup$, $\times$, $\bowtie$, $-$) as well as the *nest* (ν) and *unnest* (μ) operators. In our study, we identified three subclasses of NRA, each of them with different updatability properties. They are *nest-last views*, *nest-last project-select-join views (NPSJ)* and *well-nested NPSJ views.*

6.1.1 Nest-last XML views

A *nest-last view* is a view defined by a nested relational algebra expression of form $\nu \ldots \nu R$, where R is any relational algebra expression. We claim that this class of views can be treated by considering only the expression R, and that the nesting introduces *sets* of tuples to be inserted, deleted or modified in the underlying relational instance.

Example 6.1 *As an example of such view, consider the following NRA expression (NRA1):*

$$\nu_{\text{Books=(isbn, title, price)}}(\pi_{\text{(vendorId, vendorName, title, isbn, price)}}$$
$$(\sigma_{\text{(Vendor.vendorId = SellBook.vendorId AND Book.isbn = SellBook.isbn)}}$$
$$\text{(Vendor} \times \text{SellBook} \times \text{Book)))}$$

An instance of NRA1, constructed over the database instance of Figure 1.2 is shown in Figure 6.1. The corresponding XML view (using a straightforward mapping) is shown in Figure 6.2.

Proposition 6.1 *Let $\nu \ldots \nu R$ be a nest-last view and u an update over this view. Let $t(u)$ be the translation of u into an update over R. If R is updatable wrt $t(u)$, then $\nu \ldots \nu R$ is updatable wrt to u.*

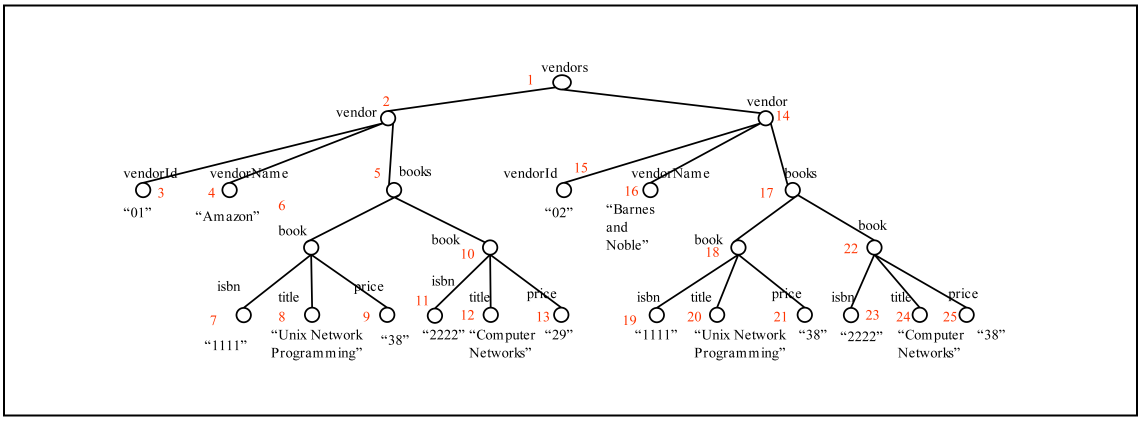


Figure 6.2: NRA1 represented in XML

vendorId	vendorName	isbn	title	price
1	Amazon	1111	Unix Network Programming	38
1	Amazon	2222	Computer Networks	29
2	Barnes and Noble	1111	Unix Network Programming	38
2	Barnes and Noble	2222	Computer Networks	38

Figure 6.3: NRA2: Tuples resulting from unnesting NRA1

Proof: The proof is based on the fact that the nest (ν) operator is invertible (JAESCHKE; SCHEK, 1982; THOMAS; FISCHER, 1986). That is, after a nest operation, it is always possible to obtain the original relation by applying an unnest (μ) operation. Since in this type of view the nest operation is always the last operation to be applied, we can apply a reverse sequence of unnest operators to obtain the (flat) relational expression. ■

As an example, by unnesting on *Books* in the view of Example 6.1, we would obtain a flat relational expression (NRA2):

$$\pi_{(vendorId,\ vendorName,\ isbn,\ title,\ price)}$$
$$(\sigma_{(Vendor.vendorId\ =\ SellBook.vendorId\ AND\ Book.isbn\ =\ SellBook.isbn)}$$
$$(Vendor \times SellBook \times Book))$$

The tuples resulting from this expression are shown in Figure 6.3.

Proposition 6.1 reduces the problem of investigating updatability of XML views to the problem of updates through relational views. That's why we are able to use existing work on updates through relational views for XML views of this class.

For XML views constructed with NRA expressions, we consider the same Update Language of Section 4.2. The mapping to updates over relational views is similar. The difference is that it is not necessary to check to which view the update must be mapped to, since an XML view constructed by the NRA is always mapped to a single corresponding relational view.

As an example of update operation, consider the update u, where ref = `/vendor/vendor[vendorId="01"]/books/book[isbn="1111"]/title`, $\Delta = \{$new title$\}$ and t = *modification*. This modification will be translated to NRA1 as follows:

```
UPDATE NRA1
SET title = "new title"
WHERE vendorId = "01" AND isbn = "1111".
```

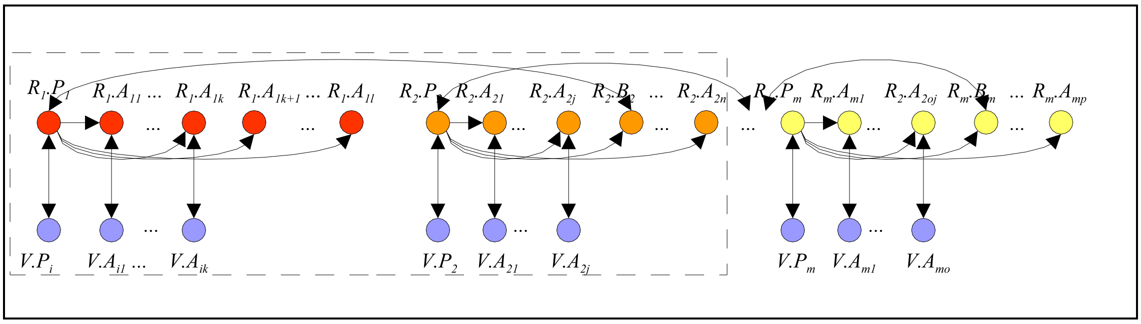


Figure 6.4: View graph

6.1.2 Nest-last Project-Select-Join Views

We now investigate a special subset of nest-last views that are well behaved with respect to updates.

Definition 6.2 (Nest-last Project-Select-Join View) *A* nest-last project-select-join view *(NPSJ) is a nest-last view with the following restrictions: the relational expression is a project-select-join; the keys of the base relations are not projected out; and joins are made only through foreign keys.*

Lemma 6.1 (Updatability of NPSJ Views) *NPSJ views are always updatable for insertions.*

Proof Sketch: Proposition 6.1 shows how to reduce an XML view to a relational view. Based on this result, we are now able to use the technique of Dayal and Bernstein (DAYAL; BERNSTEIN, 1982a) to prove that there is always an exact translation for insertions for NPSJ views. Since the nest can be ignored, we start by defining a general PSJ view that is the join of relations $R_1, R_2, \ldots, R_m$, where the keys of $R_1, R_2, \ldots, R_m$ are preserved in the view and joins are done over foreign keys. We then draw a view graph for this view, as illustrated in Figure 6.4. Nodes in this graph represent attributes. The upper nodes represent attributes of the base relations, and the lower ones represent view attributes. Primary keys are represented as Ps and foreign keys as Bs. Edges represent functional dependencies between attributes in the relations, join conditions or the derivation of view attributes. The proof for insertions are based on finding paths in this directed graph, as discussed in Section 2.2.1.3.

Insertions. For insertions, (DAYAL; BERNSTEIN, 1982b) divides the problem into smaller ones. They claim that insertions are always exactly translatable if we can express the view definition as a sequence of views definitions, each one defined over only two relations, say R and S. In the case of NPSJ, this is obviously true. The additional conditions are: (i) the two relations must be equijoined over foreign keys (true by definition of NPSJ); (ii) and there must be a path from the attributes of R to all view attributes that originated from these two relations (R and S); (iii) and relation R contains a foreign key to S, which was used to join R and S. Conditions (ii) and (iii) are also obviously true. By looking at the graph of Figure 6.4, it is easy to see that the relation containing the foreign key has always the path required in (ii). As an example, consider the two relations inside the dotted box in Figure 6.4. There is a path from the attributes of R_2 to all attributes in the view that originated from R_1 or R_2. In this proof, R_2 corresponds to R, and R_1 corresponds to S. ■

vendorId	vendorName	Deposits			
1	Amazon	**depId**	**address**	**city**	**state**
		D1	1245, Bourbom Street	Seatle	WA
		D3	4545, 15th Avenue	Seatle	WA
2	Barnes and Noble	**depId**	**address**	**city**	**state**
		D2	1478, 25th Avenue	New York	NY

Figure 6.5: An instance of NRA3

For modifications and deletions, even in the relational case there may fail to be an exact translation for certain types of updates over a PSJ view. This type of update attempts to change (or delete) some but not all occurrences of data that is repeated in the view, and thus causes side effects. As an example, consider NRA2, the unnested version of the view 6.1. This view has the values of *vendorId* and *vendorName* repeated in several tuples. An attempt to modify a vendor name could be stated as

```
UPDATE NRA1
SET vendorName = "New Name"  WHERE vendorId= "01".
```

This is exact, since it modifies all occurrences of tuples with that vendor name (notice that *vendorId* is the key of the *Vendor* relation). However, consider this same example with a slight modification.

```
UPDATE NRA1 SET vendorName= "New Name"
WHERE vendorId= "01" AND isbn="1111"
```

As one can easily see, there is no way to translate this request without causing side effects, because tuples that do not satisfy the qualification of this modification request would also be affected (more specifically, tuples with $vendorId =$ "01" and $isbn \neq$ "1111"). The same problem happens for deletions.

6.1.3 Well-nested NPSJ

Fortunately, proper application of the nest operator can be used to avoid this type of ambiguity. For example, for the view of Example 6.2 (NRA3) this kind of bad modification (or deletion) request cannot happen.

Example 6.2 *(NRA3):*

$$\nu_{\text{Deposits=(depId, addrees, city, state)}}(\pi_{\text{(vendorId, vendorName, depId, address, city, state)}}$$
$$(\sigma_{\text{(Vendor.vendorId = Deposit.vendorId)}}$$
$$(\text{Vendor} \times \text{Deposit})))$$

An instance of NRA3, is shown in Figure 6.5.

However, if we had nested this view in a different way, the same update would fail to be exact. As an example, consider the same view, now nested by {*vendorId, vendorName*} instead of {*depId, address, city, state*}. The same *vendorId* and *vendorName* appear several times in the view, as in the relational case. Thus, not all modifications and deletions over this view would be exactly translatable.

The updatability of NPSJ views with respect to modifications and deletions depends on the way in which we traverse the foreign key constraints when nesting.

In view NRA3, we traverse the foreign key constraint from 1 to n. That is, for each *Vendor* tuple there are many *Deposit* tuples, so we nest *Deposit* tuples (the n's) under their corresponding *Vendor* tuple (the 1's). In the second example (where we nested over {*vendorId, vendorName*}), we nested the 1's under the n's, causing the 1's to appear several times in the resulting view.

To define when a NPSJ view is well-nested, we reason about the foreign keys of the underlying relations. Recall that the syntax of a foreign key constraint C on table R_1 is given by C_{R_1} FOREIGN KEY $(FK_1, \ldots, FK_n)$ REFERENCES R_2 (K_1, ..., K_n). When the attribute names (K_1, ..., K_n) are the same as $(FK_1, \ldots, FK_n)$, they can be omitted, as in the example of Figure 1.2.

Definition 6.3 (Ambiguity Eliminating Nest) *Let C_{R_1} be a foreign key constraint, and $V(R_1)$ be the set of attributes of R_1 that appear in the view. An* ambiguity eliminating nest *with respect to C_{R_1} is a nest of the form $\nu_{X=(D)}$, where $D = \{V(R_1)\} - \cup_i FK_i$.*

The idea behind this definition is that by omitting the foreign keys of R_1 and the keys of R_2 in the nest, we collect their values together thus eliminating ambiguity. That is, each value appears just once in the view.

The view of Example 6.2 (NRA3) has an ambiguity eliminating nest since $R_1 =$ *Deposit*, $R_2 =$ *Vendor*, $FK = \{$*vendorId*$\}$, $V(R_1) = \{$*depId, vendorId, address, city, state*$\}$ and we are nesting over $\nu_{Deposits\ =\ (depId,\ address,\ city,\ state)}$.

Definition 6.4 (G_F Graph) *Given a set of relations $\mathcal{R} = \{R_1, ..., R_n\}$, and a set $\mathcal{F}$ of foreign key constraints involving relations of $\mathcal{R}$, we can build a directed graph of foreign keys G_F as follows:*

1. *For each table R in $\mathcal{R}$, add a node in G_F.*
2. *For each FK in R_i of type "C_{R_i} FOREIGN KEY $(FK_1, \ldots, FK_n)$ REFERENCES R_s (K_1, ..., K_n)" in $\mathcal{F}$, add an arc from R_s to R_i (R_s,R_i).*

Definition 6.5 (Nest-Path) *A* nest-path *over G_F is a path of arrows in G_F, constructed as follows:*

1. *Start with a node R_i which has only outgoing edges in G_F;*
2. *Choose one of the non-visited outgoing edges of R_i and traverse this edge, reaching node R_s. Include edge (R_i, R_s) in the* nest-path. *Mark (R_i, R_s) as visited.*
3. *Continue traversing nodes, starting from R_s. If there is no more non-visited outgoing edges in R_s, go to step 1. Repeat this until all edges are marked as visited, or abort in case a node with more than one incoming edge is found.*

By definition, the graph G_F has no nest-path *if at least one node R_i in G_F has more than one incoming edge.*

As an example, for the view NRA3, the graph G_F is shown in Figure 6.6. The *nest-path* for this graph is (*Deposit*, *Vendor*).

Definition 6.6 (Well-nested NPSJ View) *A NPSJ view that involves two or more base relations $\{R_1, ..., R_n\}$ is* well nested *if*

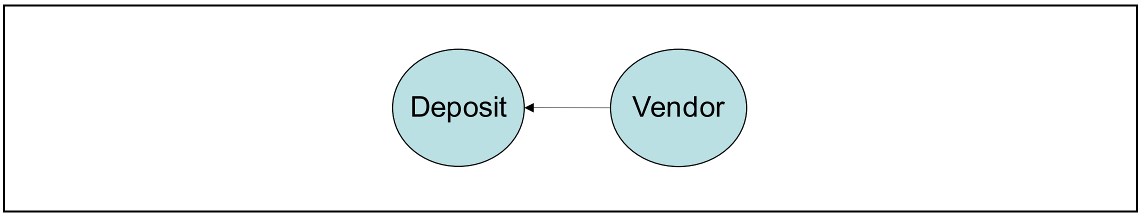


Figure 6.6: Graph G_F for view NRA3

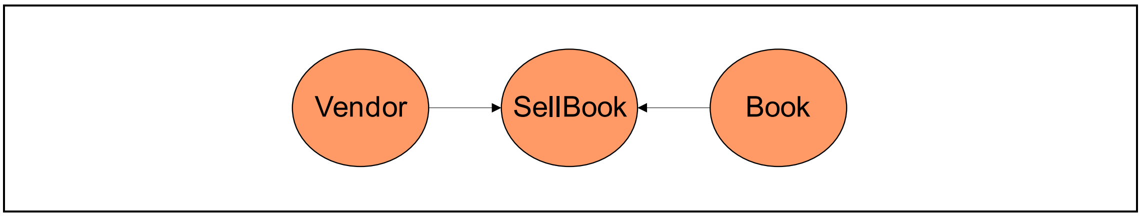


Figure 6.7: Graph G_F for view NRA1

1. *It has one ambiguity eliminating nest for each foreign key constraint that was used to join the base relations; and*
2. *There is a* nest-path *in the graph G_F constructed over $\{R_1, ..., R_n\}$; and*
3. *The ambiguity eliminating nests are executed in the order given by the pair of relations in the* nest-path.

Notice that for the view of Example 6.1 (NRA1), it is not possible to find an order of nests which makes NRA1 *well-nested*. This is because table *SellBook* implements an n:n relationship of *Vendor* and *Book*, and there is no way of finding a *nest-path* in G_F over {*Vendor, Book, SellBook*} (Figure 6.7 shows the graph G_F for view NRA1). It is important to notice that for views with more than two source relations, there may be more than one *nest-path*. This is no problem, as long as one of them is chosen to order the nests in the NRA expression.

An example of well-nested NPSJ is the view of Example 6.2.

Lemma 6.2 (Updatability of Well-nested NPSJ Views) *Well-nested NPSJ views are always updatable with respect to modifications and deletions.*

Proof Sketch: We divide the proof in two steps.

Modifications. In order to simplify the proof, we consider a view defined over two base relations, say $R_1 \bowtie R_2$. The graph of this view corresponds to the dotted box of Figure 6.4. Using the technique of (DAYAL; BERNSTEIN, 1982a), The first condition states that there must be a path from the attributes of the relation whose attributes are being modified to all view attributes that were specified in the WHERE clause. In the case of well-nested NPSJ views, this is directly related to how we specify the update against the relational view. In order for R_1 and R_2 to be well-nested, R_2 must be nested under R_1. If we want to modify an attribute from R_1, the WHERE clause will have only attributes generated from R_1. Obviously, there is a path from the attributes in R_1 to the view attributes generated from R_1. If we want to modify attributes from R_2, the WHERE clause will have attributes generated both from R_1 and R_2. Since it is possible to use the arrow $R_1.P_1$-$R_2.B_2$ to reach all the view attributes, the condition is satisfied.

The second condition states that if we are modifying a join attribute A, then there must be a path from the attributes of the relation to which A belongs, to all view attributes. Suppose B is a foreign key in table S referring to A on table R. Since joins are made through keys and foreign keys, keys are kept in the view, and we consider views are well-nested, then it is guaranteed that A is in the view, and B is not. Thus, modifying a join attribute implies in modifying a primary key[2]. Since we can modify any primary key of any of the relational tables in the view, we would need to require the existence of paths from all relational tables in the view, to all view attributes. Since this is too restrictive (for instance, there is not a path from the attributes of table *Vendor* to all attributes of the view of Figure 6.5), we decided not to allow modifications of primary keys, so that well-nested NSPJ are updatable for all possible modifications, except those in which a primary key is modified.

The proof can be easily generalized to views defined over more than two base relations.

Deletions. Deletions have a WHERE clause that specifies conditions that view tuples must satisfy in order to be deleted. The condition for exact translation for deletions says that there must be a path in the view graph from the relation chosen to translate this deletion to all attributes specified in the WHERE clause. Our proof supposes that all attributes of the view were specified in the WHERE clause, since this is the "worst case". It is easy to see that one can always choose the last relation joined to translate the deletion to the database because there is always a path from the attributes on this relation to all view attributes (see R_m in Figure 6.4) due to the edges introduced by join conditions. ■

Notice that despite we decided against the modifications of primary keys to preserve updatability, the modification of a foreign key can be achieved by simply deleting a subtree and inserting it under a different parent. For example, deleting a deposit from the view of Figure 6.5 and inserting it under a new vendor.

6.2 Updatability of Query Tree Views

In the previous section, we define conditions under which XML views constructed by "nest-last" nested relational algebra (NRA) expressions are updatable. Since query trees also express nesting and are mapped to a *set* of corresponding relational views, we can use these results to reason about the updatability of XML views constructed by query trees. We assume the underlying relational database is in BCNF (to be able to automatically determine the functional dependencies), and impose three restrictions on the query tree and updates: (1) each table must be bound to at most one variable; (2) each value in a leaf node must be unique, that is, if the value of n is specified as $\$x/A$, then this value specification does not appear on any other node in the query tree; (3) comparisons on $u.ref$ must be conjunctions of equalities. These restrictions are imposed so that the resulting relational views do not include joins of the same tables, and projections of the same attribute (as required by (DAYAL; BERNSTEIN, 1982a)). The restrictions on equalities are also required by (DAYAL; BERNSTEIN, 1982a).

[2] By definition of foreign key in the relational model, if B is a foreign key referring to A, then A must be a primary key.

Theorem 6.1 (Side-effect Free XML Update) *A correct update u to an XML view defined by a query tree qt is side-effect free if for all (U_i, V_i), where V_i is the corresponding relational view of qt_i and U_i is the translation of u over V_i, U_i is side-effect free in V_i.*

Proof: In our approach, a given XML update u can be mapped to a *set* of updates over the corresponding relational views. Formally, $u = \bigcup_{i=n}^{i=1} (U_i, V_i)$, $n \geq 1$. The update u is an atomic operation, that is, it has to be executed completely, or aborted.

Suppose one of the updates (U_k, V_k), $1 \leq k \leq n$ is not side-effect free. Since u is an atomic operation, it needs all of its n components to work correctly to be considered successful. Consequently, if update (U_k, V_k) fails, u also fails. Additionally, if update (U_k, V_k) is not side-effect free, then u is also not side-effect free. ■

Based on theorem 6.1, we can now answer a more general question: Is there a class of query tree views for which all possible updates are side-effect free? To answer this question, we summarize the results of (BRAGANHOLO; DAVIDSON; HEUSER, 2003a) and (DAYAL; BERNSTEIN, 1982a) (presented in Section 6.1) for conditions under which NRA views are updatable, and generalize them for XML views constructed by query trees.

Insertions. An insertion over an NRA view is side-effect free when the corresponding relational view V is a select-project-join view, the primary and foreign keys of the source relations of V are in the view and joins are made only through foreign keys. In terms of query trees, this means that the primary keys of the source relations of qt_i must appear as values in leaf nodes of qt_i and the *where* annotations in qt_i specifies joins using foreign keys, for all split trees qt_i corresponding to a query tree qt.

Deletions and modifications. Deletions and modifications over an NRA view V are side-effect free when the above conditions for insertions are met and V is *well-nested* (BRAGANHOLO; DAVIDSON; HEUSER, 2003a). We rephrase this condition in terms of query trees as follows:

Definition 6.7 (Well-Nested Query Tree) *A query tree qt is* well-nested *if for any two source relations R and S in qt, if S is related to R by a foreign key constraint then the source annotation for R occurs in an ancestor of the node s containing the source annotation for S. Additionally, attributes of R must not appear as values in the descendants of s.*

The results above identify three classes of updatable XML views: one that is updatable for all possible insertions; one that is updatable for all possible insertions, deletions and modifications; and a general one whose updatability with respect to a given update can be reasoned about using theorem 6.1. Furthermore, we can now prove the following:

Theorem 6.2 (No side-effects) *Given a query tree qt with the restrictions mentioned above and defined over a BCNF database $\mathcal{D}$, then for any instance d of $\mathcal{D}$ and correct update u over qt:* apply(eval(qt,d), u) $\equiv$ eval(qt, d'), *where d' is the updated relational database state resulting from the application of the translated view updates {U_{11}, ..., U_{1m_1}, ..., U_{n1}, ..., U_{nm_n}} using the techniques of (DAYAL; BERNSTEIN, 1982a).*

We leave the study of updatability using other existing relational techniques for future work. We have also studied the updatability of query trees extended to support group nodes. The results are shown in Appendix A.3.

6.3 Chapter Remarks

The updatability study shown in this chapter was published in two papers. The first one (BRAGANHOLO; DAVIDSON; HEUSER, 2003a) comprises the updatability of NRA views, and was published on the WebBD Workshop, held in conjunction with SIGMOD 2003. The second one (BRAGANHOLO; DAVIDSON; HEUSER, 2004a) presents the updatability of query tree views, and was published on the VLDB Conference of 2004. The main focus of this later paper was not the updatability study, so some of the details were omitted. They are available in (BRAGANHOLO; DAVIDSON; HEUSER, 2004b).

7 QUERY TREES APPLIED IN PRACTICE

To show the feasibility of our ideas, we have implemented our approach in a system that we call PATAXÓ. PATAXÓ is the name of a native Brazilian tribe (there are still a few living in Bahia) and stands for "**P**ermitindo **AT**ualizações **A**través de visões **X**ml em bancos de dados relaci**O**nais", which is loosely translated as permitting updates on relational databases through XML views.

As mentioned before, query trees are the formalism we chose to work with because it facilitates reasoning about update translation and updatability. However, they are not adequate for the end-user. A user needs a language in which he can specify XML views, that is, he needs a syntax. Additionally, the user should not be forced to learn another language just because he wants/needs to define XML views. In this scenario, there were two options:

1. To use SQL, which is the standard query language for relational databases (ISO, 2003a,b,c). SQL was extended to support XML, that is, it is able to produce XML as a result of a query. This extension is called SQLX (EISENBERG; MELTON, 2002) and it was standardized by ISO in 2003 (ISO, 2003d).

2. To use XQuery, which is the query language for XML. XQuery has been used as the view definition language in some of the most important work in the research area of XML extraction from relations – SilkRoute (FERNÁNDEZ et al., 2002) and XPERANTO (SHANMUGASUNDARAM et al., 2001).

We chose XQuery as the XML query language since it is widely accepted, and is becoming somewhat of a standard. We also borrowed some ideas from SQLX (EISENBERG; MELTON, 2002): we use the SQLX representation for relational tables (`row`), and define an input function to XQuery called `table` to access relational tables. This function, however, is slightly different from the one proposed in SQLX.

We have not chosen SQLX for several reasons: First, SQLX is not implemented yet in most of the RDBMSs. To the extent of our knowledge, the only RDBMS that supports SQLX is Oracle (ORACLE CORPORATION, 2002). Second, to the user well acquainted to XML, XQuery is more intuitive, since it allows him to specify the output format in an XML like way. Notice that the XQuery syntax is *not* XML, but it allows constructing XML elements by using XML tags. An XML syntax for XQuery is being developed, but is not a W3C recommendation yet (MALHOTRA et al., 2003).

In the next section we present our subset of XQuery, which we call UXQuery. The UXQuery language was proposed in (BRAGANHOLO; DAVIDSON; HEUSER, 2003b), together with the mapping to query trees and from there to relational views.

It is important to state that the term *auxiliary query tree* in (BRAGANHOLO; DAVIDSON; HEUSER, 2003b) corresponds to a previous version of our *query trees*. We show how a query in UXQuery is mapped to the current version of query trees in the next section. Section 7.2 shows the architecture of the PATAXÓ System and gives more details on the implementation.

7.1 UXQuery

XQuery's syntax is very broad and has lots of operators. Some of these operators - such as order related operators - do not really make sense when we are producing views of relational databases in which there is no inherent order. Furthermore, aggregate operators create ambiguity when mapping a given view tuple to the underlying relational database. We will therefore ignore ordering operators and outlaw aggregate operators. This means that the use of `let` in our subset of XQuery must be very carefully controlled, and for this reason we will allow it only as expanded by a new macro called `xnest`.

The subset we have chosen is called UXQuery (*Updatable XQuery*), and contains the following:

- FWOR `for/where/order by/return` expressions (note that we do not allow `let` expressions).
- Element and attribute constructors.
- Comparison expressions.
- An input function `table`, which binds a variable to tuples of a relational table that is specified as a parameter to the function.
- A macro operator called `xnest`, which facilitates the construction of heterogeneous nested sets.

In this chapter, we assume the reader is familiar with XQuery, its syntax and semantics. We will not get into details on the syntax and semantics that UXQuery "inherited" from XQuery. For further details on XQuery, please refer to (BOAG et al., 2004). The XQuery use cases are an easy way to understand XQuery (CHAMBERLIN et al., 2003).

As a first example, we show a very simple view definition query in UXQuery which retrieves vendors and their deposits. The query is shown in Figure 7.1. The only difference from this query to a query in XQuery is the `table` input function. Lines 2 and 6 show the invocation of such function. The name of the relational table is passed as an argument to the function. The XML view resulting from this query is shown in Figure 7.2.

This sample query and all queries in UXQuery must respect the EBNF of UXQuery, which is shown in Figure 7.3. This EBNF was based on the EBNF of the XQuery Core (BOAG et al., 2004), and it was simplified to remove operators not allowed by UXQuery. We have also added the `xnest` operator, and the `table` input function. In the EBNF we use a set of grammar definitions available in the XML documentation. The basic tokens `Letter` and `Digit` are defined in (BRAY et al.,

```
 1. <vendors>
 2.  {for $v in table(’Vendor’)
 3.   return
 4.   <vendor id=’{$v/vendorid/text()}’>
 5.      {$v/vendorname}
 6.      {for $d in table(’Deposit’)
 7.       where $v/vendorid=$d/vendorid
 8.       return
 9.       <deposit>
10.           <idDeposit>{$d/depid/text()}</idDeposit>
11.           <address>
12.              <street>{$d/address/text()}</street>
13.              {$d/city}
14.              {$d/state}
15.              {$d/country}
16.           </address>
17.        </deposit>
18.       }
19.    </vendor>
20.   }
21. </vendors>
```

Figure 7.1: Example of a simple query that retrieves vendors and deposits

2004). The identifier `QName` is defined in (BRAY; HOLLANDER; LAYMAN, 1999). Literals and numbers are defined in (BOAG et al., 2004).

The formal semantics of UXQuery matches the semantics of XQuery (DRAPER et al., 2004) with the exception of the new input function `table` and the macro `xnest`, which we discuss next.

Semantics of `table()`. XQuery has two input functions: `collection` and `doc` (MALHOTRA; MELTON; WALSH, 2004). In UXQuery, the only input function available to the user is `table`. This function takes as input a table from a relational database and returns a set of tuples in the following form:

```
<row>
   <!-- tuple attributes -->
   ...
</row>
...
```

Following SQLX (EISENBERG; MELTON, 2002), we translate this input function to XQuery as follows.

```
define function table($tableName as xs:string) as node*
{
   let $tuples := doc(concat($tableName,".xml"))//row
   return $tuples
}
```

For this input function to work, the relational table used as the parameter in the function call must be represented in XML. As an example, the function call shown in line 2 of Figure 7.1 assumes that table *Vendor* is available in a file named `vendor.xml` which has the following structure:

```
<vendors>
  <vendor id="01">
    <vendorname>Amazon</vendorname>
    <deposit>
      <idDeposit>D1</idDeposit>
      <address>
        <street>1245, Bourbom Street</street>
        <city>Seatle</city>
        <state>WA</state>
        <country>USA</country>
      </address>
    </deposit>
    <deposit>
      <idDeposit>D3</idDeposit>
      <address>
        <street>4545, 15th Avenue</street>
        <city>Seatle</city>
        <state>WA</state>
        <country>USA</country>
      </address>
    </deposit>
  </vendor>
  <vendor id="02">
    <vendorname>Barnes and Noble</vendorname>
    <deposit>
      <idDeposit>D2</idDeposit>
      <address>
        <street>1478, 25th Avenue</street>
        <city>New York</city>
        <state>NY</state>
        <country>USA</country>
      </address>
    </deposit>
  </vendor>
</vendors>
```

Figure 7.2: XML view resulting from the query of Figure 7.1

```
<vendor>
  <row>
    <vendorid>01</vendorid>
    <vendorname>Amazon</vendorname>
    <url>www.amazon.com</url>
    <state>WA</state>
    <country>USA</country>
  </row>
  <row>
    ...
  </row>
  ...
</vendor>
```

Semantics of `xnest`. The `xnest` operator is used to specify possibly heterogeneous sets of nested tuples that agree in the value of one or more attributes. The tuples are clustered according to the value of these attributes, which we call *nesting attributes*. A simple (non-heterogeneous) example of such a query is shown in Figure 7.4 (lines 1-23). The query specifies a join of tables *Vendor*, *Book* and *SellBook*. For each vendor, it shows the vendor name, the vendor Id, and the books sold by that vendor clustered by price. The nesting attribute in this case is *price*.

```
[1]  UXQuery         ::= QueryBody
[2]  QueryBody       ::= ElmtConstructor
[3]  ElmtConstructor ::= "<" QName AttList "/>" | "<" QName AttList? ">" ElmtContent+ "</" QName ">"
[4]  ElmtContent     ::= ElmtConstructor | EnclosedExpr+
[5]  AttList         ::= ((QName "=" AttValue)?)+
[6]  AttValue        ::= ('"' AttValueContent '"') | ("'" AttValueContent "'")
[7]  AttValueContent ::= "{" PathExprAtt "}"
[8]  PathExprAtt     ::= "$" VarName "/" QName "/" NodeTest
[9]  VarName         ::= QName
[10] EnclosedExpr    ::= "{" (FWRExpr | PathExpr | Nest) "}"
[11] Expr            ::= OrExpr
[12] OrExpr          ::= AndExpr ( "or"  AndExpr )*
[13] AndExpr         ::= ComparisonExpr ("and" ComparisonExpr)*
[14] FWRExpr         ::= ((ForClause)+ WhereClause? OrderByClause? "return")* ElmtConstructor
[15] ComparisonExpr  ::= ValueExpr (GeneralComp ValueExpr )?
[16] ValueExpr       ::= PathExpr | PrimaryExpr
[17] PathExpr        ::= "$" VarName "/" QName ("/" NodeTest)?
[18] NodeTest        ::= TextTest
[19] TextTest        ::= "text" "(" ")"
[20] ForClause       ::= "for" "$" VarName "in" TableExpr ("," "$" VarName "in" TableExpr)*
[21] TableExpr       ::= "table (" '"' QName '"' ")" | "table (" "'" QName "'" ")"
[22] WhereClause     ::= "where" Expr
[23] GeneralComp     ::= "=" |  "!=" |  "<" |  "<=" |  ">" |  ">="
[24] OrderByClause   ::= "order" "by" OrderSpecList
[25] OrderSpecList   ::= OrderSpec ("," OrderSpec)*
[26] OrderSpec       ::= PathExpr
[27] PrimaryExpr     ::= Literal | ParenthesizedExpr
[28] Literal         ::= NumericLiteral |  StringLiteral
[29] NumericLiteral  ::= IntegerLiteral |  DecimalLiteral |  DoubleLiteral
[30] ParenthesizedExpr  ::= "(" Expr? ")"
[31] Nest       ::= NestClause ByClause WhereClause "return" Header
[32] NestClause ::= "xnest "$" VarName "in" TableExpr ("," "$" VarName "in" TableExpr)*
[33] ByClause   ::= "by" "$" VarName "in" UnionExpr ("," "$" VarName "in" UnionExpr)*
[34] Header ::= "<" QName (QName "=" NestAttValue)+ ">" ( "{" ElGroup "}" )+ "</" QName ">"
       | "<" QName ">" (( "{" "$" VarName "}") | ("<" QName ">" "{" "$" VarName "/" TextTest "}"
         "</" QName ">"))+ ( "{" ElGroup "}" )+ "</" QName ">"
[35] NestAttValue ::= "'" "{" "$" VarName "/" TextTest "}" "'"
                  |  '"' "{" "$" VarName "/" TextTest "}" '"'
[36] ElGroup    ::= ElmtConstructor
[37] UnionExpr  ::= "(" "$" VarName "/" QName ( ("union" | "|")  "$" VarName "/" QName)* ")"
```

Figure 7.3: EBNF of UXQuery

We believe `xnest` is an interesting addition to our subset of XQuery. In fact, there have been lots of discussions at W3C to add an operator similar to `xnest` to XQuery. Additionally, other researchers have also identified the need of such operator. An example is the `group by` operator, proposed in (DEUTSCH; PAPAKONSTANTINOU; XU, 2004a,b).

7.1.1 Normalization to XQuery

A query containing an `xnest` operator can be normalized to one using pure XQuery syntax. The normalized query corresponding to the query in Figure 7.4 (lines 1-23) is shown in Figure 7.4 (lines 24-49). The normalization process makes sure that the nest variable (in the example, *$price*) appears in the *Header* element as an attribute or a sub-element. In the example, the *Header* element is *books*. Notice that in the normalized query, we still use the input function `table`.

Continuing with the example, the `xnest` operation (lines 6-20) is normalized to the expression shown in lines 29-46. The expression consists of a `let/for` (lines 29-31) and an additional `for` (lines 34-44) for each *ElGroup* (lines 13-18) specified

```
 1. <vendors>
 2.   {for $v in table("Vendor")
 3.    return
 4.    <vendor id="{$v/vendorid/text()}">
 5.       {$v/vendorname}
 6.       {xnest $b in table("Book"),
 7.              $sb in table("SellBook")
 8.        by $price in ($sb/price)
 9.        where $v/vendorid=$sb/vendorid
10.              and $sb/isbn=$b/isbn
11.        return
12.        <books price="{$price/text()}">
13.           {
14.           <book>
15.              {$b/isbn}
16.              {$b/title}
17.           </book>
18.           }
19.        </books>
20.       }
21.    </vendor>
22.   }
23. </vendors>
```

```
24. <vendors>
25.   {for $v in table("Vendor")
26.    return
27.    <vendor id="{$v/vendorid/text()}">
28.       {$v/vendorname}
29.       {let $b’ := table("Book"),
30.            $sb’ := table("SellBook")
31.        for $price in distinct-values($sb’/price)
32.        return
33.        <books price="{$price/text()}">
34.          {for $b in table("Book"),
35.                $sb in table("SellBook")
36.           where $v/vendorid=$sb/vendorid
37.                 and $sb/isbn=$b/isbn
38.                 and $sb/price=$price
39.           return
40.           <book>
41.              {$b/isbn}
42.              {$b/title}
43.           </book>
44.          }
45.        </books>
46.       }
47.    </vendor>
48.   }
49. </vendors>
```

Figure 7.4: Example of a query that uses the xnest operator (lines 1-23) and its translation to regular XQuery syntax (lines 24-49)

in the query. In the normalization process, we introduce new variables in the `let` clause. These variables are primed (’), and correspond to the variables specified in the `xnest` operator. There will be one primed variable in the `let` clause for each variable specified in the `xnest` operator (XQuery does not accept variable names with (’). However, we use them here for ease of explanation).

The normalization process also makes sure that nested elements are related to the nesting variable. This is done by adding a new condition in the `where` clause. In the example (line 38) we added a condition requiring that the book is sold by the price specified by *$price*.

Note that this example shows a nesting over a single attribute, but that it is possible to specify nests over more than one attribute.

The query of Figure 7.4 has a single element group (*ElGroup*) (lines 13-18). In this example, it is not necessary to divide the conditions and variable bindings that appear in the `xnest` operator through the corresponding `for`s in the normalized query. We now show an example where this is necessary.

Figure 7.5 shows a query that has two element groups (lines 17-22 and 23-28). In this case, the normalized query will have two `for`s, one for each of the element groups (lines 46-56 and 57-67). The variable bindings and where conditions must then be carefully analyzed in order to identify to each of the `for`s they belong to. This is done by functions *fs:SubVariable(i)* and *fs:SubExpr(i)* of the normalization process shown below.

The normalization process described through the above examples can be formally stated as:

```
 1. <vendors>
 2.   {for $v in table("Vendor")
 3.    return
 4.    <vendor id="{$v/vendorid/text()}">
 5.       {$v/vendorname}
 6.       {xnest $b in table("Book"),
 7.              $sb in table("SellBook"),
 8.              $d in table("DVD"),
 9.              $sd in table("SellDVD")
10.        by $price in ($sb/price | $sd/price)
11.        where $v/vendorid=$sb/vendorid
12.              and $v/vendorid=$sd/vendorid
13.              and $sb/isbn=$b/isbn
14.              and $sd/asin=$d/asin
15.        return
16.        <products price="{$price/text()}">
17.           {
18.           <book>
19.              {$b/isbn}
20.              {$b/btitle}
21.           </book>
22.           }
23.           {
24.           <dvd>
25.              {$d/asin}
26.              {$d/dtitle}
27.           </dvd>
28.           }
29.        </products>
30.       }
31.    </vendor>
32.   }
33. </vendors>
```

```
34. <vendors>
35.   {for $v in table("Vendor")
36.    return
37.    <vendor id="{$v/vendorid/text()}">
38.       {$v/vendorname}
39.       {let $b' := table("Book"),
40.            $sb' := table("SellBook"),
41.            $d' in table("DVD"),
42.            $sd' in table("SellDVD")
43.        for $price in
           distinct-values($sb'/price | $sd'/price)
44.        return
45.        <products price="{$price/text()}">
46.          {for $b in table("Book"),
47.               $sb in table("SellBook")
48.           where $v/vendorid=$sb/vendorid
49.                  and $sb/isbn=$b/isbn
50.                  and $sb/price=$price
51.           return
52.           <book>
53.              {$b/isbn}
54.              {$b/btitle}
55.           </book>
56.          }
57.          {for $d in table("DVD"),
58.               $sd in table("SellDVD")
59.           where $v/vendorid=$sd/vendorid
60.                  and $sd/asin=$d/asin
61.                  and $sd/price=$price
62.           return
63.           <dvd>
64.              {$d/asin}
65.              {$d/dtitle}
66.           </dvd>
67.          }
68.        </products>
69.       }
70.    </vendor>
71.   }
72. </vendors>
```

Figure 7.5: Example of a query with two element groups (lines 1-33) and its translation to regular XQuery syntax (lines 34-72)

```
[xnest Variable$_1$ in TableExpr$_1$, ..., Variable$_n$ in TableExpr$_n$
by NestVariable$_1$ in (Variable$_{1_1}$/QName$_{1_1}$ | ... | Variable$_{1_m}$/QName$_{1_m}$),
..., NestVariable$_k$ in (Variable$_{k_1}$/QName$_{k_1}$ | ... | Variable$_{k_m}$/QName$_{k_m}$)
where Expr return
<ElName AttName$_1$="{NestVariable$_1$/text()}"... AttName$_k$="{NestVariable$_k$/text()}">
{ElGroup$_1$} ...{ElGroup$_m$} </ElName> ]$_{xnest}$
==
let Variable$'_1$ := TableExpr$_1$, ..., Variable$'_n$ := TableExpr$_n$
for NestVariable$_1$ in distinct-values(Variable$_{1_1}$/QName$_{1_1}$ | ... | Variable$_{1_m}$/QName$_{1_m}$),
..., NestVariable$_k$ in distinct-values(Variable$_{k_1}$/QName$_{k_1}$ | ... | Variable$_{k_m}$/QName$_{k_m}$)
return
<ElName AttName$_1$="{NestVariable$_1$/text()}", ..., AttName$_k$="{NestVariable$_k$/text()}">
{for fs:SubVariable(1)
where fs:SubExpr(1) and (Variable$_{1_1}$ = NestVariable$_1$ and ...and Variable$_{k_1}$ = NestVariable$_k$)
return ElGroup$_1$ }
...
{for fs:SubVariable(m)
where fs:SubExpr(m) and (Variable$_{1_m}$ = NestVariable$_1$ and ...and Variable$_{k_m}$ = NestVariable$_k$)
return ElGroup$_m$ }
</ElName>
```

The notation for the normalization process is the same as that in (DRAPER et al., 2004). The process assumes that:

- {Variable$_{1_1}$, ..., Variable$_{1_m}$, ..., Variable$_{k_1}$, ..., Variable$_{k_m}$} $\subseteq$ {Variable$_1$, ..., Variable$_n$}

- The auxiliary function *fs:SubVariable(i)* returns all variables V_x referenced in *ElGroup$_i$* and also all variables V_y appearing in a condition of the form "$V_x/QName_x$ **cmp** $V_y/QName_y$" or "$V_y/QName_y$ **cmp** $V_x/QName_x$" in *Expr* in the `where` clause of the `xnest` operator; **cmp** $\in \{=, <, >, !=, <=, >=\}$.

- The auxiliary function *fs:SubExpr(i)* returns every expression specified in *Expr* in the `where` clause of the `xnest` operator that references a variable returned by *fs:SubVariable(i)*.

Returning to the example of Figure 7.5, the first element group (lines 17-22) references variable `$b`. Additionally, there is a where condition that uses `$b` and references `$sb` (`$sb/isbn=$b/isbn`, line 13). That's why the function *fs:SubVariable(1)* returns `$b` and `$sb`. These variables are used in the `for` clause corresponding to this element group (lines 46-47). The where conditions for this element group are found by function *fs:SubExpr(1)*, which analyzes the where condition of the `xnest` expression and takes all such conditions that references variables `$b` and `$sb`. A condition requiring that each book is sold by the price specified by `$price` is also added. The resulting where condition is shown in lines 48-50. The same process is done with the second element group.

In Section 7.2, we show how the normalized query is used to produce the XML view.

```
<vendors>
{for $v in table('Vendor')
 return
 <vendor id='{$v/vendorid/text()}'>
    {$v/vendorname}
    {for $d in table('Deposit')
     where $v/vendorid=$d/vendorid
     return
     <deposit>
          <idDeposit>{$d/depid/text()}</idDeposit>
          <address>
             <street>{$d/address/text()}</street>
             {$d/city}
             {$d/state}
             {$d/country}
          </address>
       </deposit>
      }
   </vendor>
  }
</vendors>
```

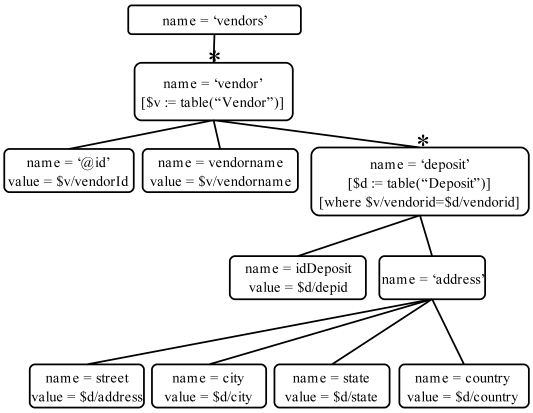


Figure 7.6: Example of UXQuery that joins two relations and its query tree

7.1.2 From UXQuery to Query Trees

As mentioned before, query trees are used as an intermediate representation of the view definition query. To be able to use query trees internally, it is necessary to define how a view definition query expressed in UXQuery is translated to its corresponding query tree. In the translation, we use query trees extended with *group nodes*, since the `xnest` operator groups nodes together according to one or more values. Appendix A shows details on this extension. To illustrate the mapping to extended query trees, we start with the query of Figure 7.1. For readability reasons, the query is presented again in Figure 7.6, together with its query tree.

Each XML element specified in the query is represented by a node in the query tree. Each node in the query tree needs a name, and possibly a value (if it is a leaf node). Since UXQuery allows constructing XML elements and attributes in three distinct ways, we analyze each case separately:

- The leaf element is generated by an expression `{$x/A}`: in this case, the corresponding node in the query tree has name A and value $\$x/A$.

- The leaf element is constructed by an expression `<tagName> {$x/A/text()} </tagName>`: in this case, the corresponding node in the query tree has name *tagName* and value $\$x/A$.

- The leaf element is an attribute constructed by an expression `attName = "{$x/A/text()}"`: in this case, the node in the query tree has name *@attName* and value $\$x/A$.

As an example, the expression `$v/vendorname` in the query of Figure 7.6 is mapped to a node named *vendorname* in the query tree. As an example of mapping of an attribute, see node *@id*.

An exception to the above rules is an element or attribute which uses a *nesting variable* to specify its content. For example, attribute `price` in the query of Figure

7.5 is constructed using variable `$price` as its content (line 16). The variable `$price` was specified as `$price in ($sb/price | $sd/price)` (line 10). In this case, the rules for the node name are the same as above, (in this example, the node will be named *price*), but its value is *GROUP($sb/price | $sd/price)*. *GROUP* is defined in extended query tress (Appendix A). The formal rule for this case can be specified as:

- The leaf element `tagName` is specified by a nesting variable `$y`, and is constructed as `<tagName>{$y}</tagName>`. Variable `$y` is in turn specified as `$y in ($x`$_1$`/A`$_1$` | ... | $x`$_n$`/A`$_n$`)`. The corresponding node will be named *tagName* and its value will be $GROUP(\$x_1/A_1 \mid ... \mid \$x_n/A_n)$.
- The attribute `attName` is specified by a nesting variable `$y`, and is constructed as `attName ="{$y/text()}"`. Variable `$y` is in turn specified as `$y in ($x`$_1$`/A`$_1$` | ... | $x`$_n$`/A`$_n$`)`. The corresponding node will be named *@attName* and its value will be $GROUP(\$x_1/A_1 \mid ... \mid \$x_n/A_n)$.

Non-leaf elements can only be constructed with an expression of type `<tagName> {content} </tagName>`, where `content` are other element constructors, `for`s and/or `xnest`s. In this case, the corresponding node in the query tree will have name *tagName*, but no value. As an example, the XML element `address` in the query of Figure 7.6 is a non-leaf element whose content are four element constructors. Its corresponding node in the query tree is named *address*, and has no value.

Nodes in the query tree are connected to represent the parent/child relationship of XML elements in the view definition query. As an example, the node *address* is connected to nodes *street*, *city*, *state* and *country* in the query tree. In the view definition query, elements `street`, `city`, `state` and `country` are children of `address`. We will explain how starred edges are identified later.

Source and where annotations are identified as follows. Each `for` expression in the view definition query has variable bindings, optional where conditions and a return clause followed by an element constructor. Suppose this element is named `e`. The variable bindings are placed as source annotations in the node e that represents element `e` in the query tree. A variable binding of type `$x in table("X")` becomes a source annotation of type [*$x := table("X")*]. The where conditions (if any) are placed in node e as where annotations (`where x` becomes [*where x*]). After this, change the edge that connects e to its parent to a *-edge. As an example, the query of Figure 7.6 has a `for` expression at line 2. The expression has an element constructor after the `return` clause that constructs the element `vendor` (line 4). As a consequence, the node *vendor* in the query tree is a starred node, and it has a source annotation [*$v := table("Vendor")*].

When a query has an `xnest` operation, the source and where annotations are identified using the functions *fs:SubVariable(i)* and *fs:SubExpr(i)*, shown in Section 7.1.1. The *Header* element is mapped to a node that has a *-edge, but no source annotation. In the query of Figure 7.5, the *Header* element is `products`, and the corresponding node is shown in the query tree of Figure 7.7. After this query tree is typed, this node will receive a type τ_G.

The root element of each element group in the query receives a *-edge. The source annotations are selected using the function *fs:SubVariable(i)* to identify the relevant variables for that node. In the same way, the function *fs:SubExpr(i)* is used to identify the where annotations for the node. As an example, node *book* in Figure 7.7 has

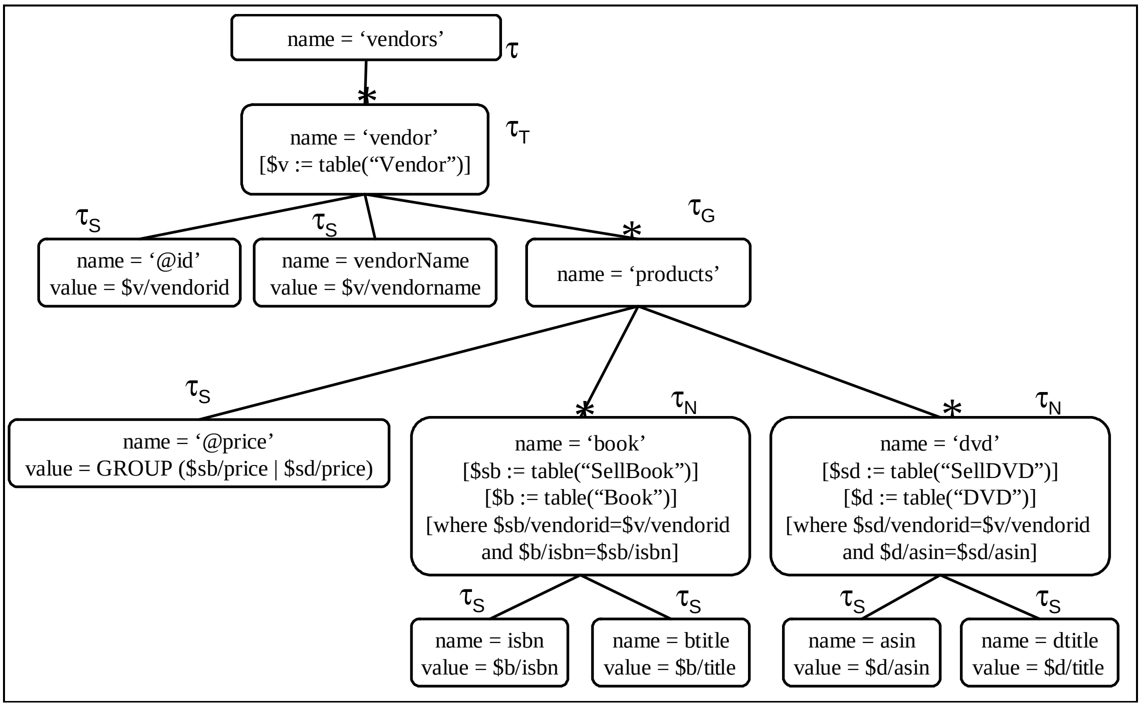


Figure 7.7: Query tree corresponding to the query of Figure 7.5

source annotations [*$b := table("Book")*] and [*$sb := table("SellBook")*]. Similarly, its where annotation is [*where $v/vendorid=$sb/vendorid AND $b/isbn=$sb/isbn*].

After obtaining the query tree corresponding to a given view definition query in UXQuery, it is possible to use the strategy specified in chapters 5 and 6 to map updates over the resulting XML view to the underlying relational database. In the next section, we show the system which implements this mechanism.

7.2 PATAXÓ: The Prototype

The PATAXÓ System implements the UXQuery language, and allows users to issue updates over an XML view constructed by an UXQuery query. PATAXÓ was implemented in Java (SUN MICROSYSTEMS, 1994) using the following additional packages:

- To create, parse and manipulate XML we have used `xerces.jar` (APACHE SOFTWARE FOUNDATION, 2000);
- To create, parse and manipulate DTDs we have used `de.jar` [1];
- To parse and execute XQuery queries we have used `saxon7.jar` (KAY, 2001);
- To create a parser for UXQuery we have used the JavaCC parser generator (SUN MICROSYSTEMS, 2001). JavaCC does not require any import of packages in the source code of PATAXÓ. The only necessary package is the UXQuery parser generated by JavaCC, which we called `parser`. JavaCC was also used to build a parser for the XPath expressions used to provide update paths. We called such parser `xpathparser`.

[1] `http://www.rpbourret.com/xmldbms/docs/Package-de.tudarmstadt.ito.schemas.dtd.html`

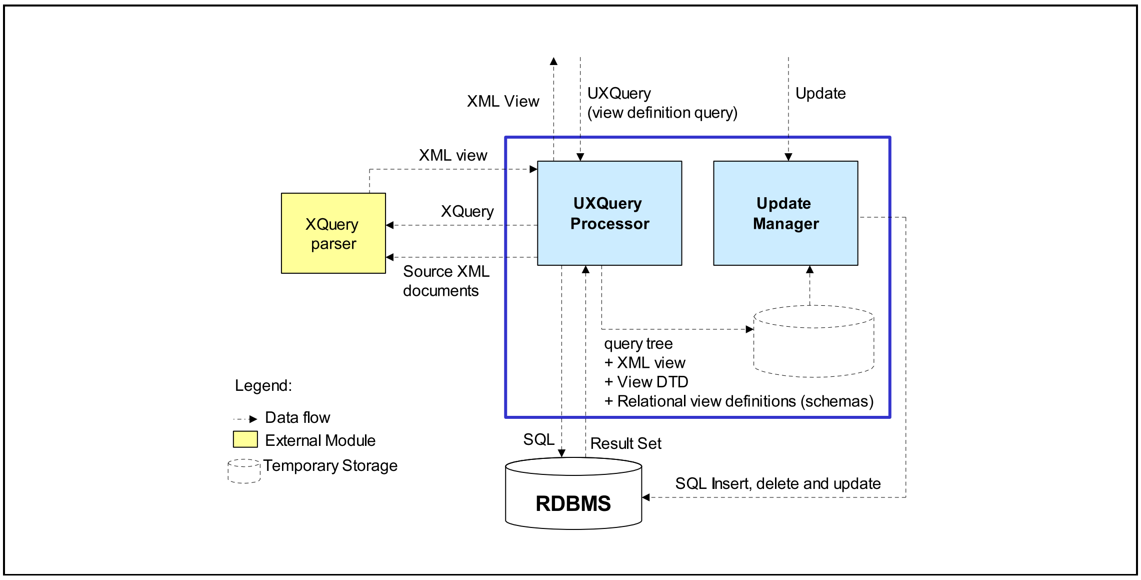


Figure 7.8: PATAXÓ System architecture

The overall architecture of PATAXÓ is shown in Figure 7.8. The system is composed of two main modules: the *UXQuery Processor* and the *Update Manager*.

The *UXQuery Processor* is responsible for processing the view query definition and generating the XML view. The *Update Manager* receives update requests from users and maps them to updates in the corresponding relational views. A sub-module called *Relational View Updater* checks whether or not the updates are translatable to the underlying relational database using the algorithm of (DAYAL; BERNSTEIN, 1982a). If so, this module produces the SQL insert/delete/update statements in order to reflect the changes to the underlying relational database.

7.2.1 UXQuery Processor

The UXQuery processor (Figure 7.9) is the module responsible for processing a view definition query expressed in UXQuery and producing the corresponding XML view. In order to do this, it translates a query in UXQuery to a query using pure XQuery syntax. The relational source data is then translated to XML by a submodule called *XML Extractor*. XML Extractor takes a relational table and encodes it in XML using an element `row` as a tuple delimiter. For example, the *Vendor* table of Figure 1.2 is represented in XML as:

```
<vendor>
  <row>
    <vendorid>01</vendorid>
    <vendorname>Amazon</vendorname>
    <url>www.amazon.com</url>
    <state>WA</state>
    <country>USA</country>
  </row>
  <row>
    ...
  </row>
  ...
</vendor>
```

Notice that the column names are represented in lower case, even though they are in mixed case in the database of Figure 1.2. This is because we had to adopt a standard for names in the implementation. Each relational database engine provides the column names in an arbitrary format. Some of them use uppercase, others provide the column names exactly like written by the user in the `CREATE TABLE` command, still others use lower case. Since XML is case sensitive, we chose to use lower case in the extraction of tables to XML, to avoid problems to the user. If we had left the RDBMS to decide the case of the column names, the user would need to "guess" how to reference those names in the view definition query. As a consequence of our decision, the queries must always reference column names in lowercase. See the queries in this chapter for examples.

In order to know which are the column names of each table of the underlying relational database, the XML Extractor uses the metadata provided by the RDBMS. The package `java.sql` provides methods to access such data.

The XML Extractor does not extract the entire table. It uses the selection conditions specified in the UXQuery (`where` conditions) to eliminate unnecessary tuples, and projects only the columns specified in the query. In this way, we avoid extracting data that would be discarded by the XQuery processor.

The XQuery query is generated using the parse tree produced by the *UXQuery Parser*. The parser was built using the JavaCC (SUN MICROSYSTEMS, 2001) parser generator, and catches syntactic errors in the UXQuery query. The *XQuery generator* uses the normalization rules presented in Section 7.1.1 to produce the XQuery query.

The parse tree is also used to generate the *query tree*. In our implementation, we use extended query trees (see Appendix A), since UXQuery supports clustering XML elements according to a value or a set of values (`xnest`). This extended query tree is used by the *Relational View Mapper* to generate the relational views that correspond to the XML view as explained in Section 5.1. The query tree is also used by the *DTD Generator* to generate the schema of the XML view (Section 4.1.4).

After extracting the XML files that represent the underlying relational tables (*XML Extractor*) and producing the XQuery query (*XQuery Generator*), an external XQuery processor (Saxon (KAY, 2001)) is used to process the query. The result of this processing is the XML view which is returned to the user.

7.2.2 Update Manager

The Update Manager (Figure 7.10) is the module responsible for receiving update requests and mapping the updates to the underlying relational database. In order to do so, it first checks whether or not the update conforms to the view schema and rejects updates that do not conform.

Using the query tree, the *Relational View Update Generator* takes the requested update and translates it to the corresponding relational views (as specified in Section 5.2). The *Relational View Updater* then uses the techniques of Dayal and Bernstein (DAYAL; BERNSTEIN, 1982a), which are shown in chapter 2, to translate the updates to the underlying database. Updates with side-effects are rejected. The *Relational View Updater* was implemented by Angelo Agra in his Bachelor Thesis (AGRA, 2004).

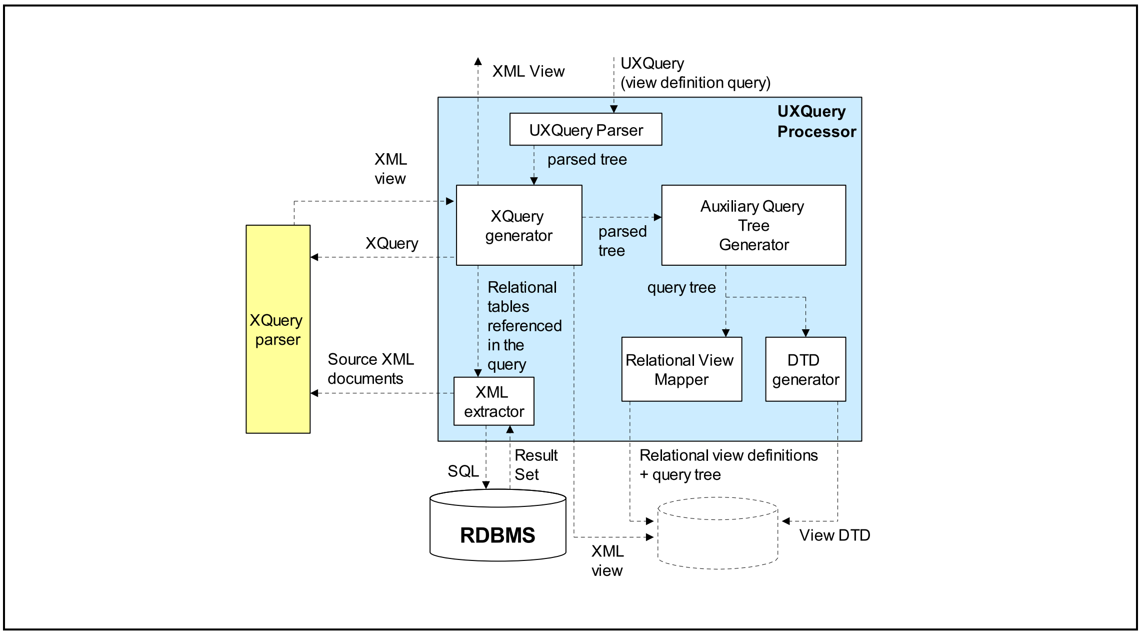


Figure 7.9: UXQuery Processor

7.2.3 Graphical User Interface

We have developed a graphical interface through which the user can submit view definition queries in UXQuery and updates over the resulting view.

When a user starts the PATAXÓ system, he must first establish a database connection using JDBC. Figure 7.11 shows the system taskbar, where the Database Connection is the first available button. The main screen of the system is shown in Figure 7.12.

After establishing the connection, the user can either write a view definition query, or load one from an existing text file. A click on "Execute query" sends the view query to the UXQuery Processor, and the resulting XML view is shown to the user as a tree in the graphical interface (Figure 7.13 (left hand side)). The interface also shows the view DTD (Figure 7.14) and the schema of the corresponding relational views (Figure 7.15).

The user can now issue updates against the XML view. Updates can be specified in two ways through the graphical interface:

- by editing the XML tree directly;
- or by using the alternative update interface (right hand side of Figure 7.13).

The updates supported by editing the XML tree directly include changing the value of a leaf node, deleting subtrees, and inserting subtrees. When finished, the user informs the system that he wants the updates to be propagated to the database by clicking the "Update" button. This sends the update request to the *Update Manager*, which tries to propagate the updates to the underlying relational database. In case of a problematic update, the user is informed and the operation is rolled back.

To use the alternative interface (shown on the right hand side of Figure 7.13), the user writes an update path expression to indicate the set of nodes affected by the update. In the case of insertion and modification, the user must also specify

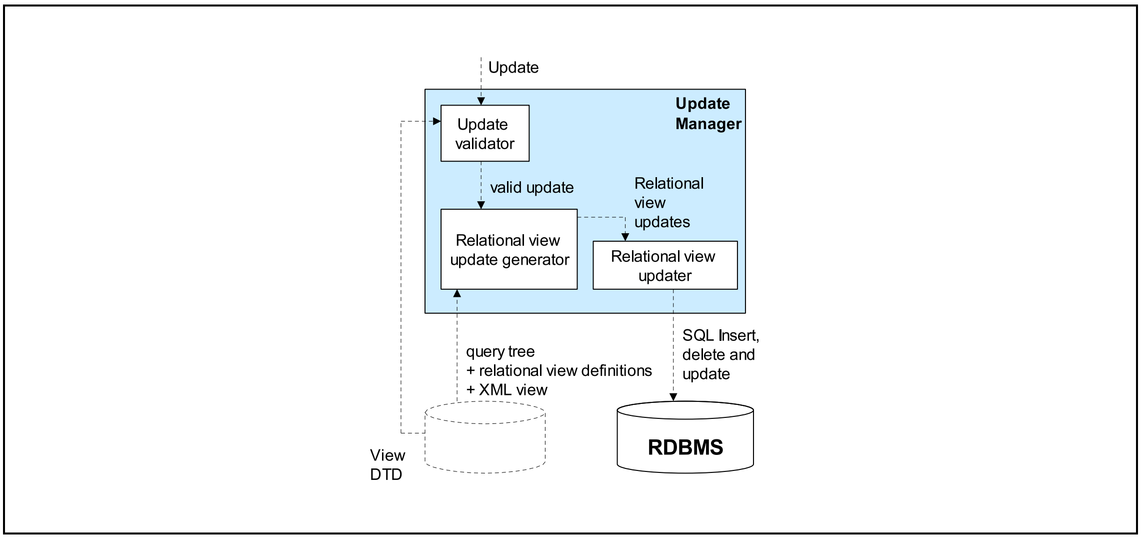


Figure 7.10: Update Manager

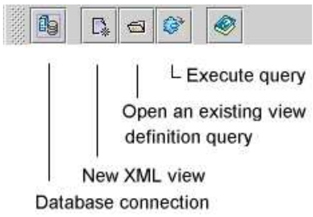


Figure 7.11: System taskbar

additional parameters to the update in a text area. This additional information consists of:

- the new node value, if the update type is a modification;
- the subtree to be inserted under the nodes selected by the update XPath expression, if the update type is an insertion;

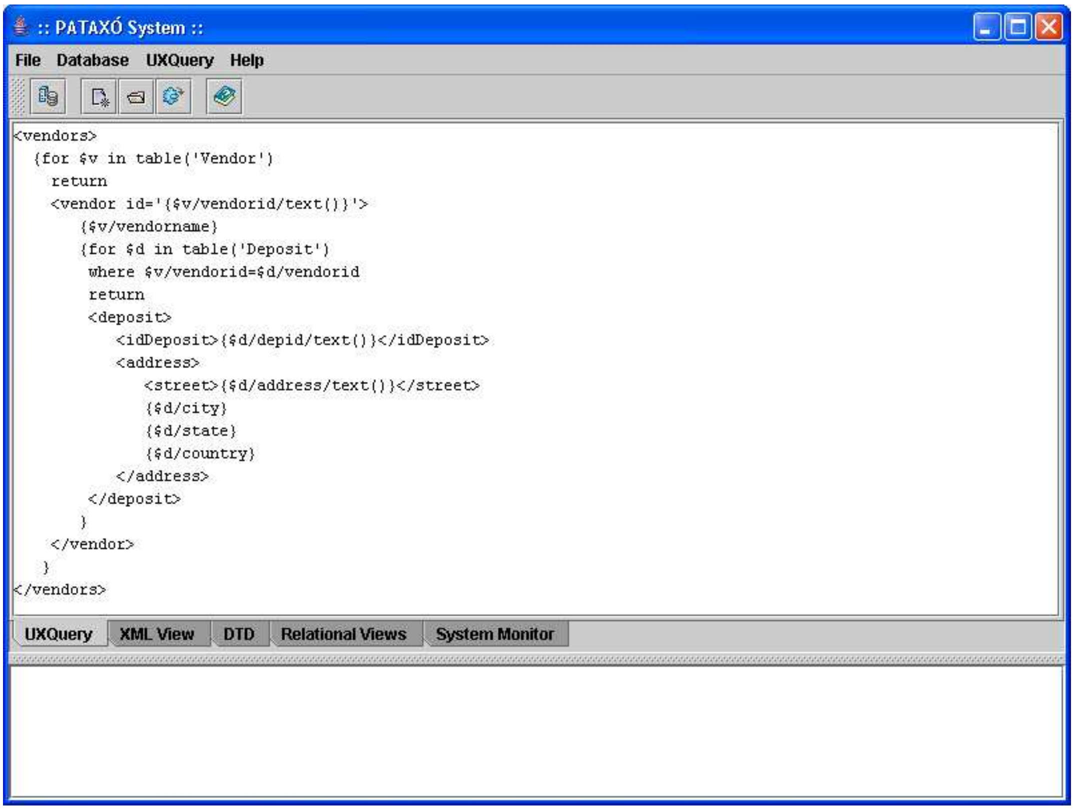
Deletions do not require any additional information, since all nodes under the nodes selected by the update XPath expression will be deleted.

The graphical interface simplifies the specification of updates as much as possible. For example, an update path expression can be generated automatically by clicking on a node in the XML tree and then clicking the "»" button.

7.2.4 Main Difficulties

When implementing PATAXÓ, we have faced some problems related to how SQL is implemented is each RDBMS. Since PATAXÓ is Open Source, we would like to test it with Open Source RDBMS's like PostgresSQL (POSTGRESQL, 1995) and MySQL (MYSQL AB, 1995). The problem is that we need nested queries in the update translations, such as:

```
UPDATE Book
SET title="New Title"
```



```
<vendors>
  {for $v in table('Vendor')
    return
    <vendor id='{$v/vendorid/text()}'>
       {$v/vendorname}
       {for $d in table('Deposit')
        where $v/vendorid=$d/vendorid
        return
        <deposit>
           <idDeposit>{$d/depid/text()}</idDeposit>
           <address>
              <street>{$d/address/text()}</street>
              {$d/city}
              {$d/state}
              {$d/country}
           </address>
        </deposit>
       }
    </vendor>
  }
</vendors>
```

Figure 7.12: User Interface

```
WHERE Book.isbn IN
    (SELECT Book.isbn
     FROM Vendor v, Book b, SellBook sb
     WHERE v.vendorid=b.vendorid AND sb.isbn=b.isbn
     AND isbn="1111")
```

and the current version of MySQL (4.0) (MYSQL AB, 1995) does not support such queries. As for PostgreSQL, there is no problem regarding nested queries.

Another problem was faced when using the JDBC drivers provided by the developers of each RDBMS we used. Most of them do not implement access to metadata, such as get the attribute names of a given table, the primary keys of a given table, and so on. For this reason, we had to use specific drivers for each RDBMS we used. This is not really a problem, but it creates an additional difficult for the user. He has to get the correct driver on the internet in order to connect to the database. Notice that we can not provide those drivers together with the distribution code, since most of them are copyrighted.

7.3 Chapter Remarks

We have shown how the ideas presented in Chapters 4 and 5 were implemented in a prototype called PATAXÓ. The prototype shows the feasibility of our ideas, and was implemented in Java (SUN MICROSYSTEMS, 1994). Java was chosen because it is platform independent.

The main contributions of this chapter are:

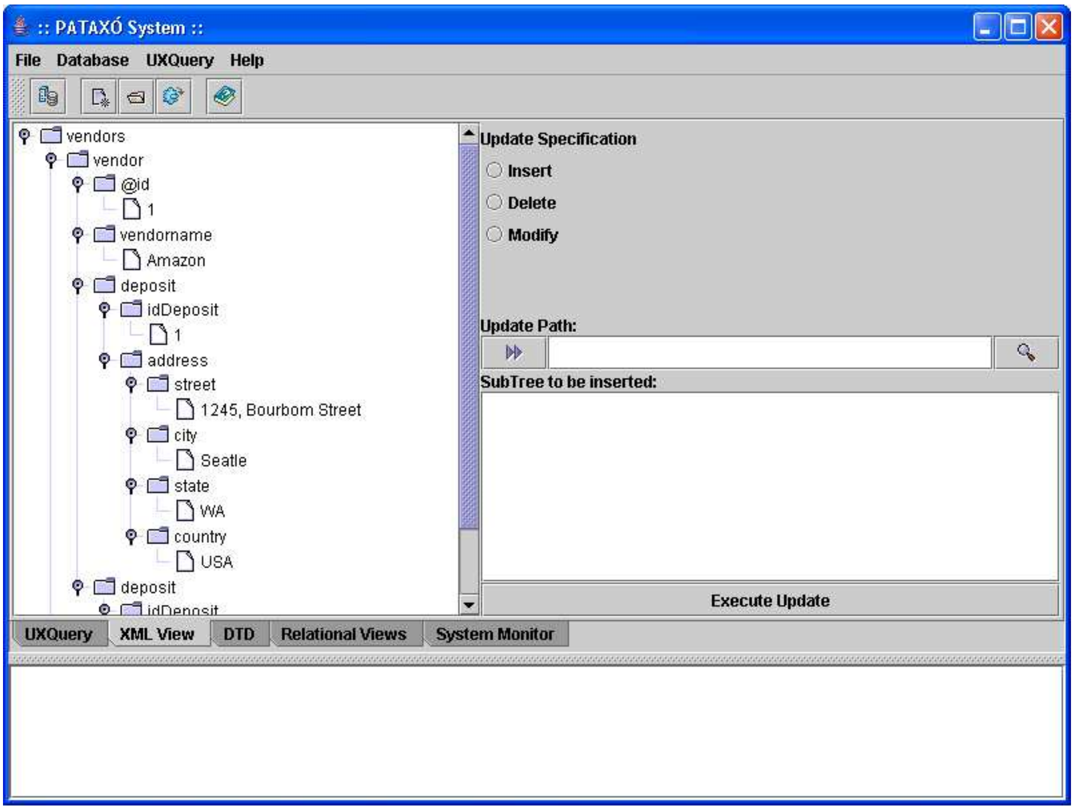


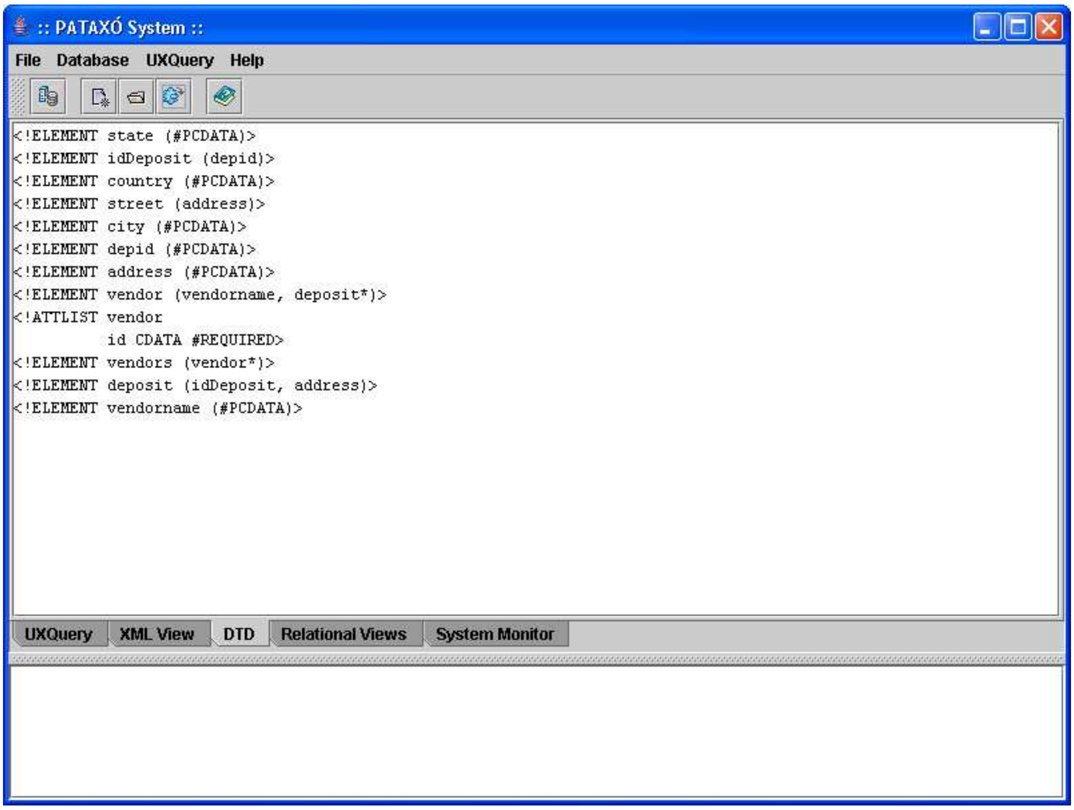
Figure 7.13: Alternative interface to update the XML view

- The definition of UXQuery – a subset of XQuery augmented with an input function `table` and a macro called `xnest`.

- The definition of a mapping from queries in UXQuery to extended query trees, which makes it possible to use the approach presented in the previous chapters of this thesis.

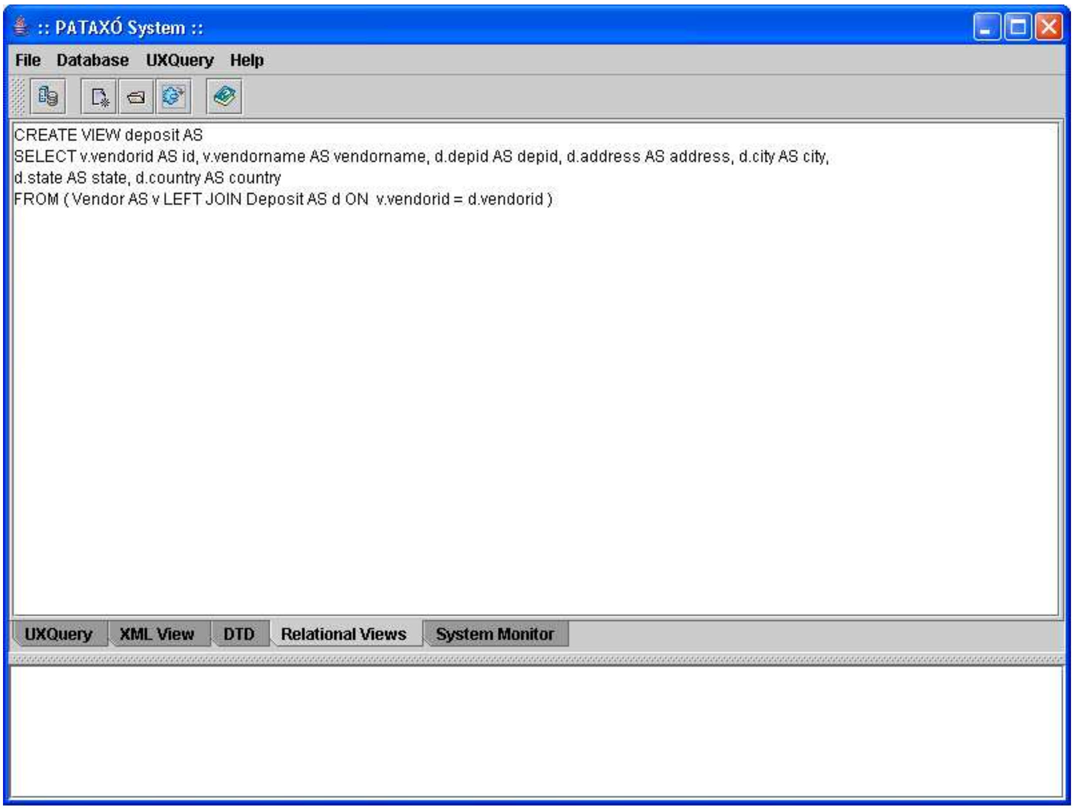
We have also done an analysis of UXQuery and query trees with respect to real world XML views. We show such evaluation in the next chapter.

We plan to improve the prototype in future work. One of the improvements we are planning to add is related to the locking strategy of the prototype. In this sense, we plan to add a comparison mechanism to PATAXÓ, which will help in cases where the XML view is generated and updated after a long period of time. In the current approach, the data used in the XML view stays locked until an update is issued (and committed), or a certain amount of time is elapsed (time out). In this new approach, no locking would be made when the XML view is constructed. When an update against the view is issued, the system would check if the database is still in the same status, and if so, translate the update to the base tables.



```
<!ELEMENT state (#PCDATA)>
<!ELEMENT idDeposit (depid)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT street (address)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT depid (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT vendor (vendorname, deposit*)>
<!ATTLIST vendor
          id CDATA #REQUIRED>
<!ELEMENT vendors (vendor*)>
<!ELEMENT deposit (idDeposit, address)>
<!ELEMENT vendorname (#PCDATA)>
```

Figure 7.14: *DTD* Tab of PATAXÓ System



```
CREATE VIEW deposit AS
SELECT v.vendorid AS id, v.vendorname AS vendorname, d.depid AS depid, d.address AS address, d.city AS city,
d.state AS state, d.country AS country
FROM ( Vendor AS v LEFT JOIN Deposit AS d ON  v.vendorid = d.vendorid )
```

Figure 7.15: *Relational Views* Tab of PATAXÓ System

8 EVALUATION

For purposes of presentation, the query tree language presented in this thesis was kept simple to highlight how the mapping of the query tree and updates are performed.

Query trees can be extended in a number of ways, for example to deal with grouping, aggregates, function applications and so on. An example of such extension can be found in Appendix A, where we allow *grouped values* which allow tuples that agree on a given value to be clustered together.

However, another consideration that must be kept in mind when extending the language is whether or not the relational views resulting from the XML view are updatable. The language presented in this thesis, with suitable restrictions on the way in which joins and nesting are performed with respect to keys and foreign keys in the underlying relational database, presents a subset of XQuery in which *side-effect free* updates can be defined, as discussed in Chapter 6. While grouped values do not affect these results, the addition of functions and aggregates would. Analogous to work on updating views in relational databases which restricts views to select-project-join queries, we have therefore initially decided against considering a richer language (although we plan to do so in future work).

To evaluate our language, we first discuss the restrictions in our form of queries, and what query trees can or cannot express. Second, we examine the power of expression of query trees, and compare it with existing proposals in literature. We have also analyzed the "practicality" of XML views constructed by query trees by collecting examples of real XML views extracted from relational databases and evaluating whether or not query trees can capture them. For these real XML views, query trees were sufficiently expressive.

8.1 Limitations of Query Trees

Although query trees are quite expressive, there are some restrictions.

Values must come from the relational database. We do not allow constants to be introduced as values in leaves, nor do we allow functions to calculate new values from values in the database. Allowing constant values in leaves is potentially useful (for example, to add a version number to the view), but they are not interesting from the perspective of updates to the relational database nor can they themselves be updated since they are not part of the database schema. Calculating a value from a set of values (e.g. taking the average of a relational column) creates a one to many mapping which cannot be updated; research on relational views also disallows

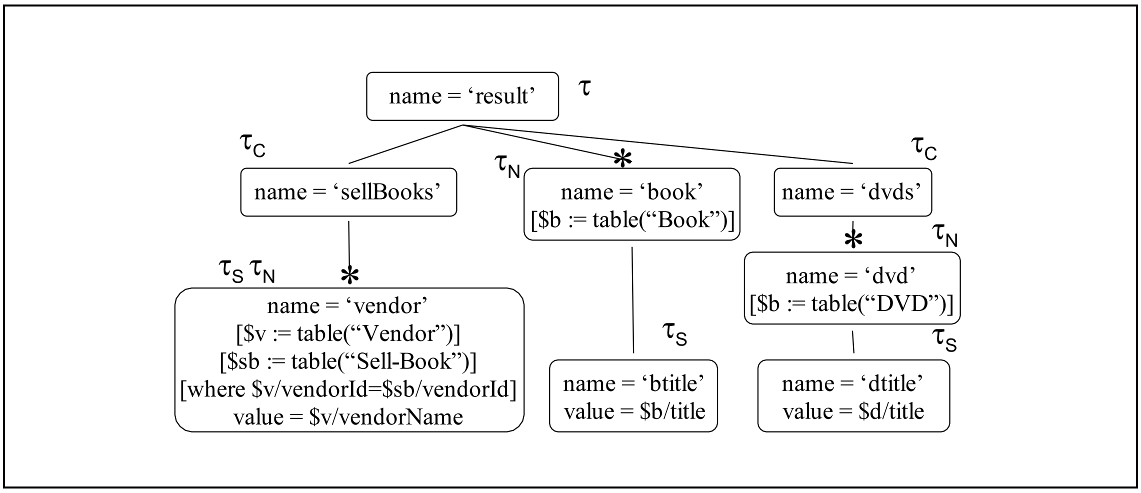


Figure 8.1: Example of query tree

this case. However, calculating a new value from a single value in the database (e.g. translating length in centimeters to length in inches) could be allowed as long the reverse function was also specified.

Queries are trees rather than graphs. This restriction disallows recursive queries, which are also disallowed in SilkRoute (FERNÁNDEZ et al., 2002). For example, suppose the relational database contained a relation Patriarchs(PName, CName) with instance {(John, Marc), (John, Chris), (Justin, John)}. An XML view of this that one might wish to construct would be:

```
<Patriarch>
 <Name>Justin</Name>
 <Children>
    <Name>John</Name>
    <Children> <Name>Marc</Name>
               <Name>Chris</Name>
    </Children>
 </Children>
</Patriarch>
```

Since recursive queries cannot be mapped to select-project-join queries, our technique would have to be extended significantly to reason about them.

On the other hand, query trees are flexible enough to represent heterogeneous structures (e.g. the view in Figure 4.2). It can also represent query trees with a repeating leaf node, as shown in Figure 8.1 (note that *vendor* is labeled with τ_N and τ_S). The XML view resulting from this query tree is as follows:

```
<result>
  <sellBooks>
    <vendor>Amazon</vendor>
    <vendor>Barnes and Nobel</vendor>
  </sellBooks>
  <book><btitle>Unix Network Programming</btitle></book>
  <book><btitle>Computer Networks</btitle></book>
  ...
  <dvds>
    <dvd><dtitle>Friends</dtitle></dvd>
    ...
  </dvds>
</result>
```

It turns out that XML views with heterogeneous content and repeating leaves arise frequently in practice, but that recursive views are not common. We therefore believe that the above restrictions do not limit the usefulness of our approach.

8.2 Power of Expression

We have shown how to transform the XML view update problem into the well studied relational view update problem. However, since our proposal is based on *query trees*, the next question is: Are query trees expressive enough to be used in practice? To answer this question, we compare the power of expression of query trees with SilkRoutes' *view forests* (FERNÁNDEZ et al., 2002); XPERANTO (SHANMUGASUNDARAM et al., 2001); and DB2 DAD files (CHENG; XU, 2000).

Since most of these proposals are based on the XQuery (BOAG et al., 2004) query language, we use XQuery on our comparison. In chapter 7, we have shown how a query in UXQuery can be translated to a query tree. However, it is also possible to do the opposite, that is, to generate a query in XQuery from a given query tree. In this section, we first show how a query tree can be translated to an XQuery query. Based on the structuring rules of query trees and on these translation rules, we then present an EBNF for the subset of XQuery that query trees are capable of expressing.

Given a query tree *qt*, an XQuery *xq* is generated by the *generateXQuery* algorithm (algorithm 8.1). The algorithm is recursive, and it starts with n being the root of *qt*. Function *value(n)* returns the value associated with a leaf node. For example, suppose node n has value $\$x/A$, then *value(n)* returns the expression $\$x/A$. Function *name(n)* returns the node name in *qt*. When n is an attribute, the function returns the name without "@". As an example, if n has name *@id*, then *name(n)* returns *id*.

As an example, the XQuery query corresponding to the query tree of Figure 4.2 is shown in Figure 8.2.

This translation algorithm assumes that each relational table X with attributes $A, B, C, ...$ is exported to XML as follows:

```
<X>
   <row>
      <A> ... </A>
      <B> ... </B>
      <C> ... </C>
      ...
   </row>
   ...
</X>
```

According to the translation algorithm and to the structuring rules of query trees, the XQuery queries corresponding to query trees follow the EBNF shown in Figure 8.3. Notice that this EBNF is not equal to the EBNF of UXQuery (Figure 7.3). The difference is that the EBNF of UXQuery has the `xnest` operator, while this EBNF considers pure XQuery operators (with the exception of the `table` input function). In fact, when we normalize a query in UXQuery to XQuery (according to

```
generateXQuery($n$)
case abstract_type($n$)
  $\tau|\tau_C$: buildElement($n$)
  $\tau_T|\tau_N$: table($n$)
  $\tau_G$: group($n$)
  $\tau_S$: leaf($n$)
end case

buildElement($n$)
let $tag$ = "name($n$)"
for each attribute $c$ in children($n$) whose value is $grouped$ do
  add "name($c$) = '$\$$ name($c$)/text()'"to $tag$
end for
for each other remaining attribute $c$ in children($n$) do
  add "name($c$) = 'value($c$)/text()'"to $tag$
end for
print "$<$ $tag$ $>$"
for each non-attribute $c$ in children($n$) do
  generateXQuery($c$)
end for
print "$<$/name($n$)$>$"

leaf($n$)
if $n$ has a $grouped$ value then
  print "$<$name($n$)$>${$\$$name($n$)/text()}$<$/name($n$)$>$"
else
  print "$<$name($n$)$>${value($n$)/text()}$<$/name($n$)$>$"
end if

table($n$)
print "{"
for each source annotation binding a table $X$ to a variable $\$x$ in $n$ do
  print "for $\$x$ in document('X.xml')//row"
end for
Let $W$ be the set of where annotations in $n$
Let $count = 1$
for each $w$ in $W$ do
  if $count > 1$ then
    print "AND w"
  else
    print "WHERE w"
  end if
  $count = count + 1$
end for
if $n$ is a child of a node $a$, and abstract_type($a$) = $\tau_G$ then
  let $G = \{g_1, \ldots\ g_s\}$ be the GROUP children of $n$
  for each $g_i$ in $G$ do
    Let value($g_i$) be of the form GROUP($\$x_1/A_1$ | ... | $\$x_k/A_k$)
    Find each of the $x_i$ in the value of $g_i$ that is declared in a source annotation in $n$
    if $count > 1$ then
      print "AND $\$$ name($g_i$) = $\$x_i/A_i$"
    else
      print "WHERE $\$$ name($g_i$) = $\$x_i/A_i$"
    end if
    $count = count + 1$
  end for
end if
print "return"
buildElement($n$)
print "}"

group($n$)
let $G = \{g_1, \ldots\ g_s\}$ be the GROUP children of $n$
let $S$ be the set of source annotations in $m$, for all starred nodes $m$ that are children of $n$
print "{"
for each $s$ in $S$ binding a variable $\$x$ to a table $X$ do
  print "let $\$x'$ := document('X.xml')//row"{Notice that variable $\$x$ is primed in the generated $let$ expression}
end for
for each $g_i$ in $G$ do
  Let value($g_i$) be of the form GROUP($\$x_1/A_1$ | ... | $\$x_k/A_k$)
  print "for $\$$ name($g_i$) in distinct values ($\$x'_1/A_1$ | ... | $\$x'_k/A_k$ {Notice again the use of primed variables. They correspond to
  the variables bound by the $let$ expression on line (68)}
end for
buildElement($n$)
print "}"
```

Algorithm 8.1: The *generateXQuery* algorithm

```
<vendors>
   {for $v in document("Vendor.xml")//row
    return
    <vendor id='{$v/vendorId/text()}'>
       <vendorName>{$v/vendorName/text()}</vendorName>
       <address>
          <state>{$v/state/text()}</state>
          <country>{$v/country/text()></country>
          <web>
             <url>{$v/url/text()}</url>
          </web>
       </address>
       {let $sb' := document("Sell-Book.xml")//row
        let $b' := document("Book.xml")//row
        let $sd' := document("Sell-DVD.xml")//row
        let $d' := document("DVD.xml")//row
        for $price in distinct-values($sb'/price | $sd'/price)
        return
        <products price='{$price/text()}'>
           {for $sb in document("Sell-Book.xml")//row
            for $b in document("Book.xml")//row
            where $sb/vendorId = $v/vendorId and
                  $b/isbn = $sb/isbn and
                  $price = $sb/price
            return
            <book>
               <btitle>{$b/title}</btitle>
               <isbn>{$b/isbn}</isbn>
            </book>
            }
            {for $sd in document("Sell-DVD.xml")//row
            for $d in document("DVD.xml")//row
            where $sd/vendorId = $v/vendorId and
                  $d/asin = $sd/asin and
                  $price = $sd/price
            return
            <dvd>
               <dtitle>{$d/title}</dtitle>
               <asin>{$d/asin}</asin>
            </dvd>
            }
        </products>
       }
    </vendor>
   }
</vendors>
```

Figure 8.2: XQuery representation of query tree of Figure 4.2

```
[1]  XQuery          ::= QueryBody
[2]  QueryBody       ::= ElmtConstructor
[3]  ElmtConstructor ::= "<" QName AttList "/>" | "<" QName AttList? ">" ElmtContent+ "</" QName ">"
[4]  ElmtContent     ::= ElmtConstructor | EnclosedExpr+
[5]  AttList         ::= ((QName "=" AttValue)?)+
[6]  AttValue        ::= ('"' AttValueContent '"') | ("'" AttValueContent "'")
[7]  AttValueContent ::= "{" PathExprAtt "}"
[8]  PathExprAtt     ::= "$" VarName "/" QName "/" NodeTest
[9]  VarName         ::= QName
[10] EnclosedExpr    ::= "{" (FWRExpr | LFWRExpr | PathExpr) "}"
[11] Expr            ::= OrExpr
[12] OrExpr          ::= AndExpr ( "or"  AndExpr )*
[13] AndExpr         ::= ComparisonExpr ("and" ComparisonExpr)*
[14] FWRExpr         ::= ((ForClause)+ WhereClause? OrderByClause? "return")* ElmtConstructor
[15] LFWRExpr        ::= ((LetClause)+ (ForClauseDistinct)+ WhereClause? OrderByClause? "return")*
                         ElmtConstructor
[16] ComparisonExpr  ::= ValueExpr (GeneralComp ValueExpr )?
[17] ValueExpr       ::= PathExpr | PrimaryExpr
[18] PathExpr        ::= "$" VarName "/" QName ("/" NodeTest)?
[19] NodeTest        ::= TextTest
[20] TextTest        ::= "text" "(" ")"
[21] ForClause       ::= "for" "$" VarName "in" DocExpr
[22] ForClauseDistinct ::= "for" "$" VarName "in distinct-values" UnionExpr
[23] DocExpr         ::= "table (" '"' QName '"' ")//row" | "table (" "'" QName "'" ")//row"
[24] WhereClause     ::= "where" Expr
[25] GeneralComp     ::= "=" |  "!=" |  "<" |  "<=" |  ">" |  ">="
[26] OrderByClause   ::= "order" "by" OrderSpecList
[27] OrderSpecList   ::= OrderSpec ("," OrderSpec)*
[28] OrderSpec       ::= PathExpr
[29] PrimaryExpr     ::= Literal | ParenthesizedExpr
[30] Literal         ::= NumericLiteral |  StringLiteral
[31] NumericLiteral  ::= IntegerLiteral |  DecimalLiteral |  DoubleLiteral
[32] ParenthesizedExpr  ::= "(" Expr? ")"
[33] UnionExpr  ::= "(" "$" VarName "/" QName ( ("union" | "|")  "$" VarName "/" QName)* ")"
```

Figure 8.3: EBNF of the subset of XQuery corresponding to query trees

the rules on Section 7.1.1) the result is a query conforming to the EBNF of Figure 8.3.

XPERANTO (SHANMUGASUNDARAM et al., 2001) is quite expressive, and can express all queries in XQuery. View forests (FERNÁNDEZ et al., 2002) are capable of expressing any query in the XQueryCore that does not refer to element order, use recursive functions or use is/is not operators. Query trees present the same limitations as (FERNÁNDEZ et al., 2002), and are also not capable of expressing *if/then/else* expressions; sequence of expressions (since we require that the result of the query always be an XML document); function applications; arithmetic and set operations. Input functions are also a limitation of query trees. It is not possible to bind results of expressions to variables. Variables can only be bound to relational tables, while in SilkRoute, they can be bound to arbitrary expressions.

DB2 XML Extender provides mappings from relations to XML through DAD (*Data Access Definition*) files. DB2 DAD files with `RDB_node` method are equivalent to query trees in expressive power, since all the data come directly from the relational database and functions cannot be applied over the retrieved data. This is meaningful, since DB2 DAD files represent features that are useful in practice, and because this subset can easily be mapped to relational views.

It is important to state, however, that all of these three approaches are focused in querying XML views, and not on updates. Therefore, it is understandable that they are more expressive (with the exception of DB2 DAD files) than our approach. We had to sacrifice a little bit on expressive power to favor updatability.

8.3 Real applications

We were able to obtain three real world applications:

- the XBrain Project (XBRAIN PROJECT, 2003);
- a Tobacco company (KROTH, 2003); and
- the Mondial Database (MAY, 1999).

The views produced by these applications are available in Appendix C.

The XBrain Project is an application of SilkRoute (FERNÁNDEZ et al., 2002). The application queries a brain mapping database, over which an XML *public view* is defined.

The Tobacco company example is the most interesting. The company needs to send monthly reports to a Tobacco Producer's Association[1]. The report shows data about the producers from whom the company bought tobacco, as well as prices, quantities, etc. The company stores all the transactions in a relational database, and at the end of the month it generates an XML report containing all the information solicited by the Producer's Association.

The Mondial database is a case study for information extraction and integration. Facts about global geography are extracted from the Web and integrated in a very large database. An XML view over this database is then provided.

These applications highlight several characteristics of real XML views: They are large and complex, typically involving several relational tables. They are also well structured, which is no surprise since the source data is relational and therefore structured. Additionally, joins are made through keys and foreign keys - a desired property of updatable views. All of these views can be expressed using query trees except for the portion of the Mondial view shown below:

```
<religions percentage="70">Muslim</religions>
<religions percentage="10">Roman Catholic</religions>
<religions percentage="20">Albanian Orthodox</religions>
```

As presented so far, query trees cannot represent this XML view since *religions* has a child *percentage* but must be a leaf to have a value, a contradiction. The same problem occurs in the Mondial view for an element called *ethnicgroups*. Note that allowing attributes within atomic elements is similar to allowing mixed content elements.

It turns out that query trees can easily be extended to deal with this case. We can introduce a special attribute called *textContent* to represent the text content of the element *religions*. The tree will be processed with this additional attribute, and transformed back to its original structure before being presented to the user.

There are also three problematic attributes in the Mondial view: attribute *membership* of element *country*, and the attributes *id* and *is_ country_ capital*. These attributes require computation to construct their value, and are therefore not addressed by our work as discussed earlier in this chapter.

[1] We omit the company and the association name for copyright reasons.

8.4 Normalized XML documents

A proposal to extract a nested normalized XML document (ARENAS; LIBKIN, 2002) from a relational database is presented in (WEIS, 2003). The proposal explores keys and foreign key constraints to build a graph of dependencies between tables. By traversing this graph, it is possible to decide which table(s) will be a top-level element in the resulting XML document and how the remaining tables will be nested under this element. When nesting a given table leads to redundancy, it is placed directly below the root, and relationships are expressed using IDs and IDREFs. Such views can be easily expressed using query trees, and are updatable for all correct insertions, deletions and modifications.

8.5 XQuery use cases

For completeness, we also analyzed the relational use cases of XQuery (CHAMBERLIN et al., 2003). Of the eighteen queries presented in (CHAMBERLIN et al., 2003), our query trees are capable of expressing only two (Q3 and Q4). This is mainly caused by the use of aggregate operations. We believe that these use cases highlight the difference between queries and view definitions, rather than demonstrating shortcomings of query trees. Aggregate operators and specialized functions are typically not considered in work on updating views.

8.6 XML documents stored in relations

XML documents are frequently mapped to relational databases for efficient storage. We analyzed the most popular approach, hybrid inlining (SHANMUGASUNDARAM et al., 1999), to check if the XML view definitions resulting from this mapping could be expressed by query trees.

We analyzed six different XML documents. Three of them represent information about courses of different universities[2]. We also analyzed the SIGMOD Record (SIGMOD RECORD, 2002), the DBLP in XML (LEY, 2003) and the action XML file of XMark (BUSSE et al., 2002).

All of the three course documents are fully compatible with query trees. DBLP and XMark are also compatible except for one element that has mixed content (*title* in DBLP and *text* in XMark). Although hybrid inlining does not support mixed content, we mapped it by assuming an upper bound n on the number of fragments of text content for an element, and used special attributes called $textContent_1$, ..., $textContent_n$ to capture the fragmented text values. The resulting mapping could be expressed using query trees.

As for updates, since hybrid inlining introduces artificial primary keys, these keys must be projected in the view in order to update the underlying relational database through the reconstructed document. This can be handled by introducing an *id* attribute that holds the primary keys in elements that represent database tuples.

[2] `http://www.cs.washington.edu/research/xmldatasets/www/`

8.7 Chapter Remarks

Query trees are able to capture many of the "real" XML views of relational databases that we were able to find. The evaluations were also incredibly valuable to identify extensions that should be made to our definitions, such as those to handle atomic elements with attributes and mixed content elements.

As for the power of expression, we believe that the limitations of our approach are completely justified by the need of updating the resulting view. In fact, work on updates through relational views also considers only a subset of queries, namely SELECT, PROJECT, JOIN.

9 CONCLUSIONS

We believe that this thesis has given the first step towards the solution of the (previously unsolved) problem of updates through XML views over relational databases. The proposed solution takes advantage of existing work on updates through relational views. The XML views are constructed using *query trees*, which allow nesting as well as heterogeneous sets of tuples, and can be used to capture most of the features we encountered in real views.

The main contributions of this thesis are the mapping of the XML view to a set of underlying relational views, and the mapping of updates on an XML view instance to a set of updates on the underlying relational views. By providing these mappings, the XML update problem is reduced to the relational view update problem and existing techniques on updates through relational views (DAYAL; BERNSTEIN, 1982a; KELLER, 1985; BANCILHON; SPYRATOS, 1981; LECHTENBÖRGER, 2003) can be leveraged. As an example, we show how to use the approach of (DAYAL; BERNSTEIN, 1982a) to produce side-effect free updates on the underlying relational database.

Another benefit of our approach is that query trees are agnostic with respect to a query language. Query trees represent an intermediate query form, and any (subset of an) XML query language that can be mapped to this form could be used as the top level language. In particular, we have implemented our approach in a system called Pataxó that uses a subset of XQuery to build the XML views and translates XQuery expressions into query trees as an intermediate representation.

Similarly, our update language represents an intermediate form that could be mapped into from a number of high-level XML update languages. In our implementation, we use a graphical user interface which allows users to click on the update point or (in the case of a set oriented update) specify the path in a separate window and see what portions of the tree are affected.

The remaining of this chapter is structured as follows. We present a detailed list of contributions in Section 9.1. A list of published papers resulting from this thesis is presented in Section 9.2. Section 9.3 compares our work with related work. Future work is discussed in Section 9.4.

9.1 Contributions

We now present a detailed list of the main contributions of this thesis:

A formalism to specify XML views over relational databases. Query trees can be used as an intermediate representation of a top-level query language

(BRAGANHOLO; DAVIDSON; HEUSER, 2004a). This makes our approach syntax independent. Any language that can be mapped to query trees can be used to specify the views. We have made an evaluation of the power of expression of query trees that shows that query trees are expressive enough to be applied in practice.

Mapping from XML views to relational views. Given an XML view specified by a query tree, we provide algorithms to map it to a set of corresponding relational views. We also provide algorithms to translate updates over the XML view to updates over the corresponding relational views (BRAGANHOLO; DAVIDSON; HEUSER, 2004a). We thus transform an open problem – that of updating relational databases through XML views – into an existing problem – that of updating relational databases through relational views.

An XML updatability study. We have made an updatability study of XML views constructed by query trees based on a preliminary study (BRAGANHOLO; DAVIDSON; HEUSER, 2003a) that uses the nested relational algebra as the view definition language. Our study is based on the side-effect free idea of (DAYAL; BERNSTEIN, 1982a), and uses their theory to identify classes of updatable views. We have identified three different classes of views: (i) one that is updatable for all possible insertions; (ii) one that is updatable for all possible insertions, deletions and modifications; and (iii) one whose updatability can be reasoned about using Theorem 6.1.

A subset of XQuery to specify XML views over relational databases. We have proposed and implemented a subset of XQuery which is capable of constructing XML views over relational databases (BRAGANHOLO; DAVIDSON; HEUSER, 2003b). UXQuery uses query trees as an intermediate representation to map the resulting XML view to relational views.

PATAXÓ. We have implemented our ideas in the Pataxó system to show the feasibility of our approach. Pataxó uses UXQuery as the view definition language, and uses the approach of (DAYAL; BERNSTEIN, 1982a) to translate updates from the relational views to the underlying relational database.

9.2 Published Papers

This thesis has resulted in several published papers:

1. BRAGANHOLO, V.; DAVIDSON, S.; HEUSER, C. From XML View Updates to Relational View Updates: old solutions to a new problem. In: Proceedings of VLDB - International Conference on Very Large Databases, Toronto, Canada, 2004. Pages 276–287(BRAGANHOLO; DAVIDSON; HEUSER, 2004a).

2. BRAGANHOLO, V.; DAVIDSON, S.; HEUSER, C. UXQuery: building updatable XML views over relational databases. In: Proceedings of SBBD - Brazilian Symposium on Databases, Manaus, AM, Brazil, 2003. Pages 26–40 (BRAGANHOLO; DAVIDSON; HEUSER, 2003b). This paper was nominated to the *José Mauro de Castilho Best Paper Award.*

3. BRAGANHOLO, V.; DAVIDSON, S.; HEUSER, C. On the updatability of XML views over relational databases. In: Proceedings of WebDB, held in conjunction with SIGMOD/PODS, San Diego, California, 2003. Pages 31–36. (BRAGANHOLO; DAVIDSON; HEUSER, 2003a).

4. BRAGANHOLO, V.; HEUSER, C. Updating Relational Databases through XML Views. In: Proceedings of WTDBD - I Workshop de Teses e Dissertações em Banco de Dados, held in conjunction with SBBD 2002, Gramado, RS, Brazil, 2002. Pages 67–71. (BRAGANHOLO; HEUSER, 2002a).

5. BRAGANHOLO, V.; HEUSER, C.; VITTORI, C. Updating Relational Databases through XML Views. In: Proceedings of IIWAS 2001 - Third International Conference on Information Integration and Web-based Applications & Services, Linz, Austria, 2001. (BRAGANHOLO; HEUSER; VITTORI, 2001).

There are also some papers not directly related with the thesis, but that are results of the initial studies on XML.

1. BRAGANHOLO, V.; HEUSER, C. XML Schema, RDF(S) e UML: uma comparação. (Title in English: "XML Schema, RDF(S) and UML: a comparison"). In: Proceedings of IDEAS 2001 - 4th Iberoamerican Workshop on Requirements Engineering and Software Environments, Santo Domingo, Heredia, Costa Rica, 2001. Pages 78–90. (BRAGANHOLO; HEUSER, 2001).

2. MELLO, R.; DORNELES, C.; KADE, A.; BRAGANHOLO, V.; HEUSER, C. Dados Semi-Estruturados. (Title in English: "Semistructured Data"). In: Proceedings of SDDB 2000, João Pessoa, PB, 2000. Tutorial. (MELLO et al., 2000).

We have also developed several technical reports. They are longer versions of the published papers, including theorem proofs and details that were omitted from the papers due to space restrictions.

1. BRAGANHOLO, V.; DAVIDSON, S.; HEUSER, C. Propagating XML View Updates to a Relational Database. Technical Report RP-341. PPGC, UFRGS. Porto Alegre, RS, Brazil. 2004. (BRAGANHOLO; DAVIDSON; HEUSER, 2004b).

2. BRAGANHOLO, V.; DAVIDSON, S.; HEUSER, C. Using XQuery to build updatable XML views over relational databases. Technical Report MS-CIS-03-18. University of Pennsylvania. Philadelphia, PA, USA. 2003. (BRAGANHOLO; DAVIDSON; HEUSER, 2003c).

3. BRAGANHOLO, V.; DAVIDSON, S.; HEUSER, C. Reasoning about the updatability of XML views over relational databases. Technical Report MS-CIS-03-13. University of Pennsylvania. Philadelphia, PA, USA. 2003. (BRAGANHOLO; DAVIDSON; HEUSER, 2003d).

4. BRAGANHOLO, V.; HEUSER, C. Updating Relational Databases through XML Views. Technical Report TP-328. UFRGS, Porto Alegre, RS, Brasil, 2002. 61 pages. (BRAGANHOLO; HEUSER, 2002b).

Table 9.1: Comparison with Related Work

	Number of nodes in the tree	Number of SQL queries	Data Input
Query Trees	n	x, where x is the number of τ_N nodes on the tree, and $x < n$	`table` input function
SilkRoute	n	n	default variables
XPERANTO	n	1	`view` input function

Finally, but not less important, this thesis has also originated four Undergraduate Final Projects. All of them were advised by Carlos Heuser (advisor) and Vanessa Braganholo (co-advisor).

1. AGRA, A. Implementação de uma Proposta para Atualização de Bancos de Dados através de Visões. (Title in English: "An implementation of a proposal for updating Relational Databases through views"). Instituto de Informática, UFRGS. July of 2004. (AGRA, 2004).

2. NONNENMACHER, M. Uma Extensão para a API DOM para Suportar Atualização de Bancos de Dados Relacionais através de Visões XML. (Title in English: "An extention to the DOM API to support updates through XML views"). Instituto de Informática, UFRGS. March of 2003. (NONNENMACHER, 2003).

3. VICTOLLA, F. Ferramenta para Automatização do Projeto Lógico de um Banco de Dados e da Implementação do SGBD usando Transformações XSLT. (Title in English: "A tool to automate the Logical Design of a Database and its implementation using XSLT Transformations"). Instituto de Informática, UFRGS. May of 2002. (VICTOLLA, 2002).

4. FEIJÓ, D. Extrator de Esquema de um Banco de Dados. (Title in English: "Schema extractor of a Database"). Instituto de Informática, UFRGS. May of 2002. (FEIJÓ, 2002).

9.3 Comparison with Related Work

We now briefly compare our approach with SilkRoute and XPERANTO. This comparison comprises the way we generate the view, and what are the SQL queries generated. Table 9.1 summarizes the comparison.

9.3.1 SilkRoute

In SilkRoute, for a view forest with n nodes, n SQL queries are generated. Each of those queries are executed and one XML element is generated for each tuple in the result set associated to n. Nodes are then glued together using variable bindings to find out how to connect them. In our approach, we do not use the SQL engine to generate the XML view. Instead, we use an XQuery engine to generate the view, and we produce SQL statements to map an XML view to a set of relational views.

To illustrate, assume the query tree of Figure 4.2 and the view forest of Figure 3.3. They both generate the XML view of Figure 1.1. For that view forest we have 26 SQL queries, while for the query tree we generate 2 relational views. The interesting point is to compare the SQL queries generated for nodes *book* and *dvd*. In the view forest, we have:

```
N1.1.4.1(<book>)       := SELECT * FROM Vendor v, Book b, SellBook sb
                          WHERE sb.vendorId = v.vendorId AND b.isbn = sb.isbn

N1.1.4.2(<dvd>)        := SELECT * FROM Vendor v, DVD d, SellDVD sd
                          WHERE sd.vendorId = v.vendorId AND d.asin = sd.asin
```

For the query tree, we have:

```
<book> :=
SELECT v.vendorId AS id, v.vendorName AS vendorName,
 v.state AS state, v.country AS country,
 sb.price AS bprice, b.isbn AS isbn, b.title AS btitle
FROM (Vendor AS v LEFT JOIN (SellBook AS sb INNER JOIN
Book AS B ON b.isbn=sb.isbn) ON v.vendorId=sb.vendorId);

<dvd> :=
SELECT v.vendorId AS id, v.vendorName AS vendorName,
 v.state AS state, v.country AS country,
 sd.price AS dprice, d.asin AS asin, d.title AS dtitle
FROM (Vendor AS v LEFT JOIN (SellDVD AS sd INNER JOIN
DVD AS d ON d.asin=sd.asin) ON v.vendorId=sd.vendorId)
```

Notice that the same base tables are used, and also the same join conditions. The difference is the SELECT clause, since in SilkRoute a star was used, while in our approach we select each attribute by its name. The star was used in SilkRoute because there will be specific queries that will select the appropriate values for the leaf nodes. Another difference is that we use LEFT JOIN, while SilkRoute uses regular join. The reason for this difference is the same - there is one SQL query for each node. In this way, the SQL query for node *vendor* will select only data from table *Vendor* (see the SQL query corresponding to node *vendor* on Section 3.1.1). This will retrieve all vendors, despite the fact that they sell or do not sell books or DVDs. In our case, we use LEFT JOIN to make sure a vendor appears in the view even if it does not sell any book or DVD.

Another difference between SilkRoute and our approach is the way in which data is made available to queries. In our approach, we have an input function `table` which is used to access relational data. In SilkRoute, the default variables `$CanonicalView` and `$PublicView` are used to this purpose.

9.3.2 XPERANTO

In XPERANTO (SHANMUGASUNDARAM et al., 2001), a single SQL query is generated for each view definition. This query, however, has several subqueries. It is not possible to know the exact syntax of the SQL queries generated by XPERANTO, since we have only access to its internal representation (XQGM). One similarity to our work, however, is the use of outer joins to preserve top elements that have no descendants. In XPERANTO, this is done after a process called query decorrelation.

Another similarity with our approach is the input function XPERANTO uses to access relational data. They have defined an input function `view`, which is the only input function available. This function accesses other XML views, as well as the default view, which represents the database in a canonical XML. Our input function `table` plays a similar role, but it can only access relational tables, not other XML views.

9.4 Future Work

We now describe in details some open problems that could lead to future research work:

Long Transactions An interesting direction is to support "long transactions". The scenario is as follows: a user specifies an XML view, and this view is sent to another application (or user) to be updated. The updated view may take hours or days to return to the system. In this case, locking is not a good solution. One possibility is to compare the original database state with the current database state to detect if anything has changed – this can be done simply by comparing the original view with a new view reconstructed from the current database state, since only portions of data relevant to the view matters in this case. If nothing has changed since then, then the updates could be propagated to the database. However, if something has changed, then it is necessary to carefully study the possible actions. The Software Configuration Management (ISO, 1995; IEEE, 1998, 1987) area can bring some light to this problem. It studies both how to identify conflicts in data, and how to deal with them.

Another problem in this scenario is to identify what had changed in the view. Since we are using long transactions, we are assuming that the updates will not be issued using an update language. Instead, the user or application would alter (edit) the view directly. The changes would need to be "discovered" by comparing the original XML view with the updated one, and finding *deltas*. There is a master student that is currently starting to look at this problem.

User feedback Another important open problem in our approach is on how to explain to users the reasons why a given update was refused. This happens because of the mismatch between what users see (and how he understands the system), and how updates are being translated. As an example, suppose the user wants to delete a subtree t at the XML view. This update is translated to a deletion over the corresponding relational view V. Suppose that this update fails because the translation procedure found out that there would be a side-effect. The side-effect was detected using the relational views and its constraints, which the user is not aware of. The question is: how to explain to the user the reason of the rejection of that update?

This gets even more complicate in our scenario, since any translation procedure of updates through relational views can be used. Consequently, different notions of correctness can be adopted, and different error messages can happen.

A nice future direction is then to investigate if it is possible to find a generic mapping that explains the results of updates to users.

Expressive Power Query trees are obviously less expressive than XQuery. In particular, they are not capable of expressing aggregations and arbitrary restructuring in the XML view. Although this is a trade-off imposed by our goal of updating the relational database through XML views, it may be possible to recognize updatable portions of views expressed in a more general language. For example, if the view presented author information and the total number of papers they had written, it is still possible to update author information even if updating the total number of papers is not allowed. We plan to overcome this limitation in future work.

We also plan to extend query trees to support mixed content elements and leaf nodes with attributes using the directions suggested in Sections 8.3 and 8.6.

Order Another interesting direction is to support order. There are cases where preserving order is important. An example is a database that stores papers and their authors. Then, changing the order of the authors in the view means something that can not be disregarded. We plan to study this carefully and include order support in our approach. This becomes more interesting with the adoption of a more powerful update language. We plan to adopt one as soon as it becomes available.

Updatability We also plan to study the updatability of XML views using other proposals of updates through relational views in the literature. It would be interesting to compare the updatability results using different notions of correct updates found in literature.

Optimization of view generation Another point we plan to address is the optimization of the view generation process. We believe that we can use the relational engine to execute the data extraction part of the view definition. Our initial studies show that it would not be difficult to implement an approach similar to SilkRoute in order to improve the way XML views are generated. However, we do not plan to generate as much SQL statements as SilkRoute does, as this is also inefficient. We believe it is possible to use the SQL statements of the relational views corresponding to a query tree to generate the view.

Redundancy Detection We are currently studying the database constraints and the view definition query to try do identify redundancy in the XML view. As illustrated in this thesis, redundancy is a source of bad updates when we try to update just some of the occurrences of a redundant data. By identifying the redundant portions of an XML view, we can help users to construct update paths that touches *all* the occurrences of a given redundant data.

REFERENCES

ABITEBOUL, S.; CLUET, S.; MILO, T. Querying and Updating the File. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 1993, Dublin, Ireland. **Proceedings. . .** San Francisco: Morgan Kaufmann, 1993. p.73–84.

ABITEBOUL, S.; HULL, R.; VIANU, V. **Foundations of Databases**. [S.l.]: Addison-Wesley, 1996.

ABITEBOUL, S.; QUASS, D.; MCHUGH, J.; WIDOM, J.; WIENER, J. The Lorel Query Language for Semistructured Data. **International Journal on Digital Libraries**, [S.l.], v.1, n.1, p.68–88, 1997.

AGRA, A. **Implementação de uma Proposta para Atualização de Bancos de Dados através de Visões**. 2004. Projeto de Diplomação — Instituto de Informática, UFRGS, Porto Alegre, RS, Brasil.

AGRAWAL, D.; ABBADI, A. E.; SINGH, A.; YUREK, T. Efficient View Maintenance at Data Warehouses. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD, 1997, Tucson, Arizona. **Proceedings. . .** [S.l.: s.n.], 1997. p.417–427.

APACHE SOFTWARE FOUNDATION. **Xerces Java Parser**. 2000. Avaliable at: <`http://xml.apache.org/xerces-j/`>. Visited on July 10, 2003.

APACHE SOFTWARE FOUNDATION. **Apache Xindice**. 2002. Available at: <`http://xml.apache.org/xindice`>. Visited on Feb. 2, 2004.

ARENAS, M.; LIBKIN, L. A Normal Form for XML Documents. In: INTERNATIONAL CONFERENCE ON PRINCIPLES OF DATABASE SYSTEMS, PODS, 2002, Madison, Wisconsin. **Proceedings. . .** [S.l.: s.n.], 2002.

BANCILHON, F.; SPYRATOS, N. Update Semantics of Relational Views. **ACM Transactions on Database Systems, TODS**, [S.l.], v.6, n.4, Dec. 1981.

BARSALOU, T.; SIAMBELA, N.; KELLER, A. M.; WIEDERHOLD, G. Updating Relational Databases through Object-Based Views. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD, 1991, Denver, CO. **Proceedings. . .** [S.l.: s.n.], 1991. p.248–257.

BOAG, S.; CHAMBERLIN, D.; FERNANDEZ, M. F.; FLORESCU, D.; ROBIE, J.; SIMéON, J. **XQuery 1.0**: an XML query language. Available at: <`http://www.`

w3.org/TR/2004/WD-xquery-20040723/>. Visited on Aug. 10, 2004. W3C Working Draft.

BOHANNON, P.; GANGULY, S.; KORTH, H.; NARAYAN, P.; SHENOY, P. Optimizing View Queries in ROLEX to Support Navigable Result Trees. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 2002, Hong Kong, China. **Proceedings. . .** San Francisco: Morgan Kaufmann, 2002.

BONIFATI, A.; BRAGA, D.; CAMPI, A.; CERI, S. Active XQuery. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 2002, San Jose, California. **Proceedings. . .** [S.l.: s.n.], 2002.

BONIFATI, A.; FLESCA, S.; PUGLIESE, A. **Semantic Issues of XML Updates**. [S.l.]: Istituto di Calcolo e Reti ad Alte Prestazioni, 2003. (Technical Report RT-ICAR-CS-03-15).

BOUCHOU, B.; ALVES, M. H. F. Updates and Incremental Validation of XML documents. In: INTERNATIONAL WORKSHOP ON DATABASE PROGRAMMING LANGUAGES, DBPL, 2003, Potsdam, Germany. **Proceedings. . .** Berlin: Springer, 2003. p.216–232. (Lecture Notes in Computer Science, v.2921).

BRAGANHOLO, V.; DAVIDSON, S. B.; HEUSER, C. A. On the updatability of XML views over relational databases. In: INTERNATIONAL WORKSHOP ON THE WEB AND DATABASES, WEBDB, 2003, San Diego, CA. **Proceedings. . .** [S.l.: s.n.], 2003.

BRAGANHOLO, V.; DAVIDSON, S. B.; HEUSER, C. A. UXQuery: building updatable XML views over relational databases. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, SBBD, 2003, Manaus, AM, Brasil. **Anais. . .** Belo Horizonte: Departamento de Ciência da Computação/UFMG, 2003. p.26–40.

BRAGANHOLO, V.; DAVIDSON, S. B.; HEUSER, C. A. **Using XQuery to build updatable XML views over relational databases**. [S.l.]: Department of Computer and Information Science, University of Pennsylvania, 2003. (MS-CIS-03-18).

BRAGANHOLO, V.; DAVIDSON, S. B.; HEUSER, C. A. **Reasoning about the updatability of XML views over relational databases**. [S.l.]: Department of Computer and Information Science, University of Pennsylvania, 2003. (MS-CIS-03-13).

BRAGANHOLO, V.; DAVIDSON, S. B.; HEUSER, C. A. From XML View Updates to Relational View Updates: old solutions to a new problem. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 2004, Toronto, Canada. **Proceedings. . .** San Francisco: Morgan Kaufmann, 2004. p.276–287.

BRAGANHOLO, V.; DAVIDSON, S. B.; HEUSER, C. A. **Propagating XML View Updates to a Relational Database**. Porto Alegre, RS, Brasil: UFRGS, 2004. (TR-341).

BRAGANHOLO, V.; HEUSER, C. A. XML Schema, RDF(S) e UML: uma comparaçao. In: IBEROAMERICAN WORKSHOP ON REQUIREMENTS ENGINEERING AND SOFTWARE ENVIRONMENTS, IDEAS, 2001, Santo Domingo, Heredia, Costa Rica. **Proceedings. . .** [S.l.: s.n.], 2001. p.78–90.

BRAGANHOLO, V.; HEUSER, C. A. Updating Relational Databases through XML Views. In: WORKSHOP DE TESES E DISSERTAÇÕES EM BANCO DE DADOS, WTDBD, 2002, Gramado, RS, Brasil. **Anais...** [S.l.: s.n.], 2002. p.67–71.

BRAGANHOLO, V.; HEUSER, C. A. **Updating Relational Databases through XML Views**. Porto Alegre, RS, Brasil: PPGC: UFRGS, 2002. (TR-328).

BRAGANHOLO, V.; HEUSER, C. A.; VITTORI, C. Updating Relational Databases through XML Views. In: INTERNATIONAL CONFERENCE ON INFORMATION INTEGRATION AND WEB-BASED APPLICATIONS & SERVICES, IIWAS, 2001, Linz, Austria. **Technical Sessions...** Wien: Österreichische Computer Gesellschaft, 2001. p.85–94.

BRAY, T.; HOLLANDER, D.; LAYMAN, A. **Namespaces in XML**. 1999. Available at: <http://www.w3.org/TR/REC-xml-names/>. Visited on Aug. 10, 2004. W3C Recommendation.

BRAY, T.; PAOLI, J.; SPERBERG-MCQUEEN, C. M.; MALER, E.; YERGEAU, F. **Extensible Markup Language (XML) 1.0 (Third Edition)**. 2004. Available at: <http://www.w3.org/TR/2004/REC-xml-20040204/>. Visited on Aug. 10, 2004. W3C Recommendation.

BUSSE, R.; CAREY, M.; FLORESCU, D.; KERSTEN, M.; MANOLESCU, I.; SCHMIDT, A.; WAAS, F. **XMARK - An XML Benchmark Project**. Available at: <http://monetdb.cwi.nl/xml/downloads.html>. Visited on Oct. 10, 2003.

CAREY, M. J.; FLORESCU, D.; IVES, Z. G.; LU, Y.; SHANMUGASUNDARAM, J.; SHEKITA, E. J.; SUBRAMANIAN, S. N. XPERANTO: publishing object-relational data as xml. In: INTERNATIONAL WORKSHOP ON THE WEB AND DATABASES, WEBDB, 2000, Dallas, Texas. **Proceedings...** [S.l.: s.n.], 2000. p.105–110.

CAREY, M. J.; KIERNAN, J.; SHANMUGASUNDARAM, J.; SHEKITA, E. J.; SUBRAMANIAN, S. N. XPERANTO: middleware for publishing object-relational data as XML documents. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 2000, Cairo, Egypt. **Proceedings...** San Francisco: Morgan Kaufmann, 2000. p.646–648.

CHAMBERLIN, D.; FANKHAUSER, P.; FLORESCU, D.; MARCHIORI, M.; ROBIE, J. **XML Query Use Cases**. 2003. Available at: <http://www.w3.org/TR/2003/WD-xquery-use-cases-20031112/>. Visited on Oct. 10, 2003. W3C Working Draft.

CHAUDHURI, S.; KAUSHIK, R.; NAUGHTON, J. On Relational Support for XML Publishing: beyond sorting and tagging. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD, 2003, San Diego, CA. **Proceedings...** [S.l.: s.n.], 2003.

CHEN, Y.; DAVIDSON, S. B.; ZHENG, Y. Constrain Preserving XML storage in Relations. In: INTERNATIONAL WORKSHOP ON THE WEB AND DATABASES, WEBDB, 2002, Madison, Wisconsin. **Proceedings...** [S.l.: s.n.], 2002. p.7–12.

CHEN, Y.; DAVIDSON, S. B.; ZHENG, Y. RRXS: redundancy reducing XML storage in relations. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 2003, Berlin, Germany. **Proceedings...** San Francisco: Morgan Kaufmann, 2003.

CHENG, J.; XU, J. XML and DB2. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 2000, San Diego, CA. **Proceedings...** [S.l.: s.n.], 2000.

CLARK, J.; DEROSE, S. **XML Path Language (XPath) Version 1.0**. Available at: <http://www.w3.org/TR/xpath>. Visited on Oct. 10, 2003. W3C Recomendation.

CONRAD, A. **A Survey of Microsoft SQL Server 2000 XML Features**. MSDN Library. July 2001. Available at: <http://msdn.microsoft.com/library/en-us/dnexxml/html/xml07162001.asp>. Visited on Mar. 20, 2004.

CONRAD, A. **Interactive Microsoft SQL Server & XML Online Tutorial**. 2001. Available at: <http://www.topxml.com/tutorials/main.asp?id=sqlxml>. Visited on Dec. 10, 2002.

COSMADAKIS, S. S.; PAPADIMITRIOU, C. H. Updates of Relational Views. **Journal of the Association for Computing Machinery**, [S.l.], v.31, n.4, p.742–760, Oct. 1984.

DATE, C. J. **An Introduction to Database Systems**. 7th ed. [S.l.]: Addison Wesley, 2000.

DAYAL, U.; BERNSTEIN, P. A. On the updatability of Relational Views. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 1978, West Berlin, Germany. **Proceedings...** [S.l.]: IEEE Computer Society, 1978. p.368–377.

DAYAL, U.; BERNSTEIN, P. A. On the correct translation of update operations on relational views. **ACM Transactions on Database Systems, TODS**, [S.l.], v.8, n.2, p.381–416, Sept. 1982.

DAYAL, U.; BERNSTEIN, P. A. On the updatability of network views - extending relational view theory to the network model. **Information Systems**, [S.l.], v.7, n.2, p.29–46, 1982.

DEHAAN, D.; TOMAN, D.; CONSENS, M.; OZSU, M. T. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD, 2003, San Diego, CA. **Proceedings...** [S.l.: s.n.], 2003.

DESCHLER, K.; RUNDENSTEINER, E. MASS: a multi-axis storage structure for large XML documents. In: CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT, CIKM, 2003, New Orleans, Louisiana. **Proceedings...** [S.l.: s.n.], 2003.

DEUTSCH, A.; FERNÁNDEZ, M.; DANIELA FLORESCU, A. L.; SUCIU, D. A Query Language for XML. In: INTERNATIONAL WORLD WIDE WEB CONFERENCE, WWW, 1999, Toronto. **Proceedings...** [S.l.: s.n.], 1999.

DEUTSCH, A.; FERNANDEZ, M.; SUCIU, D. Storing semistructured data with STORED. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD, 1999, Philadelphia, Pennsylvania. **Proceedings...** [S.l.: s.n.], 1999. p.431–442.

DEUTSCH, A.; PAPAKONSTANTINOU, Y.; XU, Y. Minimization and Group-By Detection for Nested XQueries. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 2004, Boston, USA. **Proceedings...** [S.l.: s.n.], 2004. p.839.

DEUTSCH, A.; PAPAKONSTANTINOU, Y.; XU, Y. The NEXT Framework for Logical XQuery Optimization. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 2004, Toronto, Canada. **Proceedings...** San Francisco: Morgan Kaufmann, 2004. p.168–179.

DRAPER, D.; FANKHAUSER, P.; FERNáNDEZ, M.; MALHOTRA, A.; ROSE, K.; RYS, M.; SIMÉON, J.; WADLER, P. **XQuery 1.0 and XPath 2.0 Formal Semantics**. 2004. Available at <`http://www.w3.org/TR/2003/WD-xquery-semantics-20040220/`>. Visited on Aug. 23, 2004. W3C Working Draft.

EISENBERG, A.; MELTON, J. SQL/XML is Making Good Progress. **SIGMOD Record**, [S.l.], v.31, n.2, 2002.

EMBLEY, D. W.; MOK, W. Y. Developing XML Documents with Guaranteed “Good” Properties. In: INTERNATIONAL CONFERENCE ON CONCEPTUAL MODELING, ER, 2001, Yokohama, Japan. **Proceedings...** Berlin: Springer, 2001. p.426–441. (Lecture Notes in Computer Science, v.2224).

FEIJÓ, D. V. **Extrator de Esquema de um Banco de Dados. Instituto de Informática**. 2002. Projeto de Diplomação — Instituto de Informática, UFRGS, Porto Alegre, RS, Brasil.

FERNÁNDEZ, M.; KADIYSKA, Y.; SUCIU, D.; MORISHIMA, A.; TAN, W.-C. SilkRoute: a framework for publishing relational data in XML. **ACM Transactions on Database Systems, TODS**, [S.l.], v.27, n.4, p.438–493, Dec. 2002.

FERNÁNDEZ, M.; MORISHIMA, A.; SUCIU, D. Efficient Evaluation of XML Middle-ware Queries. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD, 2001, Santa Barbara, California. **Proceedings...** [S.l.: s.n.], 2001. p.103–114.

FERNÁNDEZ, M.; MORISHIMA, A.; SUCIU, D.; TAN, W. Publishing Relational Data in XML: the SilkRoute approach. **IEEE Data Engineering Bulletin**, [S.l.], v.24, n.2, p.12–19, 2001.

FERNÁNDEZ, M.; TAN, W.-C.; SUCIU, D. SilkRoute: Trading between Relations and XML. In: INTERNATIONAL WORLD WIDE WEB CONFERENCE, WWW, 2000, Amsterdam. **Proceedings...** [S.l.: s.n.], 2000.

FLORESCU, D.; KOSSMANN, D. **A performance evaluation of alternative mapping schemes for storing XML data in a relational database**. France: INRIA, 1999. (Technical Report 3684).

FURTADO, A. L.; CASANOVA, M. A. Updating Relational Views. In: KIM, W.; REINER, D. S.; BATORY, D. S. (Ed.). **Query Processing in Database Systems**. Berlin, Heidelberg: Springer, 1985. p.127–142.

FURTADO, A. L.; SEVCIK, K. C.; SANTOS, C. S. d. Permitting updates through views of data bases. **Information Systems**, [S.l.], v.4, n.4, p.269–283, Oct. 1979.

HAAS, L.; FREYTAG, J.; LOHMAN, G.; PIRAHESH, H. Extensible Query Processing in Starburst. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD, 1989, Portland. **Proceedings. . .** [S.l.: s.n.], 1989. p.377–388.

IEEE. **IEEE Std 1042**: IEEE Guide to Software Configuration Management. [S.l.], 1987.

IEEE. **IEEE Std 828**: IEEE Standard for Software Configuration Management Plans. [S.l.], 1998.

INTELLIGENT SYSTEM RESEARCH. **ODBC2XML**: merging ODBC data into xml documents. 2001. Available at: <http://www.intsysr.com/odbc2xml.htm>. Visited on Dec. 15, 2002.

ISO. **ISO 10007**: Quality Management - Guidelines for Configuration Management. [S.l.], 1995.

ISO. **ISO/IEC 9075-1**: SQL – Part 1: framework (SQL/Framework). [S.l.], 2003.

ISO. **ISO/IEC 9075-2**: SQL – Part 2: foundation (SQL/Foundation). [S.l.], 2003.

ISO. **ISO/IEC 9075-11**: SQL – Part 11: information and definition schemas (SQL/Schemata). [S.l.], 2003.

ISO. **ISO/IEC 9075-14**: SQL – Part 14: XML-related specifications (SQL/XML). [S.l.], 2003.

ISO. **International Organization for Standardization**. Available at: <http://www.iso.ch/>. Visited on Aug. 17, 2004.

JAESCHKE, G.; SCHEK, H.-J. Remarks on the algebra of non first normal form relations. In: INTERNATIONAL CONFERENCE ON PRINCIPLES OF DATABASE SYSTEMS, PODS, 1982, Los Angeles, CA. **Proceedings. . .** [S.l.: s.n.], 1982. p.124–138.

JAGADISH, H. V.; AL-KHALIFA, S.; CHAPMAN, A.; LAKSHMANAN, L. V.; NIERMAN, A.; PAPARIZOS, S.; PATEL, J. M.; SRIVASTAVA, D.; WIWATWATTANA, N.; WU, Y.; YU, C. TIMBER: a native XML database. **The VLDB Journal**, [S.l.], v.11, n.4, p.274–291, 2002.

JIANG, H.; LU, H.; WANG, W.; OOI, B. C. XR-Tree: indexing XML data for efficient structural joins. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 2003, Bangalore, India. **Proceedings. . .** [S.l.: s.n.], 2003.

KAY, M. **Saxon XSLT and XQuery Processor**. 2001. Available at: <http://sourceforge.net/projects/saxon>. Visited on July 13, 2003.

KELLER, A. M. Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins. In: INTERNATIONAL CONFERENCE ON PRINCIPLES OF DATABASE SYSTEMS, PODS, 1985, Portland, Oregon. **Proceedings...** New York: ACM, 1985. p.154–163.

KELLER, A. M. Comments on Bancilhon and Spyratos' "Update Semantics and Relational Views". **ACM Transactions on Database Systems, TODS**, [S.l.], v.12, n.3, p.521–523, Sept. 1987.

KELLER, A. M.; WIEDERHOLD, G. Penguin: objects for programs, relations for persistence. In: CHAUDHRI, A. B.; ZICARI, R. (Ed.). **Succeeding with Object Databases**. [S.l.]: John Wiley & Sons, 2001.

KELLER, M. The role of semantics in translating view updates. **IEEE Computer**, [S.l.], v.19, n.1, p.63–73, 1986.

KROTH, E. **Tobacco Company**. Personal communication, June 2003.

LANGERAK, R. View updates in relational databases with an independent scheme. **ACM Transactions on Database Systems, TODS**, [S.l.], v.15, n.1, p.40–66, 1990.

LAUX, A.; MARTIN, L. **XUpdate WD**. Sept. 2000. Available at: <http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html>. Visited on Aug. 18, 2004. XML:DB Working Draft.

LECHTENBÖRGER, J. The Impact of the Constant Complement Approach Towards View Updating. In: INTERNATIONAL CONFERENCE ON PRINCIPLES OF DATABASE SYSTEMS, PODS, 2003, San Diego, CA. **Proceedings...** [S.l.: s.n.], 2003. p.49–55.

LEE, D.; CHU, W. W. Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. In: INTERNATIONAL CONFERENCE ON ENTITY RELATIONSHIP, ER, 2000, Salt Lake City, Utah, USA. **Proceedings...** [S.l.: s.n.], 2000. p.323–338.

LEY, M. **DBLP in XML**. 2003. Available at: <http://dblp.uni-trier.de/xml/>. Visited on Oct. 10, 2003.

MALHOTRA, A.; MELTON, J.; ROBIE, J.; RYS, M. **XML Syntax for XQuery 1.0 (XQueryX)**. 2003. Available at: <http://www.w3.org/TR/2003/WD-xqueryx-20031219>. Visited on June 15, 2004. W3C Working Draft.

MALHOTRA, A.; MELTON, J.; WALSH, N. **XQuery 1.0 and XPath 2.0 Functions and Operators**. July, 2004. Available at: <http://www.w3.org/TR/2004/WD-xpath-functions-20040723/>. Visited on Aug. 10, 2004. W3C Working Draft.

MANOLESCU, I.; FLORESCU, D.; KOSSMANN, D. **Pushing XML Queries inside Relational Databases**. France: INRIA, 2001. (Technical Report 4112).

MAY, W. **Information Extraction and Integration with Florid**: the Mondial case study. [S.l.]: Universität Freiburg, Institut für Informatik, 1999. Available at: <http://dbis.informatik.uni-goettingen.de/Mondial/>. Visited on Sept. 15, 2003. (Technical Report 131).

MEDEIROS, C.; TOMPA, F. Undestanding the Implications of View Update Policies. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 1985, Stockholm, Sweden. **Proceedings...** San Francisco: Morgan Kaufmann, 1985. p.316–323.

MELLO, R.; DORNELES, C.; KADE, A.; BRAGANHOLO, V.; HEUSER, C. A. Dados Semi-Estruturados. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, SBBD, 2000, João Pessoa, PB, Brasil. **Anais...** [S.l.: s.n.], 2000.

MERGEN, S.; KELLER, M. F.; KROTH, E. AXIS: a data exchange architecture among heterogeneous data sources using XML documents. In: SIMPÓSIO ARGENTINO DE INGENIERÍA DE SOFTWARE, ASSE, 2002, Santa Fé, Argentina. **Anales...** [S.l.: s.n.], 2002.

MYSQL AB. **MySQL – The World's Most Popular Open Source Database**. 1995. Available at: <http://www.mysql.com/>. Visited on Jun. 10, 2004.

NONNENMACHER, M. J. **Uma Extensão para a API DOM para suportar atualização de bancos de dados relacionais através de visões XML**. 2003. Projeto de Diplomação — Instituto de Informática, UFRGS, Porto Alegre, RS, Brasil.

OCLC Online Computer Library Center. **Dewey Decimal Classification**. Available at: <http://www.oclc.org/dewey/>. Visited on Aug. 17, 2004.

ORACLE CORPORATION. **Oracle 9i**. 2002. Available at: <http://www.oracle.com/database/>. Visited on Sept. 18, 2003.

PAPAKONSTANTINOU, Y.; VIANU, V. Incremental Validation of XML Documents. In: INTERNATIONAL CONFERENCE ON DATABASE THEORY, ICDT, 2003, Siena, Italy. **Proceedings...** [S.l.: s.n.], 2003.

POSTGRESQL. 1995. Available at: <http://www.postgresql.org/>. Visited on Aug. 14, 2004.

ROWE, L. A.; SHOENS, K. A. Data abstraction, views and updates in RIGEL. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD, 1979, Boston, Massachusetts. **Proceedings...** [S.l.: s.n.], 1979. p.71–81.

SALEM, K.; BEYER, K. S.; COCHRANE, R.; LINDSAY, B. G. How To Roll a Join: asynchronous incremental view maintenance. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD, 2000, Dallas, Texas. **Proceedings...** [S.l.: s.n.], 2000. p.129–140.

SCHÖNING, H. Tamino a DBMS designed for XML. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 2001, Heidelberg, Germany. **Proceedings...** [S.l.: s.n.], 2001. p.149–154.

SHANMUGASUNDARAM, J.; KIERNAN, J.; SHEKITA, E.; FAN, C.; FUNDERBURK, J. Querying XML views of relational data. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 2001, Roma, Italy. **Proceedings...** San Francisco: Morgan Kaufmann, 2001.

SHANMUGASUNDARAM, J.; SHEKITA, E. J.; BARR, R.; CAREY, M. J.; LINDSAY, B. G.; PIRAHESH, H.; REINWALD, B. Efficiently Publishing Relational Data as XML Documents. **The VLDB Journal**, [S.l.], p.65–76, 2000.

SHANMUGASUNDARAM, J.; SHEKITA, E.; KIERNAN, J.; KRISHNAMURTHY, R.; VIGLAS, E.; NAUGHTON, J.; TATARINOV, I. A general technique for querying XML documents using a relational database system. **Sigmod Record**, [S.l.], v.30, n.3, p.20–26, Sept. 2001.

SHANMUGASUNDARAM, J.; TUFTE, K.; ZHANG, C.; HE, G.; DEWITT, D. J.; NAUGHTON, J. F. Relational Databases for Querying XML Documents: limitations and opportunities. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 1999, Edinburgh, Scotland, UK. **Proceedings...** San Francisco: Morgan Kaufmann, 1999. p.302–314.

SHU, H. Using Constraint Satisfaction for View Update Translation. In: EUROPEAN CONFERENCE ON ARTIFICIAL INTELLIGENCE, ECAI, 1998, Brighton, UK. **Proceedings...** [S.l.: s.n.], 1998.

SIGMOD Record. Nov., 2002. Available at: <`http://www.acm.org/sigs/sigmod/record/xml/`>. Visited on Setp. 15, 2003.

SOFTWARE AG. **Tamino XML Server**. 2002. Available at: <`http://www.softwareag.com/tamino/details.htm`>. Visited on Sept. 10, 2003.

SUN MICROSYSTEMS. **Java Technology**. 1994. Available at: <`http://java.sun.com/`>. Visited on June 15, 2003.

SUN MICROSYSTEMS. **Java Compiler Compiler**: The Java Parser Generator. 2001. Available at: <`http://javacc.dev.java.net/`>. Visited on Apr. 25, 2003.

TAKAHASHI, T.; KELLER, A. M. Implementation of Object View Query on a Relational Database. In: DATA AND KNOWLEDGE SYSTEMS FOR MANUFACTURING AND ENGINEERING, DKSME, 1994, Hong Kong. **Proceedings...** [S.l.: s.n.], 1994.

TATARINOV, I.; IVES, Z.; HALEVY, A.; WELD, D. Updating XML. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD, 2001, Santa Barbara, CA. **Proceedings...** [S.l.: s.n.], 2001.

TATARINOV, I.; VIGLAS, E.; BEYER, K.; SHANMUGASUNDARAM, J.; SHEKITA, E. Storing and Querying Ordered XML Using a Relational Database System. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD, 2002, Madison, Wisconsin. **Proceedings...** [S.l.: s.n.], 2002.

THOMAS, S. J.; FISCHER, P. C. Nested Relational Structures. **Advances in Computing Research**, [S.l.], v.3, p.269–307, 1986.

TUCHERMAN, L.; FURTADO, A. L.; CASANOVA, M. A. A Pragmatic Approach to Structured Database Design. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 1983, Florence, Italy. **Proceedings...** San Francisco: Morgan Kaufmann, 1983. p.219–231.

TURAU, V. **DB2XML 1.4**: Transforming relational databases into XML documents. Oct., 2001. Available at: <`http://www.informatik.fh-wiesbaden.de/~turau/DB2XML/index.html`>. Visited on Apr. 9, 2004.

ULLMAN, J. D.; WIDOM, J. **A First Course in Database Systems**. [S.l.]: Prentice Hall, 1997.

VICTOLLA, F. **Ferramenta para Automatização do Projeto Lógico de um Banco de Dados e da Implementação do SGBD usando Transformações XSLT**. 2002. Projeto de Diplomação — Instituto de Informática, UFRGS, Porto Alegre, RS, Brasil.

W3C. **World Wide Web Consortium**. Available at: <`http://www.w3.org/`>. Visited on Aug. 25, 2004.

WANG, L.; MULCHANDANI, M.; RUNDENSTEINER, E. A. Updating XQuery Views Published over Relational Data: a round-trip case study. In: XML DATABASE SYMPOSIUM, 2003, Berlin, Germany. **Proceedings...** [S.l.: s.n.], 2003.

WANG, L.; RUNDENSTEINER, E. A. On the updatability of XML Views Published over Relational Data. In: INTERNATIONAL CONFERENCE ON CONCEPTUAL MODELING, ER, 2004, Shanghai, China. **Proceedings...** [S.l.: s.n.], 2004.

WEIS, M. **Development of an Algorithm for Generating an Optimal XML Schema from a given Relational Schema**. 2003. Bachelor Thesis — University of Berufsakademie Stuttgart, Germany.

XBRAIN Project. Available at: <`http://quad.biostr.washington.edu:8080/xbrain/index.jsp`>. Visited on July 13, 2003.

ZHUGE, Y.; GARCÍA-MOLINA, H.; HAMMER, J.; WIDOM, J. View Maintenance in a Warehousing Environment. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD, 1995, San Jose, California. **Proceedings...** [S.l.: s.n.], 1995. p.316–327.

APPENDIX A EXTENDING QUERY TREES TO SUPPORT GROUPING

A.1 Query Trees Redefined to Support Grouping

Figure A.1 shows an example of XML view with group nodes. It is analogous of that of Figure 1.1, with the exception that now *book*s and *dvd*s are clustered by *price* under *products.*

To support grouping, we make the following changes to the definition of query trees:

Definition A.1 (Query Tree Nodes) *Nodes of query trees can be of three types:* ***Leaf nodes*** *have a value (to be defined), which is either* projected *or* grouped. *Names of leaf nodes that start with "@" are considered to be XML attributes.*

Starred nodes *(nodes whose incoming edge is starred) may have one or more source annotations and zero or more where annotations (to be defined). An exception is made for starred nodes with* group *children, which must have no source annotation.*

A ***Group node*** *(one that has a* grouped *value) must have siblings that are starred nodes or group nodes of a restricted form (see definition A.2).*

Definition A.2 (Node Value) *The value of a node n can be* projected *or* grouped. ***A projected value*** *is of form $\$x/A$, where $A \in \mathbb{A}_T$ and $\$x$ is bound to table T by a source annotation on n or some ancestor of n.*

A grouped value *is of form* GROUP($\$x_1/A_1 \mid ... \mid \$x_m/A_m$)*, where $m \geqslant 1$ and $A_i \in \mathbb{A}_{T_i}$ and $\$x_i$ is bound to T_i by a source annotation on a sibling node of n. The domains of $A_1, ...A_m$ in $\mathcal{D}$ must be the same. Group nodes with the same parent must be defined over the same set of variables x_1, ..., x_m, and must have m siblings $b_1, ...b_m$ whose incoming edges are starred*[1]*. Furthermore, the parent of node n must be starred, and it must have no source annotations.*

The intuition behind multiple group nodes with the same parent is to allow tuples to be clustered based on a set of attributes rather than a single attribute.

Additionally, it is necessary to add another starred abstract type to our set of abstract types, so we can distinguish grouped nodes. We call this type τ_G. Nodes of type τ_G are identified as follows: each starred node which has one or more GROUP children has abstract type τ_G.

[1] Notice that we do not require that ($\$x_1/A_1 \mid ... \mid \$x_m/A_m$) in the group operation be in the same order as $b_1, ...b_m$.

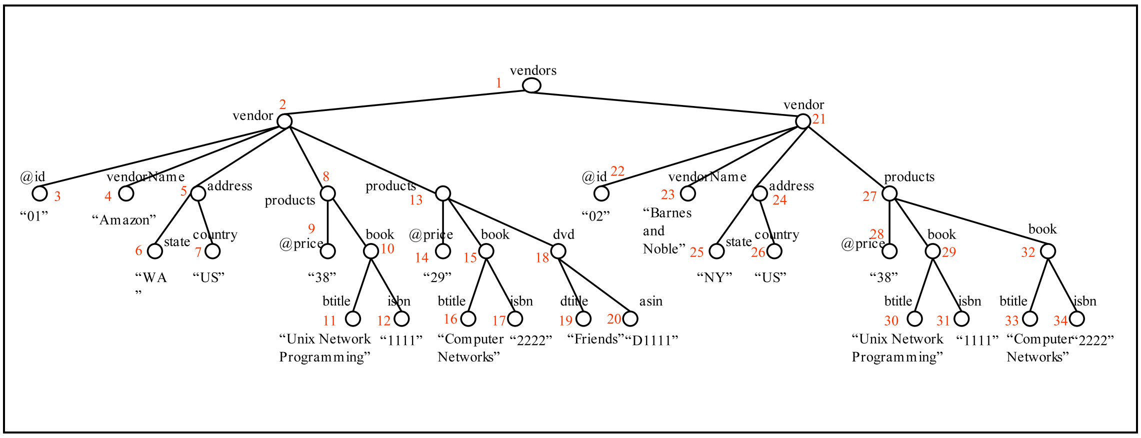


Figure A.1: XML View with books and dvds clustered by price

As an example of query tree which uses group nodes, consider the query tree shown in Figure A.2. It is the query tree that generates the XML view of Figure A.1. Notice that there is a node *price* whose value is grouped: *GROUP($sb/price | $sd/price)*. For this reason, its parent (*products*) is of abstract type τ_G.

A.2 Modifications to the *map* and *split* Algorithms

Given these definitions, we now show how the *eval* and *split* algorithms are modified to support grouping. The new eval algorithm (Algorithm A.1) has to have an additional function called *group*, which deals with the generation of grouped nodes. Notice that this is the only difference between the modified algorithm (Algorithm A.1) and the original *eval* algorithm (Algorithm 4.1).

The new *split* algorithm (Algorithm A.2) needs to take care of group values. It needs to remove parts of the value of group nodes, so that variable references are correct in each split tree. As an example, the query tree of Figure A.1 has a group node *price* whose value is *GROUP($sb/price | $sd/price)*. The two split trees generated by the *split* algorithm will have a node *price* referencing just one of the variables each ($*sb* or $*sd*), as shown in Figures A.3 and A.4.

As for the *map* algorithm, the only change that needs to be made is to add lines 64, 65 and 66, since nodes of type τ_G are starred nodes, but they do not carry any source or where annotation.

A.3 Updatability

Extending query trees with group values reflects in our updatability study. Specifically, we present a rewritten version of theorem 6.1 when group nodes are considered.

Theorem A.1 (Side-effect free XML update redefined) *A correct update u to an XML view defined by a query tree qt is side-effect free if:*

1. *u does not modify the leaf child of a τ_G node, or in other words,* ref *does not point to a* group *node;*
2. *u does not delete a starred child of a τ_G node; and*

```
eval($qt$, $d$)
evaluate(root($qt$, $d$))

evaluate($n$ ,$d$)
Let $bindings\{\}$ be a hash array of bindings of variable attributes to values, initially empty.
case abstract_type($n$)
  $\tau|\tau_C$: buildElement($n$)
  $\tau_T|\tau_N$: table($n$)
  $\tau_G$: group($n$)
  $\tau_S$: print "<name($n$)>value($n$)</name($n$)>"
end case

buildElement($n$)
let $tag$ = "name($n$)"
for each attribute $c$ in children($n$) do
  add "name($c$) = value($c$)"to $tag$
end for
print "< $tag$ >"
for each non-attribute $c$ in children($n$) do
  eval($c$)
end for
print "</name($n$)>"

table($n$)
let $w$ be a list of conditions in sources($n$)
for each $w[i]$ do
  if $w[i]$ involves a variable $v$ in bindings{} then
    substitute the value binding{$v$} for $v$
  end if
end for
calculate the set $B$ of all bindings for variables in sources($n$) that makes the conjunction of the modified $w[i]$'s
true
for each $b$ in $B$ do
  add $b$ to bindings{}
  buildElement($n$)
  remove $b$ from bindings{}
end for

group($n$)
let $g_1$, ... $g_s$ be the GROUP children of $n$
let $w$ be a list of conditions in sources($m$), for all starred nodes $m$ that are children of $n$
for each $w[i]$ do
  if $w[i]$ involves a variable $v$ in bindings{} then
    substitute the value binding{$v$} for $v$
  end if
end for
calculate the set $B$ of all bindings for variables in sources($m$) (for all starred nodes $m$ that are children of $n$) that
makes the conjunction of the modified $w[i]$'s true
let $V_1$=$\bigcup_i$ values of $i$'th group term in $g_1$, taken from $B$
let ...
let $V_s$=$\bigcup_i$ values of $i$'th group term in $g_s$, taken from $B$
for each $v_1$ in $V_1$ do
  add variable bindings $x_i/p$= value($g_1$) for each group variable $x_i$ to bindings{}
  for each $v_s$ in $V_s$ do
    add variable bindings $x_i/p$= value($g_s$) for each group variable $x_i$ to bindings{}
    buildElement($n$)
    remove variable bindings $x_i/p$= value($g_s$) for each group variable $x_i$ in bindings{}
  end for
  remove variable bindings $x_i/p$= value($g_1$) for each group variable $x_i$ in bindings{}
end for
```

Algorithm A.1: Algorithm *eval* modified to support group nodes

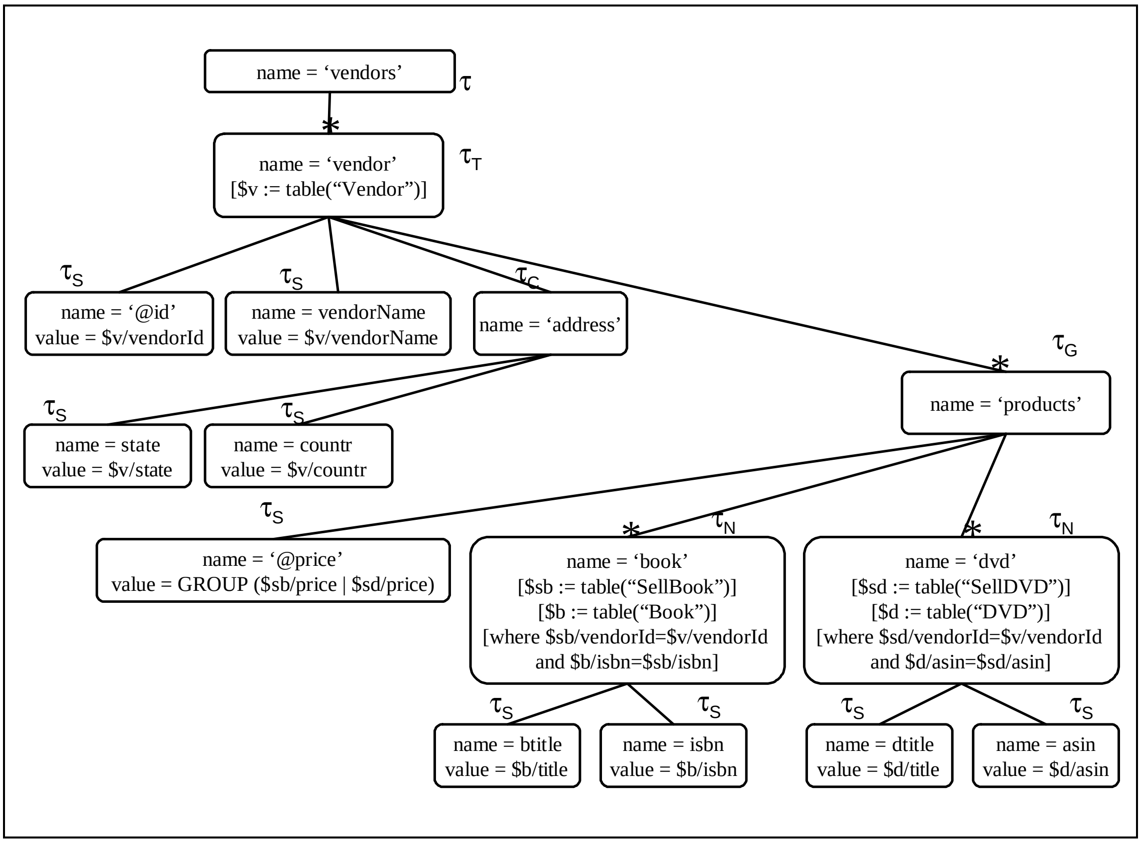


Figure A.2: Query tree with grouped values

3. *For all (U_i, V_i), where V_i is the corresponding relational view of qt_i and U_i is the translation of u over V_i, U_i is side-effect free in V_i.*

Proof: We divide the proof in three steps, one for each condition in the theorem. **Condition 1:** To prove this condition, all we need to do is to show an example of a modification of a group node that causes side-effects. Suppose we specify a modification over the view in Figure A.1 by *ref* = `/vendor/vendor[@id="1"] /products[@price="38"]/@price` and $\Delta = \{29\}$. The evaluation of *ref* yields node 8. Although it seems fine to modify the value of this node, the reconstructed XML view would collapse the subtree rooted at node 13 with the subtree rooted at node 8. This happens because we are changing the value of node 8 to a value that was

```
split($qt$)
Let $t$[] be an array of query trees, initially empty
Let $i = 0$
Let $N$ be the set of nodes of type $\tau_N$ in $qt$
for each node $n$ in $N$ do
  inc $i$
  {initialize $t[i]$ with $qt$}
  $t[i] = qt$
  repeat
    delete from $t[i]$ all subtrees rooted at a node $z$ of type $\tau_N$, where $z \neq n$
    retype the ancestors of the deleted nodes
  until $n$ is the only node of type $\tau_N$ in $t[i]$
  for each group node $g$ in $t[i]$ do
    delete from $g$ all the variable references not declared as source annotations in its starred sibling
  end for
end for
```

Algorithm A.2: Algorithm *split* modified to support group nodes

```
map($qt$[])
Let $sql$[] be an array of strings, initially empty; Let $numberqt$ be the number of split trees in $qt$[]
for $k$ from 1 to $numberqt$ do
  Let $n$ be the node of type $\tau_N$ in $qt[k]$
  $sql[k]$ = "CREATE VIEW "+ name($n$) + "AS "
  $sql[k]$ = $sql[k]$ + "SELECT "
  Let $N$ be the list of leaf nodes in $qt[k]$
  for $i$ from 1 to size($N$) do
    get next $n$ in $N$
    if $i > 1$ then
      $sql[k]$ = $sql[k]$ + ","+ variable($n$) + "."+ attribute($n$) + "AS "+ name($n$)
    else
      $sql[k]$ = $sql[k]$ + variable($n$) + "."+ attribute($n$) + "AS "+ name($n$)
    end if
    $i = i + 1$
  end for
  $sql[k]$ = $sql[k]$ + "FROM "; Let $from$ = ""; Let $N$ be the set of starred nodes in $qt[k]$
  for each $n$ in $N$ do
    Let $join$ = ""; Let $S$ be the list of source annotations in $n$; Let $W$ be the list of where annotations in $n$
    for $i = 1$ to size($S$) do
      get next $s$ in $S$
      $join$ = $join$ + table($s$) + "AS "+ variable($s$)
      if $i <$ size($S$) then
        $join$ = $join$ + "INNER JOIN "
      end if
      $i = i + 1$
    end for
    Let $count = 0$
    for $i = 1$ to size($W$) do
      get next $w$ in $W$
      if $w$ is of the form $\$x/A$ $op$ $\$y/B$ AND $\$x$ is bound to table $X$ by a source annotation $s \in S$ AND $\$y$ is bound to table $Y$ by a source annotation $s' \in S$ then
        if $count = 0$ then
          $join$ = $join$ + "ON "+ $x.A$ $op$ $y.B$
        else
          $join$ = $join$ + "AND "+ $x.A$ $op$ $y.B$
        end if
        $i = i + 1$; $count = count + 1$
      end if
    end for
    if $count = 0$ then
      $join$ = $join$ + "ON (1=1) "
    end if
    if size($S$) $> 1$ then
      $join$ = "("+ $join$ + ")"
    end if
    Let $A$ be the set of starred ancestors of $n$; Let $count = 0$
    if $n$ has a starred ancestor then
      $join$ = "LEFT JOIN "+ $join$
      for $i = 1$ to size($W$) do
        get next $w$ in $W$
        if $w$ is of the form $\$x/A$ $op$ $\$y/B$ AND (($\$x$ is bound to table $X$ on node $n$ AND $\$y$ is bound to table $Y$ on a node $a$ in $A$) OR ($\$x$ is bound to table $X$ on a node $a$ in $A$ AND $\$y$ is bound to table $Y$ on node $n$)) then
          if $count = 0$ then
            $join$ = $join$ + "ON "+ $x.A$ $op$ $y.B$
          else
            $join$ = $join$ + "AND "+ $x.A$ $op$ $y.B$
          end if
        end if
        $i = i + 1$; $count = count + 1$
      end for
      if $count = 0$ then
        $join$ = $join$ + "ON (1=1) "
      end if
      $from$ = "("+ $from$ + $join$ + ")"
    else
      if abstract_type($n$) != $\tau_G$ then
        $from$ = $from$ + $join$
      end if
    end if
  end for
  $sql[k]$ = $sql[k]$ + $from$; Let $W'$ be the set of all where annotations on nodes of $qt[k]$. Let $count = 0$
  for each $w'$ in $W'$ do
    if $w'$ is of the form $\$x/A$ $op$ $Z$ AND $Z$ is an atomic value then
      if $count = 0$ then
        $sql[k]$ = $sql[k]$ + "WHERE "+ $x.A$ $op$ $Z$
      else
        $sql[k]$ = $sql[k]$ + "AND "+ $x.A$ $op$ $Z$
      end if
    end if
  end for
end for
return $sql$[]
```

Algorithm A.3: Algorithm *map* modified to support group nodes

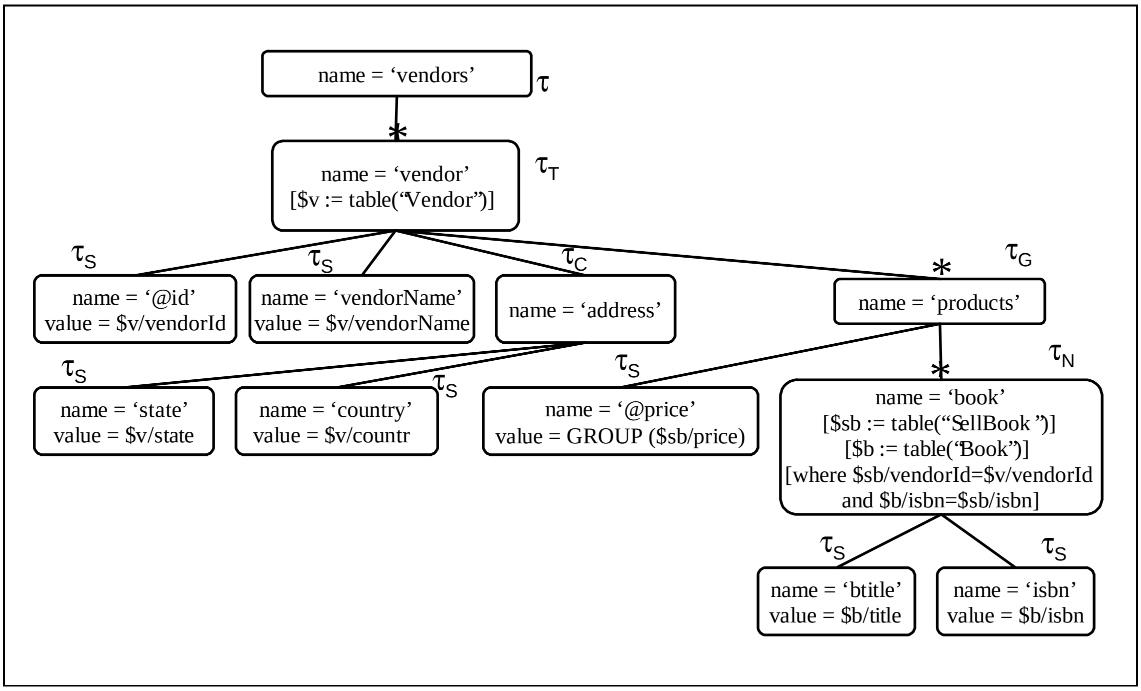


Figure A.3: Partitioned query tree for $\tau_N(book)$

already in the view, and the semantics of GROUP requires that nodes that agree in the value of *price* should be collected together. As a consequence, the XML view modified by the user will be different from the reconstructed view – a side-effect.
Condition 2: This condition can also be proved by a contra example. Consider a deletion over the view in Figure A.1 with update path *ref* = `/vendor/vendor[@id="1"]/products[@price="38"]/book`, which evaluates to node 10. The deletion of this book will also make the subtree rooted at node 8 (*products*) disappear. This is because node 10 was the only book being sold by this price under this vendor. Consequently, the update is not side-effect free.
Condition 3: This condition is the statement of theorem 6.1 itself. Please refer to that theorem for proof. ■

Condition 3 states that if an update passes the “grouping conditions” (conditions 1 and 2), then the update is side-effect free if all updates in its translation onto the underlying relational views are side-effect free. Hence, any side-effect free relational view update technique could be used.

Recall that reasoning about whether or not an update is side-effect free involves the query tree rather than the resulting XML instance. There may therefore be instances for which updates to grouped nodes do *not* cause other updates to be introduced in the recomputed view. Examples include changing the value of node 9 to $20 in Figure A.1 and deleting the subtree rooted at node 18. These (desirable) updates are outlawed in our approach.

There are two ways in which we could allow the desired updates on group nodes above: (1) perform instance analysis to catch exactly the cases that produce side-effects; or (2) allow side-effects in these special cases, or re-define side-effects to exclude empty groups or groups which collapse. We leave this for future work.

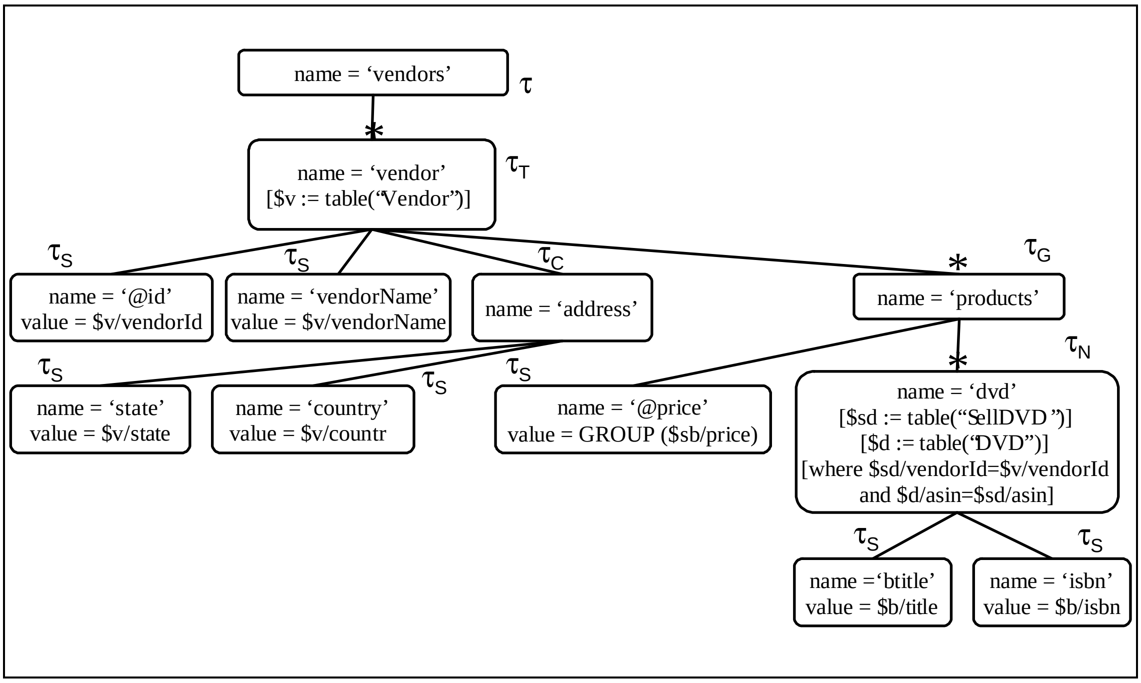


Figure A.4: Partitioned query tree for $\tau_N(dvd)$

APPENDIX B DEALING WITH QUERY TREES WITH REPEATED NODE NAMES

For query trees with repeated node names, the mapping to the relational views may present problems. This is because node names are mapped to attributes of the relational views. Since attributes in a relation schema are a set (ULLMAN; WIDOM, 1997), this implies that two attributes can not have the same name in a relation. To solve this problem, we can use one of the many numbering schemas proposed in literature (TATARINOV et al., 2002; OCLC Online Computer Library Center, 2004; JIANG et al., 2003) to associate unique numbers to nodes. We then concatenate the number with the node name to achieve unique node names for all nodes in the query tree.

In this chapter, we first discuss the numbering schemas we found in the literature, and then illustrate how the *Global Order Encoding* (TATARINOV et al., 2002) could be used in our approach.

B.1 Numbering Schemas

Given the problem we are trying to solve, there are several numbering schemes that can be used. The only requirement is that a *unique* number be associated to each node (that's why they are commonly treated as a *key* that identifies a given node). As examples, we can cite:

- The *Dewey Encoding* (OCLC Online Computer Library Center, 2004) was first applied to XML in (TATARINOV et al., 2002). It associates the number 1 to the root of the tree, and then 1.1, 1.2, ..., 1.n to the direct children of the root. The children of node 1.1 are numbered 1.1.1, ..., 1.1.m, and so on. This numbering schema makes it easier to find hierarchical relationships between nodes (parent/child, ancestor/descendant, siblings).

- The *Global Order Encoding* (TATARINOV et al., 2002) associates a unique number to each node of the tree. These numbers can be generated in a variety of ways. For example, we can start by associating 1 to the root and then increasing the numbers by one as we walk in the tree in a deep-first search order.

- The *Interval Encoding* (JIANG et al., 2003) assigns an interval (*start*, *end*) to each node in the tree. The numbers are assigned in a deep-first search order.

- The *FlexKey* encoding (DESCHLER; RUNDENSTEINER, 2003) is similar to the *Dewey Encoding*, but it uses letters instead of numbers and it leaves gaps in the keys of two consecutive nodes. This encoding is useful when the source document needs to be maintained after several updates, since it avoids renumbering existing nodes on the tree. This is achieved by using strings of variable lengths to identify nodes. As an example, if we need to insert a node between nodes *b.b* and *b.c*, we can refer to the new node as *b.bc*.

Notice that these proposals apply a numbering schema to the XML instance, while here we are proposing to apply such schemes to the query tree. Since the query tree is also a tree, nothing needs to be changed. However, a very simple adaptation needs to be made if the *interval encoding*, the *Dewey encoding* or the *FlexKey* encoding are used. Since these schemes use *comma* or *dot* to separate the numbers, this may cause problems in the translation to SQL. We can replace the *comma* or *dot* by a letter such as x. In the *FlexKey* encoding, an additional care must be taken regarding the separator. Since we are replacing *dot* by the letter x, then the letter x can not be assigned to any part of any key, to avoid confusion.

Figure B.1 shows a query tree with its nodes numbered according to each of the above numbering schemes. Notice that in this example, both nodes *book* and *dvd* have a child named *title*. Note also that the *FlexKey* encoding does not use sequential letters. It leaves gaps to facilitate the inclusion of nodes in the tree. In our case, this feature is not necessary, since we are numbering the query tree instead of the XML instance.

B.2 Applying the Global Order Encoding in Query Trees

Given the numbering schemes shown in the previous section, we can now use one of them to solve the ambiguity of names in the query trees. The mapping works as follows. First, the *Global Order Encoding* is used to associate numbers to each node in the query tree. In order to do so, we introduce an additional annotation in each node, which we call *order annotation*. Second, the extended query tree *qt* is mapped to an *intermediate query tree iqt*. The intermediate query tree *iqt* is exactly the same as *qt*, with the difference that the node names are a concatenation of the corresponding node name in *qt* with the unique number generated by the *Global Order Encoding*. Formally, we have:

Definition B.1 (Numbered Query Tree) *Let a query tree qt be a query tree defined as in Section 4.1.1. Extend qt so that each of its nodes has an additional* order annotation. *A query tree extended in this way is called* numbered query tree *nqt.*

Definition B.2 (Order Annotation) *An* order annotation *of a node n in a query tree qt is of the form [*order=*k], where k is a natural number obtained by traversing the query tree qt in the depth/first order and counting each step until node n is reached.*

We can now define the *intermediate query tree*:

Definition B.3 (Intermediate Query Tree) *An* intermediate query tree *iqt is a new query tree obtained from a* numbered query tree *nqt as follows. Copy nqt*

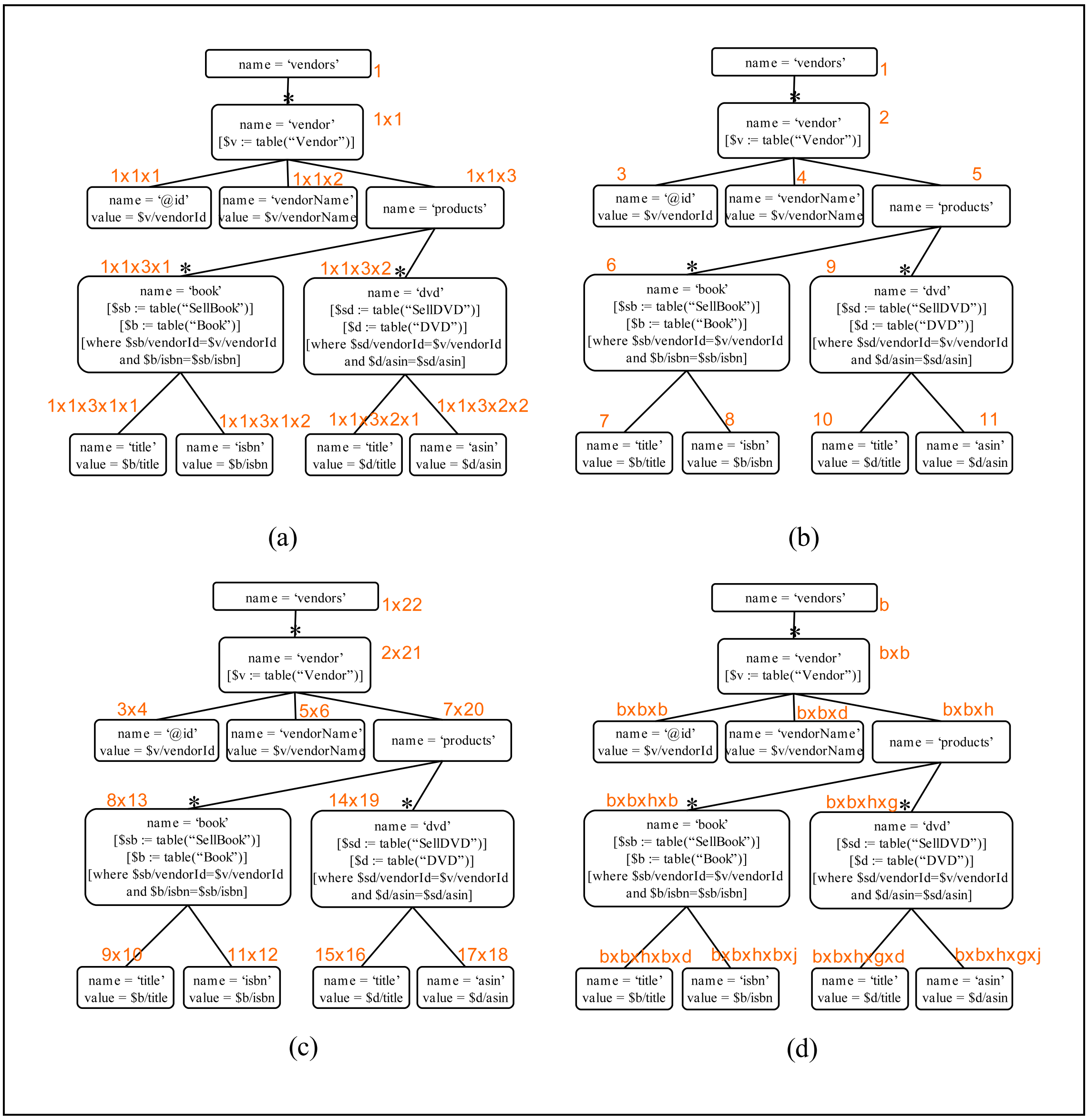

name = 'vendors'
1
*
name = 'vendor' [$v := table("Vendor")]
1x1
1x1x1
1x1x2
1x1x3
name = '@id' value = $v/vendorId
name = 'vendorName' value = $v/vendorName
name = 'products'
1x1x3x1
1x1x3x2
name = 'book' [$sb := table("SellBook")] [$b := table("Book")] [where $sb/vendorId=$v/vendorId and $b/isbn=$sb/isbn]
name = 'dvd' [$sd := table("SellDVD")] [$d := table("DVD")] [where $sd/vendorId=$v/vendorId and $d/asin=$sd/asin]
1x1x3x1x1
1x1x3x1x2
1x1x3x2x1
1x1x3x2x2
name = 'title' value = $b/title
name = 'isbn' value = $b/isbn
name = 'title' value = $d/title
name = 'asin' value = $d/asin
(a)
1
2
3
4
5
6
9
7
8
10
11
(b)
1x22
2x21
3x4
5x6
7x20
8x13
14x19
9x10
11x12
15x16
17x18
(c)
b
bxb
bxbxb
bxbxd
bxbxh
bxbxhxb
bxbxhxg
bxbxhxbxd
bxbxhxbxj
bxbxhxgxd
bxbxhxgxj
(d)

Figure B.1: (a) *Dewey encoding* (b) *Global Order Encoding* (c) *Interval Encoding* (d) *KlexKey encoding*

to iqt. Let n_i be a node in iqt, and n be the corresponding node in nqt. For all n_is in iqt, modify the name of n_i, so that $n_i.name = n.name + n.order$.

An example of *iqt* is shown in Figure B.2. The figure shows the numbered query tree (a) and the corresponding intermediate query tree (b). Notice that the query tree of Figure B.2(a) has two nodes named *id*. In the intermediate query tree of Figure B.2(b), these nodes were mapped to *id3* and *id7*, which solves the problem of unique names.

The intermediate query tree is then used in the mapping to the relational views. Algorithm *map* (Algorithm 5.1) (and also Algorithm *split*, if necessary) can be applied without any modification. According to that algorithm, the relational view for this query tree is:

```
CREATE VIEW VIEWBOOK AS
SELECT v.vendorId AS id3, v.vendorName AS vendorName4,
 b.isbn AS id7, b.title AS title8
FROM (Vendor AS v LEFT JOIN (SellBook AS sb INNER JOIN
Book AS B ON b.isbn=sb.isbn) ON v.vendorId=sb.vendorId);
```

The DTD of the resulting XML view has also to be numbered. This is because an element in a DTD can not be defined more than once (BRAY et al., 2004), and consequently can not have different definitions. When this happens, the parser usually takes the first definition and disregards the remaining ones. However, since this modified DTD is used internally, the user is not aware of the intermediate numbering schema:

```
<!ELEMENT vendors1 (vendor2*)>
<!ELEMENT vendor2 (id3, vendorName4)>
<!ELEMENT id3 (#PCDATA)>
<!ELEMENT vendorName4 (#PCDATA)>
<!ELEMENT products5 (book6*)>
<!ELEMENT book6 (id7, title8)>
<!ELEMENT id7 (#PCDATA)>
<!ELEMENT title8 (#PCDATA)>
```

Notice that without the numbering schema, the DTD would be incorrect, since element *id* would be declared twice. This case is not so problematic, since both declarations would be equal (<!ELEMENT id (#PCDATA)>). However, this is not always the case.

To check for schema conformance, the unqualified portion of the update path is checked against the query tree. The target nodes are identified and the update path is corrected with the order annotation found in the numbered query tree. In case of insertions, both the target nodes and the subtree being inserted are corrected, and the schema checking is done considering the modified element names.

Now, it is necessary to define how the updates on the (unnumbered) XML view are mapped to its corresponding (numbered) relational views. For this, the numbered query tree (Figure B.2(a)) is used. To a better understanding, consider the following example.

Suppose the user wants to modify the book *id* by supplying the update path `/vendors/vendor[id="01"]/products/book[id="1111"]/id` and $\Delta = \{"1245"\}$. To translate this to the relational view, we find the nodes in the numbered query tree, and concatenate each node name with its order annotation. The resulting update expression is as follows:

```
UPDATE VIEWBOOK
SET id7 = "1245"
WHERE id3 = "01" AND id7 = "1111"
```

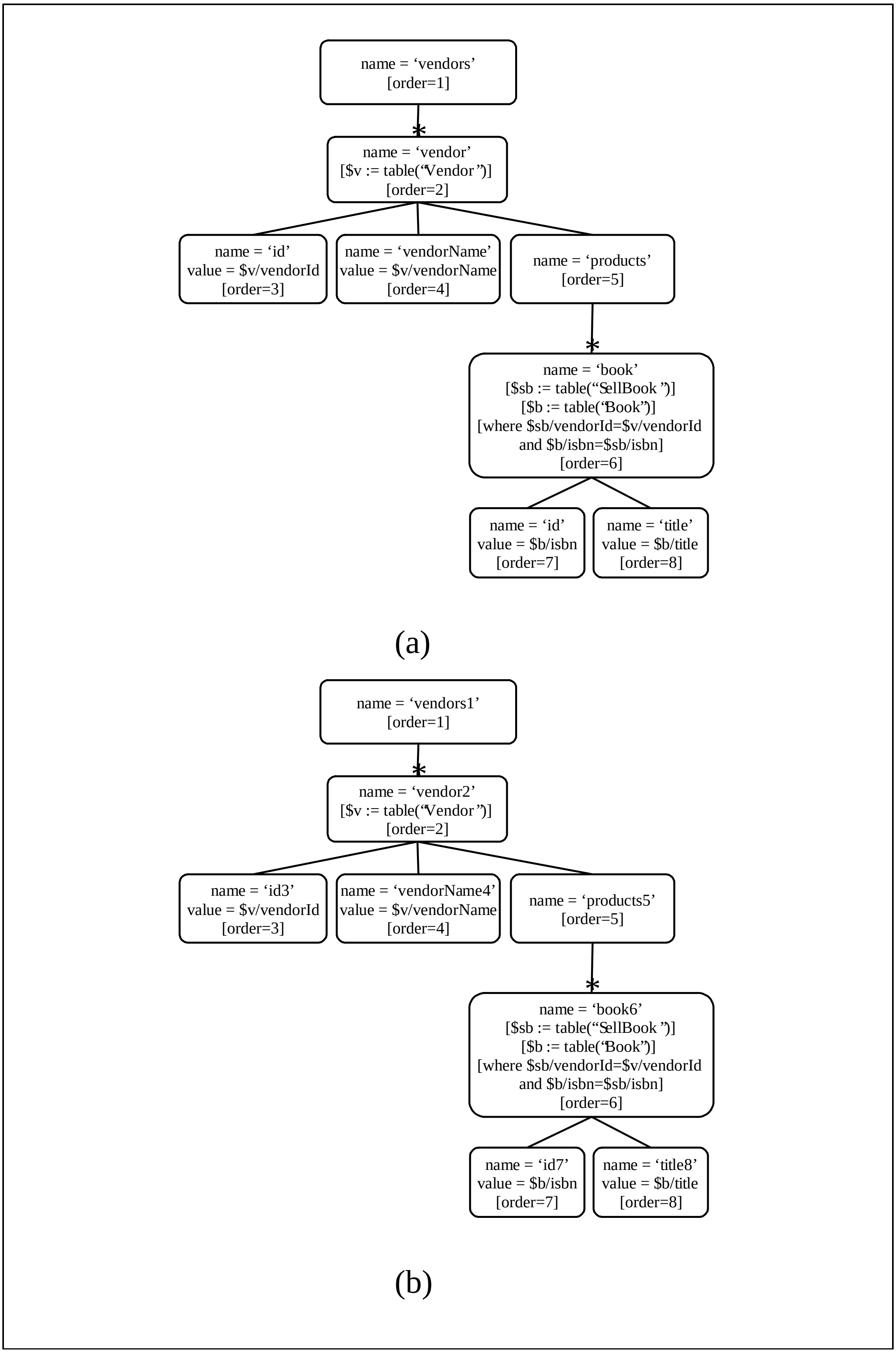


Figure B.2: (a) Numbered query tree (b) Intermediate query tree corresponding to the query tree in (a)

APPENDIX C REAL XML VIEWS

C.1 The Tobacco Company

The scenario of this application is as follows. The company needs to send monthly reports to a Tobacco Producer's Association. The report shows data about the producers from whom the company bought tobacco, as well as prices, quantities, etc. The company stores all the transactions in a relational database, and at the end of the month it generates an XML report containing all the information solicited by the Producer's Association.

For copy right reasons, we omit the company name and the Producers Association Name. We also omit the producer name and SSN.

The XML view generated by this process is as follows:

```
<tobacco>
  <header>
    <companyCode>51</companyCode>
    <companyName>xxxxxx</companyName>
    <reportDate>20030507</reportDate>
    <operationType>15</operationType>
    <crop>02</crop>
    <lotNumber>300</lotNumber>
  </header>
  <summary>
    <total>19</total>
    <totalKgB01>2448.4562</totalKgB01>
    <totalValue>7957.49</totalValue>
  </summary>
  <details>
    <producer>
      <producerID>25989</producerID>
      <SSN>xxxxx</SSN>
      <name>xxxxx</name>
      <birthDate>18/08/1978</birthDate>
      <transaction>
        <date>20020812</date>
        <quantityB01>3.2923</quantityB01>
        <value>10.7</value>
        <lotNumber>300</lotNumber>
        <paymentOption>2</paymentOption>
        <subscribed>50000</subscribed>
        <tobaccoPlants>48000</tobaccoPlants>
      </transaction>
    </producer>
    <producer>
      <producerID>30449</producerID>
      <SSN>xxxxx</SSN>
      <name>xxxxx</name>
      <birthDate>31/12/1951</birthDate>
      <transaction>
        <date>20020905</date>
        <quantityB01>21.5138</quantityB01>
        <value>69.92</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>66000</subscribed>
      <tobaccoPlants>60000</tobaccoPlants>
    </transaction>
    <transaction>
      <date>20020327</date>
      <quantityBO1>54.6862</quantityBO1>
      <value>177.73</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>66000</subscribed>
      <tobaccoPlants>60000</tobaccoPlants>
    </transaction>
  </producer>
  <producer>
    <producerID>30488</producerID>
    <SSN>xxxxx</SSN>
    <name>xxxxx</name>
    <birthDate>23/08/1954</birthDate>
    <transaction>
      <date>20020830</date>
      <quantityBO1>13.2111</quantityBO1>
      <value>42.94</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>15000</subscribed>
      <tobaccoPlants>13000</tobaccoPlants>
    </transaction>
    <transaction>
      <date>20020328</date>
      <quantityBO1>98.5569</quantityBO1>
      <value>320.31</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>15000</subscribed>
      <tobaccoPlants>13000</tobaccoPlants>
    </transaction>
  </producer>
  <producer>
    <producerID>47816</producerID>
    <SSN>xxxxx</SSN>
    <name>xxxxx</name>
    <birthDate>26/05/1974</birthDate>
    <transaction>
      <date>20020812</date>
      <quantityBO1>115.8</quantityBO1>
      <value>376.35</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>18000</subscribed>
      <tobaccoPlants>15000</tobaccoPlants>
    </transaction>
  </producer>
  <producer>
    <producerID>48745</producerID>
    <SSN>xxxxx</SSN>
    <name>xxxxx</name>
    <birthDate>06/12/1966</birthDate>
    <transaction>
      <date>20020802</date>
      <quantityBO1>189</quantityBO1>
      <value>614.25</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>47000</subscribed>
      <tobaccoPlants>48000</tobaccoPlants>
    </transaction>
  </producer>
  <producer>
    <producerID>48800</producerID>
    <SSN>xxxxx</SSN>
    <name>xxxxx</name>
    <birthDate>08/09/1960</birthDate>
    <transaction>
      <date>20020903</date>
      <quantityB01>149.4</quantityB01>
      <value>485.55</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>15000</subscribed>
      <tobaccoPlants>13000</tobaccoPlants>
    </transaction>
  </producer>
  <producer>
    <producerID>201901</producerID>
    <SSN>xxxxx</SSN>
    <name>xxxxx</name>
    <birthDate>14/12/1976</birthDate>
    <transaction>
      <date>20020815</date>
      <quantityB01>99</quantityB01>
      <value>321.75</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>18000</subscribed>
      <tobaccoPlants>20000</tobaccoPlants>
    </transaction>
  </producer>
  <producer>
    <producerID>206130</producerID>
    <SSN>xxxxx</SSN>
    <name>xxxxx</name>
    <birthDate>21/02/1982</birthDate>
    <transaction>
      <date>20020815</date>
      <quantityB01>39.6</quantityB01>
      <value>128.7</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>66000</subscribed>
      <tobaccoPlants>60000</tobaccoPlants>
    </transaction>
  </producer>
  <producer>
    <producerID>206800</producerID>
    <SSN>xxxxx</SSN>
    <name>xxxxx</name>
    <birthDate>20/08/1965</birthDate>
    <transaction>
      <date>20021010</date>
      <quantityB01>157.2</quantityB01>
      <value>510.9</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>38000</subscribed>
      <tobaccoPlants>35000</tobaccoPlants>
    </transaction>
  </producer>
  <producer>
    <producerID>230987</producerID>
    <SSN>xxxxx</SSN>
    <name>xxxxx</name>
    <birthDate>03/04/1978</birthDate>
    <transaction>
      <date>20021004</date>
      <quantityB01>225.6</quantityB01>
      <value>733.2</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>38000</subscribed>
      <tobaccoPlants>35000</tobaccoPlants>
    </transaction>
  </producer>
  <producer>
    <producerID>232872</producerID>
    <SSN>xxxxx</SSN>
    <name>xxxxx</name>
    <birthDate>06/03/1953</birthDate>
    <transaction>
      <date>20020807</date>
      <quantityBO1>193.32</quantityBO1>
      <value>628.29</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>66000</subscribed>
      <tobaccoPlants>60000</tobaccoPlants>
    </transaction>
    <transaction>
      <date>20020606</date>
      <quantityBO1>175.4</quantityBO1>
      <value>570.05</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>66000</subscribed>
      <tobaccoPlants>60000</tobaccoPlants>
    </transaction>
  </producer>
  <producer>
    <producerID>236385</producerID>
    <SSN>xxxxx</SSN>
    <name>xxxxx</name>
    <birthDate>03/09/1957</birthDate>
    <transaction>
      <date>20020827</date>
      <quantityBO1>129.6</quantityBO1>
      <value>421.2</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>18000</subscribed>
      <tobaccoPlants>20000</tobaccoPlants>
    </transaction>
    <transaction>
      <date>20020610</date>
      <quantityBO1>86.0759</quantityBO1>
      <value>279.75</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>18000</subscribed>
      <tobaccoPlants>20000</tobaccoPlants>
    </transaction>
  </producer>
  <producer>
    <producerID>236959</producerID>
    <SSN>xxxxx</SSN>
    <name>xxxxx</name>
    <birthDate>13/06/1948</birthDate>
    <transaction>
      <date>20020827</date>
      <quantityBO1>189</quantityBO1>
      <value>614.25</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>66000</subscribed>
      <tobaccoPlants>60000</tobaccoPlants>
    </transaction>
  </producer>
  <producer>
    <producerID>37491</producerID>
    <SSN>xxxxx</SSN>
    <name>xxxxx</name>
    <birthDate>29/11/1955</birthDate>
    <transaction>
      <date>20020822</date>
      <quantityBO1>405.6</quantityBO1>
      <value>1318.2</value>
      <lotNumber>300</lotNumber>
      <paymentOption>2</paymentOption>
      <subscribed>66000</subscribed>
```

```
        <tobaccoPlants>60000</tobaccoPlants>
      </transaction>
    </producer>
    <producer>
      <producerID>37541</producerID>
      <SSN>xxxxx</SSN>
      <name>xxxxx</name>
      <birthDate>05/03/1945</birthDate>
      <transaction>
        <date>20020814</date>
        <quantityB01>102.6</quantityB01>
        <value>333.45</value>
        <lotNumber>300</lotNumber>
        <paymentOption>2</paymentOption>
        <subscribed>18000</subscribed>
        <tobaccoPlants>20000</tobaccoPlants>
      </transaction>
    </producer>
  </details>
</tobacco>
```

Currently, the XML document is generated by a program that queries the database in SQL and tags the result outside the relational engine. Since the database is very big, we chose to show here just a relevant portion of it:

```
# table CONTRACT has 52 attributes
CONTRACT (CONTRACT_ID NUMBER(10) NOT NULL,
          CONTRACT_DATE DATE ,
          CONTRACT_NUMBER NUMBER(10) ,
          TOTAL_AMOUNT NUMBER(15),
          PRODUCER_ID NUMBER(10),
          COMPANY_ID NUMBER(10))

#table ACCOUNT_TRANSACTION has 46 attributes
TABLE ACCOUNT_TRANSACTION (
          ACCOUNT_TRANS_ID NUMBER(10) NOT NULL,
          TYPE VARCHAR2(1),
          DESCRIPTION VARCHAR2(120),
          TRANS_ID NUMBER(10),
          DATE CHAR(8))

#table TRANSACTION has 41 attributes
TABLE TRANSACTION (
          TRANS_ID NUMBER(10) NOT NULL,
          TRANS_TYPE_ID NUMBER(10),
          NAME VARCHAR2(60),
          NOTE VARCHAR2(200),
          DATE DATE,
          VALUE NUMBER(15),
          CONTRACT_DETAIL_ID NUMBER(10),
          SITUATION NUMBER(1),
          QUANTITY_KG NUMBER(10))

#table CONTRACT_DETAIL has 46 attributes
TABLE CONTRACT_DETAIL (
          CONTRACT_DETAIL_ID NUMBER(10) NOT NULL,
          CONTRACT_ID NUMBER(10))

#table PERSON has 39 attributes
TABLE PERSON(
          PERSON_ID NUMBER(10) NOT NULL,
          NAME VARCHAR2(66),
          SEX VARCHAR2(1),
          SSN NUMBER(11),
          BIRTH_DATE CHAR(10))

#table TOBACCO_COMPANY has 68 attributes
TABLE TOBACCO_COMPANY (
          COMPANY_ID NUMBER(10) NOT NULL,
          COMPANY_NAME VARCHAR2(30))
```

```
#table PRODUCER has 46 attributes
TABLE PRODUTOR_FUMO (
          PRODUCER_ID NUMBER(10) NOT NULL,
          PRODUCER_TYPE NUMBER(1),
          PERSON_ID NUMBER(10))

#table CROP_PRODUCER has 4 attributes
TABLE CROP_PRODUCER(
          YEAR NUMBER(4) NOT NULL,
          PRODUCER_ID NUMBER(10) NOT NULL,
          QUANTITY_TOBACCO_PLANTS NUMBER(10),
          QUANTITY_TOBACCO_PLANTS_SUBSCRIBED NUMBER(10))

#table OPERATION_TYPE has 9 attributes
TABLE OPERATION_TYPE (
          OPT_ID NUMBER(10) NOT NULL,
          OPERATION_TYPE VARCHAR2(30))

TABLE CROP(
          YEAR NUMBER(4),
          LOT NUMBER(4))
```

We now use UXQuery to show how the Tobacco view can be constructed.

Report_Date and Totals are two relational views that summarizes information that will appear in the XML view. When you see an attribute with "BO1" in it's name, it means the most noble type of tobacco.

```
<tobacco>
  {for $company in table("tobacco_company"),
       $d in table("report_date"),
       $op_type in table("operation_type"),
       $crop in table("crop")
   where $crop/year=2002 and $crop/lot=300
   return
   <header>
     <companyCode>{$company/company_id/text()}</companyCode>
     <companyName>{$company/name/text()}</companyName>
     <reportDate>{$d/date/text()}</reportDate>
     <operationType>{$op_type/opt_id/text()}</operationType>
     <crop>{$crop/year/text()}</crop>
     <lotNumber>{$crop/lot/text()}</lotNumber>
   </header>
     {
     for $totals in table("totals")
     where $totals/crop=2002
     return
     <summary>
        <total>{$top/total_quant/text()}</total>
        <totalKgBO1>{$top/total_kg_bo1/text()}</totalKgBO1>
        <totalValue>{$top/total_value/text()}</totalValue>
     </summary>
     }
   <details>
     {
     for $person in table("person"),
         $producer in table("producer"),
     where $person/person_id=$producer/person_id
     return
     <producer>
         <producerId>{$person/person_id/text()}</producerId>
         <SSN>{$person/ssn/text()}</SSN>
         <name>{$person/name/text()}</name>
         <birthDate>{$person/birth_date/text()}</birthDate>
         {
         for $contract in table("contract"),
             $cdetail in table("contract_detail"),
             $account in table("account_transaction"),
             $transaction in table("transaction"),
             $cropp in table("crop_producer")
         where $account/trans_id=$transaction/trans_id and
               $account/type="C" and
```

```
            $cropp/producer_id=$producer/producer_id and
            $cropp/year=2002 and
            $transaction/note=300 and
            $op_type/opt_id=$transaction/trans_type_id and
            $producer/producer_id=contract/producer_id and
            $contract/contract_id=cdetail/contract_id and
            $cdetail/contract_detail_id=$transaction/contract_detail_id and
            $contract/company_id=$company/company_id
        return
        <transaction>
            <date>{$account/date/text()}</date>
            <quantityB01>{$transaction/quantity_kg/text()}</quantityB01>
            <value>{$transaction/value/text()}</value>
            <lotNumber>{$transaction/note/text()}</lotNumber>
            <paymentOption>{$transaction/situation/text()}</paymentOption>
            <subscribed>{$cropp/quantity_tobacco_plants_subscribed/text()}</subscribed>
            <tobaccoPlants>{$cropp/quantity_tobacco_plants/text()}</tobaccoPlants>
        </transaction>
        }
    </producer>
    }
  </details>
  }
</tobacco>
```

C.2 The XBrain Project

The XBrain project is an application of SilkRoute (FERNÁNDEZ et al., 2002). The application queries a brain mapping database, over which an XML public view is defined. The public view is generated by this public query. The public query exports relational data as XML. Its format uses only `for`, and the arrangement of the returned elements can be expressed in UXQuery.

The database schema is as follows:

```
Patient(*oid,initials,first_name,last_name,location,registered,age,sex,viq,pnum,
        is_public,handedness,wada,size,copy,pre,description,gao_research_num);

Surgery(*oid,patient,surgery_date,surgeon,diagnosis,side,lobe,grid);

CSMStudy(*oid,surgery,function,trial_data,site_data);

File(*oid,label,domain,locator,source,mime_type,submit_date,submitted_by,
      version,context,description);

Photo(*oid,preference,image,csmstudy,image_pathname,image_filename);

StimSite(*oid,site_label,zone,lobe,csmstudy,anatomical_name);

Trial(*oid,trial_num,site_label,trial_time,current,slide,eeg_score,miriam_code,
       confidence,comments,km_score,site_suffix,csmstudy,stimulation_site);

UserPerson(*oid,login,first_name,last_name,email,password,user_group);

SiteToAnatomyMap(*oid,csmstudy,photo,scene,author,map_date,
                  sitetoanatomyfile,rendered_map,sitetoanatomy_pathname,
                  sitetoanatomy_filename,preference,modtime);

SiteToAnatomyMapElement(*oid,sitetoanatomymap,stimsite,site_label,
                         ant_coord,sup_coord,right_coord,x,y,confidence);

Scene(*oid,imaging_study,description,description_file,preference, ismapscene);

ImagingStudy(*oid,patient,image_date,billed,prefix,subject,suffix,
             computed_image_pathname,computed_image_filename,
             computed_coords_pathname,computed_coords_filename,
             lowres_surface_pathname,lowres_surface_filename,aligned_pathname);

MRExam(*oid,imaging_study,exam_num,description,import_date, import_info,location);
```

```
Rendering(*oid,rendering_type,preference,image,scene,image_pathname, image_filename);

SceneComponent(*oid,scene,description,surface_model,volume);

SurfaceModel(*oid,volume,model_instance,format,model_file,model_pathname,model_filename,
             preference);

RadialSliceModelInstance(*oid,volume,model,landmarks_file,instance_file,
                         expansion_factor,instance_pathname,instance_filename,
                         preference,landmarks_pathname,landmarks_filename,
                         derived_from);

RadialSliceModel(*oid,pathname,filename,comment,theta_radials,slices,
                  training_set,model_file,preference);

MRSeries(*oid,mrexam,location,showing,total_images,plane,scan_start,
          scan_end,psd,type,description,fov_x,fov_y,height,width,
          bytes_per_pixel,bits_per_pixel,optical_disk,start_img,stop_img,
          threshold,tissue,first,last,label,thickness,spacing);

MRSlice(*oid,sequence_num,image_file,mrseries);

AlignedVolume(*oid,series,format,volume_file,filename,tissue,patient);
```

The view definition query is fully supported by query trees, and its representation in UXQuery is shown below.

```
<root>
{
  for $patient in table("Patient")
  where $patient/is_public/text() = "1"
  return
  <patient oid="{$patient/oid/text()}">
    <initials>{$patient/initials/text()}</initials>
    <first_name>{$patient/first_name/text()}</first_name>
    <last_name>{$patient/last_name/text()} </last_name>
    <location>{$patient/location/text()}</location>
    <registered>{$patient/registered/text()}</registered>
    <age>{$patient/age/text()}</age>
    <sex>{$patient/sex/text()}</sex>
    <viq>{$patient/viq/text()}</viq>
    <pnum>{$patient/pnum/text()}</pnum>
    <is_public>{$patient/is_public/text()}</is_public>
    <handedness>{$patient/handedness/text()}</handedness>
    <wada>{$patient/wada/text()}</wada>
    <size>{$patient/size/text()}</size>
    <copy>{$patient/copy/text()}</copy>
    <pre>{$patient/pre/text()}</pre>
    <description>{$patient/description/text()}</description>
    <gao_research_num>{$patient/gao_research_num/text()}</gao_research_num>
    {
     for $surgery in table("Surgery")
     where data($surgery/patient) = data($patient/oid)
     return
     <surgery oid="{$surgery/oid/text()}">
       <surgery_date>{$surgery/surgery_date/text()}</surgery_date>
       <surgeon>{$surgery/surgeon/text()}</surgeon>
       <diagnosis>{$surgery/diagnosis/text()}</diagnosis>
       <side>{$surgery/side/text()}</side>
       <lobe>{$surgery/lobe/text()}</lobe>
       <grid>{$surgery/grid/text()}</grid>
       {
        for $csmstudy in table("CSMStudy")
        where data($csmstudy/surgery) = data($surgery/oid)
        return
        <csmstudy oid="{$csmstudy/oid/text()}">
          <function>{$csmstudy/function/text()}</function>
          <trial_data oid="{$csmstudy/trial_data/text()}">
          {
           for $trialfile in table("File")
           where data($trialfile/oid) = data($csmstudy/trial_data)
           return
```

```
 <file oid="{$trialfile/oid/text()}">
   <label>{$trialfile/label/text()}</label>
   <domain>{$trialfile/domain/text()}</domain>
   <locator>{$trialfile/locator/text()}</locator>
   <source>{$trialfile/source/text()}</source>
   <mime_type>{$trialfile/mime_type/text()}</mime_type>
   <submit_date>{$trialfile/submit_date/text()}</submit_date>
   <submitted_by>
   {
    for $trialfilesubmitter in table("UserPerson")
    where data($trialfilesubmitter/oid) = data($trialfile/submitted_by)
    return
    <userperson oid="{$trialfilesubmitter/oid/text()}">
      <login>{$trialfilesubmitter/login/text()}</login>
      <first_name>{$trialfilesubmitter/first_name/text()}</first_name>
      <last_name>{$trialfilesubmitter/last_name/text()}</last_name>
      <email>{$trialfilesubmitter/email/text()}</email>
      <user_group>{$trialfilesubmitter/user_group/text()}</user_group>
    </userperson>
    }
   </submitted_by>
   <version>{$trialfile/version/text()}</version>
   <context>{$trialfile/context/text()}</context>
   <description>{$trialfile/description/text()}</description>
 </file>
 }
</trial_data>
<site_data oid="{$csmstudy/site_data/text()}">
{
 for $sitefile in table("File")
 where data($sitefile/oid) = data($csmstudy/site_data)
 return
   <file oid="{$sitefile/oid/text()}">
     <label>{$sitefile/label/text()}</label>
     <domain>{$sitefile/domain/text()}</domain>
     <locator>{$sitefile/locator/text()}</locator>
     <source>{$sitefile/source/text()}</source>
     <mime_type>{$sitefile/mime_type/text()}</mime_type>
     <submit_date>{$sitefile/submit_date/text()}</submit_date>
     <submitted_by>
     {
      for $sitefilesubmitter in table("UserPerson")
      where data($sitefilesubmitter/oid) = data($sitefile/submitted_by)
      return
      <userperson oid="{$sitefilesubmitter/oid/text()}">
        <login>{$sitefilesubmitter/login/text()}</login>
        <first_name>{$sitefilesubmitter/first_name/text()}</first_name>
        <last_name>{$sitefilesubmitter/last_name/text()}</last_name>
        <email>{$sitefilesubmitter/email/text()}</email>
        <user_group>{$sitefilesubmitter/user_group/text()}</user_group>
      </userperson>
     }
     </submitted_by>
     <version>{$sitefile/version/text()}</version>
     <context>{$sitefile/context/text()}</context>
     <description>{$sitefile/description/text()}</description>
   </file>
}
</site_data>
{
 for $photo in table("Photo")
 where data($photo/csmstudy) = data($csmstudy/oid)
 return
 <photo oid="{$photo/oid/text()}">
    <preference>{$photo/preference/text()}</preference>
    <image oid="{$photo/image/text()}">
    {
     for $imagefile in table("File")
     where data($imagefile/oid) = data($photo/image)
     return
     <file oid="{$imagefile/oid/text()}">
       <label>{$imagefile/label/text()}</label>
       <domain>{$imagefile/domain/text()}</domain>
      <locator>{$imagefile/locator/text()}</locator>
      <source>{$imagefile/source/text()}</source>
      <mime_type>{$imagefile/mime_type/text()}</mime_type>
      <submit_date>{$imagefile/submit_date/text()}</submit_date>
      <submitted_by>
       {
        for $imagefilesubmitter in table("UserPerson")
        where data($imagefilesubmitter/oid) = data($imagefile/submitted_by)
        return
        <userperson oid="{$imagefilesubmitter/oid/text()}">
          <login>{$imagefilesubmitter/login/text()}</login>
          <first_name>{$imagefilesubmitter/first_name/text()}</first_name>
          <last_name>{$imagefilesubmitter/last_name/text()}</last_name>
          <email>{$imagefilesubmitter/email/text()}</email>
          <user_group>{$imagefilesubmitter/user_group/text()}</user_group>
        </userperson>
       }
      </submitted_by>
      <version>{$imagefile/version/text()}</version>
      <context>{$imagefile/context/text()}</context>
      <description>{$imagefile/description/text()}</description>
    </file>
   }
   </image>
   <image_pathname>{$photo/image_pathname/text()}</image_pathname>
   <image_filename>{$photo/image_filename/text()}</image_filename>
</photo>
}
{
 for $trial in table("Trial")
 where data($trial/csmstudy) = data($csmstudy/oid)
 return
 <trial oid="{$trial/oid/text()}">
   <trial_num>{$trial/trial_num/text()}</trial_num>
   <site_label>{$trial/site_label/text()}</site_label>
   <trial_time>{$trial/trial_time/text()}</trial_time>
   <current>{$trial/current/text()}</current>
   <slide>{$trial/slide/text()}</slide>
   <eeg_score>{$trial/eeg_score/text()}</eeg_score>
   <miriam_code>{$trial/miriam_code/text()}</miriam_code>
   <confidence>{$trial/confidence/text()}</confidence>
   <comments>{$trial/comments/text()}</comments>
   <km_score>{$trial/km_score/text()}</km_score>
   <site_suffix>{$trial/site_suffix/text()}</site_suffix>
   {
    for $trialstim in table("StimSite")
    where data($trialstim/oid) = data($trial/stimulation_site)
    return
    <t_stimsite oid="{$trialstim/oid/text()}">
      <t_site_label>{$trialstim/site_label/text()}</t_site_label>
      <t_zone>{$trialstim/zone/text()}</t_zone>
      <t_lobe>{$trialstim/lobe/text()}</t_lobe>
      <t_anatomical_name>{$trialstim/anatomical_name/text()}</t_anatomical_name>
    </t_stimsite>
   }
 </trial>
}
{
 for $stimsite in table("StimSite")
 where data($stimsite/csmstudy) = data($csmstudy/oid)
 return
 <stimsite oid="{$stimsite/oid/text()}">
   <site_label>{$stimsite/site_label/text()}</site_label>
   <zone>{$stimsite/zone/text()}</zone>
   <lobe>{$stimsite/lobe/text()}</lobe>
   <anatomical_name>{$stimsite/anatomical_name/text()}</anatomical_name>
   {
    for $stimtrial in table("Trial")
    where data($stimtrial/stimulation_site) = data($stimsite/oid)
    return
    <s_trial oid="{$stimtrial/oid/text()}">
      <s_trial_num>{$stimtrial/trial_num/text()}</s_trial_num>
      <s_site_label>{$stimtrial/site_label/text()}</s_site_label>
      <s_trial_time>{$stimtrial/trial_time/text()}</s_trial_time>
      <s_current>{$stimtrial/current/text()}</s_current>
      <s_slide>{$stimtrial/slide/text()}</s_slide>
      <s_eeg_score>{$stimtrial/eeg_score/text()}</s_eeg_score>
      <s_miriam_code>{$stimtrial/miriam_code/text()}</s_miriam_code>
      <s_confidence>{$stimtrial/confidence/text()}</s_confidence>
      <s_comments>{$stimtrial/comments/text()}</s_comments>
      <s_km_score>{$stimtrial/km_score/text()}</s_km_score>
      <s_site_suffix>{$stimtrial/site_suffix/text()}</s_site_suffix>
    </s_trial>
   }
 </stimsite>
}
{
 for $sitetoanatomymap in table("SiteToAnatomyMap")
 where data($sitetoanatomymap/csmstudy) = data($csmstudy/oid)
 return
 <sitetoanatomymap oid="{$sitetoanatomymap/oid/text()}">
 {
  for $sitephoto in table("Photo")
  where data($sitetoanatomymap/photo) = data($sitephoto/oid)
  return
  <photo oid="{$sitephoto/oid/text()}">
    <preference>{$sitephoto/preference/text()}</preference>
    <image oid="{$sitephoto/image/text()}">
    {
     for $sitephotoimagefile in $table("File")
     where data($sitephotoimagefile/oid) = data($sitephoto/image)
     return
     <file oid="{$sitephotoimagefile/oid/text()}">
       <label>{$sitephotoimagefile/label/text()}</label>
       <domain>{$sitephotoimagefile/domain/text()}</domain>
       <locator>{$sitephotoimagefile/locator/text()}</locator>
       <source>{$sitephotoimagefile/source/text()}</source>
       <mime_type>{$sitephotoimagefile/mime_type/text()}</mime_type>
       <submit_date>{$sitephotoimagefile/submit_date/text()}</submit_date>
       <submitted_by>
       {
        for $sitephotoimagefilesubmitter in table("UserPerson")
        where data($sitephotoimagefilesubmitter/oid) = data($sitephotoimagefile/submitted_by)
        return
        <userperson oid="{$sitephotoimagefilesubmitter/oid/text()}">
          <login>{$sitephotoimagefilesubmitter/login/text()}</login>
          <first_name>{$sitephotoimagefilesubmitter/first_name/text()}</first_name>
          <last_name>{$sitephotoimagefilesubmitter/last_name/text()}</last_name>
          <email>{$sitephotoimagefilesubmitter/email/text()}</email>
          <user_group>{$sitephotoimagefilesubmitter/user_group/text()}</user_group>
        </userperson>
       }
      </submitted_by>
      <version>{$sitephotoimagefile/version/text()}</version>
      <context>{$sitephotoimagefile/context/text()}</context>
      <description>{$sitephotoimagefile/description/text()}</description>
     </file>
    }
    </image>
    <image_pathname>{$sitephoto/image_pathname/text()}</image_pathname>
    <image_filename>{$sitephoto/image_filename/text()}</image_filename>
  </photo>
 }
 {
 for $sitescene in table("Scene")
 where data($sitetoanatomymap/scene) = data($sitescene/oid)
 return
 <scene oid="{$sitescene/oid/text()}">
   <imaging_study>{$sitescene/imaging_study/text()}</imaging_study>
   <description>{$sitescene/description/text()}</description>
   <description_file oid="{$sitescene/description_file/text()}">
   {
    for $sitescenefile in table("File")
    where data($sitescenefile/oid) = data($sitescene/description_file)
    return
    <file oid="{$sitescenefile/oid/text()}">
   <label>{$sitescenefile/label/text()}</label>
   <domain>{$sitescenefile/domain/text()}</domain>
   <locator>{$sitescenefile/locator/text()}</locator>
   <source>{$sitescenefile/source/text()}</source>
   <mime_type>{$sitescenefile/mime_type/text()}</mime_type>
   <submit_date>{$sitescenefile/submit_date/text()}</submit_date>
   <submitted_by>
   {
    for $sitescenefilesubmitter in table("UserPerson")
    where data($sitescenefilesubmitter/oid) = data($sitescenefile/submitted_by)
    return
    <userperson oid="{$sitescenefilesubmitter/oid/text()}">
      <login>{$sitescenefilesubmitter/login/text()}</login>
      <first_name>{$sitescenefilesubmitter/first_name/text()}</first_name>
      <last_name>{$sitescenefilesubmitter/last_name/text()}</last_name>
      <email>{$sitescenefilesubmitter/email/text()}</email>
      <user_group>{$sitescenefilesubmitter/user_group/text()}</user_group>
    </userperson>
   }
  </submitted_by>
  <version>{$sitescenefile/version/text()}</version>
  <context>{$sitescenefile/context/text()}</context>
  <description>{$sitescenefile/description/text()}</description>
 </file>
}
</description_file>
<preference>{$sitescene/preference/text()}</preference>
<ismapscene>{$sitescene/ismapscene/text()}</ismapscene>
{
 for $rendering in table("Rendering")
 where data($rendering/scene) = data($sitescene/oid)
 return
 <rendering oid="{$rendering/oid/text()}">
   <rendering_type>{$rendering/rendering_type/text()}</rendering_type>
   <preference>{$rendering/preference/text()}</preference>
   <image>
   {
    for $renderingfile in table("File")
    where data($renderingfile/oid) = data($rendering/image)
    return
    <file oid="{$renderingfile/oid/text()}">
      <label>{$renderingfile/label/text()}</label>
      <domain>{$renderingfile/domain/text()}</domain>
      <locator>{$renderingfile/locator/text()}</locator>
      <source>{$renderingfile/source/text()}</source>
      <mime_type>{$renderingfile/mime_type/text()}</mime_type>
      <submit_date>{$renderingfile/submit_date/text()}</submit_date>
      <submitted_by>
      {
       for $renderingfilesubmitter in table("UserPerson")
       where data($renderingfilesubmitter/oid) = data($renderingfile/submitted_by)
       return
       <userperson oid="{$renderingfilesubmitter/oid/text()}">
         <login>{$renderingfilesubmitter/login/text()}</login>
         <first_name>{$renderingfilesubmitter/first_name/text()}</first_name>
         <last_name>{$renderingfilesubmitter/last_name/text()}</last_name>
         <email>{$renderingfilesubmitter/email/text()}</email>
         <user_group>{$renderingfilesubmitter/user_group/text()}</user_group>
       </userperson>
      }
      </submitted_by>
      <version>{$renderingfile/version/text()}</version>
      <context>{$renderingfile/context/text()}</context>
      <description>{$renderingfile/description/text()}</description>
    </file>
   }
  </image>
  <image_pathname>{$rendering/image_pathname/text()}</image_pathname>
  <image_filename>{$rendering/image_filename/text()}</image_filename>
 </rendering>
}
{
 for $scenecomponent in table("SceneComponent")
```

```
where data($scenecomponent/scene) = data($sitescene/oid)
return
<scenecomponent oid="{$scenecomponent/oid/text()}">
  <description>{$scenecomponent/description/text()}</description>
 {
  for $surfacemodel in table("SurfaceModel")
  where data($surfacemodel/oid) = data($scenecomponent/surface_model)
  return
  <surfacemodel oid="{$surfacemodel/oid/text()}">
    <format>{$surfacemodel/format/text()}</format>
    <model_file>
    {
     for $modelfile in table("File")
     where data($modelfile/oid) = data($surfacemodel/model_file)
     return
     <file oid="{$modelfile/oid/text()}">
       <label>{$modelfile/label/text()}</label>
       <domain>{$modelfile/domain/text()}</domain>
       <locator>{$modelfile/locator/text()}</locator>
       <source>{$modelfile/source/text()}</source>
       <mime_type>{$modelfile/mime_type/text()}</mime_type>
       <submit_date>{$modelfile/submit_date/text()}</submit_date>
       <submitted_by>
       {
        for $modelfilesubmitter in table("UserPerson")
        where data($modelfilesubmitter/oid) = data($modelfile/submitted_by)
        return
        <userperson oid="{$modelfilesubmitter/oid/text()}">
          <login>{$modelfilesubmitter/login/text()}</login>
          <first_name>{$modelfilesubmitter/first_name/text()}</first_name>
          <last_name>{$modelfilesubmitter/last_name/text()}</last_name>
          <email>{$modelfilesubmitter/email/text()}</email>
          <user_group>{$modelfilesubmitter/user_group/text()}</user_group>
        </userperson>
       }
       </submitted_by>
       <version>{$modelfile/version/text()}</version>
       <context>{$modelfile/context/text()}</context>
       <description>{$modelfile/description/text()}</description>
     </file>
    }
    </model_file>
    <model_pathname>{$surfacemodel/model_pathname/text()}</model_pathname>
    <model_filename>{$surfacemodel/model_filename/text()}</model_filename>
    <preference>{$surfacemodel/preference/text()}</preference>
    {
     for $radialslicemodelinstance in table("RadialSliceModelInstance")
     where data($radialslicemodelinstance/oid) = data($surfacemodel/model_instance)
     return
     <radialslicemodelinstance oid="{$radialslicemodelinstance/oid/text()}">
     <landmarks_file>
     {
      for $landmarksfile in table("File")
      where data($landmarksfile/oid) = data($radialslicemodelinstance/landmarks_file)
      return
      <file oid="{$landmarksfile/oid/text()}">
        <label>{$landmarksfile/label/text()}</label>
        <domain>{$landmarksfile/domain/text()}</domain>
        <locator>{$landmarksfile/locator/text()}</locator>
        <source>{$landmarksfile/source/text()}</source>
        <mime_type>{$landmarksfile/mime_type/text()}</mime_type>
        <submit_date>{$landmarksfile/submit_date/text()}</submit_date>
        <submitted_by>
        {
         for $landmarksfilesubmitter in table("UserPerson")
         where data($landmarksfilesubmitter/oid) = data($landmarksfile/submitted_by)
         return
         <userperson oid="{$landmarksfilesubmitter/oid/text()}">
            <login>{$landmarksfilesubmitter/login/text()}</login>
            <first_name>{$landmarksfilesubmitter/first_name/text()}</first_name>
            <last_name>{$landmarksfilesubmitter/last_name/text()}</last_name>
            <email>{$landmarksfilesubmitter/email/text()}</email>
            <user_group>{$landmarksfilesubmitter/user_group/text()}</user_group>
    </userperson>
   }
   </submitted_by>
   <version>{$landmarksfile/version/text()}</version>
   <context>{$landmarksfile/context/text()}</context>
   <description>{$landmarksfile/description/text()}</description>
 </file>
}
</landmarks_file>
<instance_file>
{
 for $instancefile in table("File")
 where data($instancefile/oid) = data($radialslicemodelinstance/instance_file)
 return
 <file oid="{$instancefile/oid/text()}">
   <label>{$instancefile/label/text()}</label>
   <domain>{$instancefile/domain/text()}</domain>
   <locator>{$instancefile/locator/text()}</locator>
   <source>{$instancefile/source/text()}</source>
   <mime_type>{$instancefile/mime_type/text()}</mime_type>
   <submit_date>{$instancefile/submit_date/text()}</submit_date>
   <submitted_by>
   {
    for $instancefilesubmitter in table("UserPerson")
    where data($instancefilesubmitter/oid) = data($instancefile/submitted_by)
    return
    <userperson oid="{$instancefilesubmitter/oid/text()}">
      <login>{$instancefilesubmitter/login/text()}</login>
      <first_name>{$instancefilesubmitter/first_name/text()}</first_name>
      <last_name>{$instancefilesubmitter/last_name/text()}</last_name>
      <email>{$instancefilesubmitter/email/text()}</email>
      <user_group>{$instancefilesubmitter/user_group/text()}</user_group>
    </userperson>
   }
   </submitted_by>
   <version>{$instancefile/version/text()}</version>
   <context>{$instancefile/context/text()}</context>
   <description>{$instancefile/description/text()}</description>
 </file>
 }
</instance_file>
<expansion_factor>{$radialslicemodelinstance/expansion_factor/text()}</expansion_factor>
<instance_pathname>{$radialslicemodelinstance/instance_pathname/text()}
   </instance_pathname>
<instance_filename>{$radialslicemodelinstance/instance_filename/text()}
   </instance_filename>
<preference>{$radialslicemodelinstance/preference/text()}</preference>
<landmarks_pathname>{$radialslicemodelinstance/landmarks_pathname/text()}
   </landmarks_pathname>
<landmarks_filename>{$radialslicemodelinstance/landmarks_filename/text()}
   </landmarks_filename>
{
 for $radialslicemodel in table("RadialSliceModel")
 where data($radialslicemodelinstance/model) = data($radialslicemodel/oid)
 return
 <radialslicemodel oid="{$radialslicemodel/oid/text()}">
   <pathname>{$radialslicemodel/pathname/text()}</pathname>
   <filename>{$radialslicemodel/filename/text()}</filename>
   <comment>{$radialslicemodel/comment/text()}</comment>
   <theta_radials>{$radialslicemodel/theta_radials/text()}</theta_radials>
   <slices>{$radialslicemodel/slices/text()}</slices>
   <training_set>{$radialslicemodel/training_set/text()}</training_set>
   <model_file>
   {
    for $modelfile in table("File")
    where data($modelfile/oid) = data($radialslicemodel/model_file)
    return
    <file oid="{$modelfile/oid/text()}">
      <label>{$modelfile/label/text()}</label>
      <domain>{$modelfile/domain/text()}</domain>
      <locator>{$modelfile/locator/text()}</locator>
      <source>{$modelfile/source/text()}</source>
      <mime_type>{$modelfile/mime_type/text()}</mime_type>
```

```
          <submit_date>{$modelfile/submit_date/text()}</submit_date>
          <submitted_by>
          {
           for $modelfilesubmitter in table("UserPerson")
           where data($modelfilesubmitter/oid) = data($modelfile/submitted_by)
           return
           <userperson oid="{$modelfilesubmitter/oid/text()}">
             <login>{$modelfilesubmitter/login/text()}</login>
             <first_name>{$modelfilesubmitter/first_name/text()}</first_name>
             <last_name>{$modelfilesubmitter/last_name/text()}</last_name>
             <email>{$modelfilesubmitter/email/text()}</email>
             <user_group>{$modelfilesubmitter/user_group/text()}</user_group>
           </userperson>
          }
          </submitted_by>
          <version>{$modelfile/version/text()}</version>
          <context>{$modelfile/context/text()}</context>
          <description>{$modelfile/description/text()}</description>
        </file>
       }
       </model_file>
       <preference>{$radialslicemodel/preference/text()}</preference>
     </radialslicemodel>
    }
    </radialslicemodelinstance>
   }
   </surfacemodel>
  }
  </scenecomponent>
 }
 </scene>
}
<author>{$sitetoanatomymap/author/text()}</author>
<map_date>{$sitetoanatomymap/map_date/text()}</map_date>
<sitetoanatomyfile oid="{$sitetoanatomymap/sitetoanatomyfile/text()}">
{
 for $sitetofile in table("File")
 where data($sitetofile/oid) = data($sitetoanatomymap/sitetoanatomyfile)
 return
 <file oid="{$sitetofile/oid/text()}">
   <label>{$sitetofile/label/text()}</label>
   <domain>{$sitetofile/domain/text()}</domain>
   <locator>{$sitetofile/locator/text()}</locator>
   <source>{$sitetofile/source/text()}</source>
   <mime_type>{$sitetofile/mime_type/text()}</mime_type>
   <submit_date>{$sitetofile/submit_date/text()}</submit_date>
   <submitted_by>
    {
    for $sitetofilesubmitter in table("UserPerson")
    where data($sitetofilesubmitter/oid) = data($sitetofile/submitted_by)
    return
    <userperson oid="{$sitetofilesubmitter/oid/text()}">
      <login>{$sitetofilesubmitter/login/text()}</login>
      <first_name>{$sitetofilesubmitter/first_name/text()}</first_name>
      <last_name>{$sitetofilesubmitter/last_name/text()}</last_name>
      <email>{$sitetofilesubmitter/email/text()}</email>
      <user_group>{$sitetofilesubmitter/user_group/text()}</user_group>
    </userperson>
    }
   </submitted_by>
   <version>{$sitetofile/version/text()}</version>
   <context>{$sitetofile/context/text()}</context>
   <description>{$sitetofile/description/text()}</description>
 </file>
}
</sitetoanatomyfile>
<rendered_map oid="{$sitetoanatomymap/rendered_map/text()}">
{
 for $mapfile in table("File")
 where data($mapfile/oid) = data($sitetoanatomymap/rendered_map)
 return
 <file oid="{$mapfile/oid/text()}">
   <label>{$mapfile/label/text()}</label>
```

```
    <domain>{$mapfile/domain/text()}</domain>
    <locator>{$mapfile/locator/text()}</locator>
    <source>{$mapfile/source/text()}</source>
    <mime_type>{$mapfile/mime_type/text()}</mime_type>
    <submit_date>{$mapfile/submit_date/text()}</submit_date>
    <submitted_by>
    {
     for $mapfilesubmitter in table("UserPerson")
     where data($mapfilesubmitter/oid) = data($mapfile/submitted_by)
     return
     <userperson oid="{$mapfilesubmitter/oid/text()}">
       <login>{$mapfilesubmitter/login/text()}</login>
       <first_name>{$mapfilesubmitter/first_name/text()}</first_name>
       <last_name>{$mapfilesubmitter/last_name/text()}</last_name>
       <email>{$mapfilesubmitter/email/text()}</email>
       <user_group>{$mapfilesubmitter/user_group/text()}</user_group>
     </userperson>
    }
    </submitted_by>
    <version>{$mapfile/version/text()}</version>
    <context>{$mapfile/context/text()}</context>
    <description>{$mapfile/description/text()}</description>
  </file>
 }
 </rendered_map>
 <sitetoanatomy_pathname>{$sitetoanatomymap/sitetoanatomy_pathname/text()}
   </sitetoanatomy_pathname>
 <sitetoanatomy_filename>{$sitetoanatomymap/sitetoanatomy_filename/text()}
   </sitetoanatomy_filename>
 <preference>{$sitetoanatomymap/preference/text()}</preference>
 <modtime>{$sitetoanatomymap/modtime/text()}</modtime>
 {
  for $sitetoanatomymapelement in table("SiteToAnatomyMapElement")
  where data($sitetoanatomymapelement/sitetoanatomymap) = data($sitetoanatomymap/oid)
  return
  <sitetoanatomymapelement>
  {
   for $sitestimsite in table("StimSite")
   where data($sitestimsite/oid) = data($sitetoanatomymapelement/stimsite)
   return
   <stimsite oid="{$sitestimsite/oid/text()}">
     <site_label>{$sitestimsite/site_label/text()}</site_label>
     <zone>{$sitestimsite/zone/text()}</zone>
     <lobe>{$sitestimsite/lobe/text()}</lobe>
     <anatomical_name>{$sitestimsite/anatomical_name/text()}</anatomical_name>
     {
      for $sitestimtrial in table("Trial")
      where data($sitestimtrial/stimulation_site) = data($sitestimsite/oid)
      return
      <trial oid="{$sitestimtrial/oid/text()}">
        <trial_num>{$sitestimtrial/trial_num/text()}</trial_num>
        <site_label>{$sitestimtrial/site_label/text()}</site_label>
        <trial_time>{$sitestimtrial/trial_time/text()}</trial_time>
        <current>{$sitestimtrial/current/text()}</current>
        <slide>{$sitestimtrial/slide/text()}</slide>
        <eeg_score>{$sitestimtrial/eeg_score/text()}</eeg_score>
        <miriam_code>{$sitestimtrial/miriam_code/text()}</miriam_code>
        <confidence>{$sitestimtrial/confidence/text()}</confidence>
        <comments>{$sitestimtrial/comments/text()}</comments>
        <km_score>{$sitestimtrial/km_score/text()}</km_score>
        <site_suffix>{$sitestimtrial/site_suffix/text()}</site_suffix>
      </trial>
     }
   </stimsite>
  }
  <site_label>{$sitetoanatomymapelement/site_label/text()}</site_label>
  <ant_coord>{$sitetoanatomymapelement/ant_coord/text()}</ant_coord>
  <sup_coord>{$sitetoanatomymapelement/sup_coord/text()}</sup_coord>
  <right_coord>{$sitetoanatomymapelement/right_coord/text()}</right_coord>
  <x>{$sitetoanatomymapelement/x/text()}</x>
  <y>{$sitetoanatomymapelement/y/text()}</y>
  <confidence>{$sitetoanatomymapelement/confidence/text()}</confidence>
</sitetoanatomymapelement>
```

```
          }
        </sitetoanatomymap>
       }
     </csmstudy>
    }
 </surgery>
}
{
 for $imagingstudy in table("ImagingStudy")
 where data($imagingstudy/patient) = data($patient/oid)
 return
 <imagingstudy oid="{$imagingstudy/oid/text()}">
   <image_date>{$imagingstudy/image_date/text()}</image_date>
   <billed>{$imagingstudy/billed/text()}</billed>
   <prefix>{$imagingstudy/prefix/text()}</prefix>
   <subject>{$imagingstudy/subject/text()}</subject>
   <suffix>{$imagingstudy/suffix/text()}</suffix>
   <computed_image_pathname>{$imagingstudy/computed_image_pathname/text()}</computed_image_pathname>
   <computed_image_filename>{$imagingstudy/computed_image_filename/text()}</computed_image_filename>
   <computed_coords_pathname>{$imagingstudy/computed_coords_pathname/text()}</computed_coords_pathname>
   <computed_coords_filename>{$imagingstudy/computed_coords_filename/text()}</computed_coords_filename>
   <lowres_surface_pathname>{$imagingstudy/lowres_surface_pathname/text()}</lowres_surface_pathname>
   <lowres_surface_filename>{$imagingstudy/lowres_surface_filename/text()}</lowres_surface_filename>
   <aligned_pathname>{$imagingstudy/aligned_pathname/text()}</aligned_pathname>
   {
    for $scene in table("Scene")
    where data($scene/imaging_study) = data($imagingstudy/oid)
    return
    <scene oid="{$scene/oid/text()}">
      <imaging_study>{$scene/imaging_study/text()}</imaging_study>
      <description>{$scene/description/text()}</description>
      <description_file>
      {
       for $scenefile in table("File")
       where data($scenefile/oid) = data($scene/description_file)
       return
       <file oid="{$scenefile/oid/text()}">
         <label>{$scenefile/label/text()}</label>
         <domain>{$scenefile/domain/text()}</domain>
         <locator>{$scenefile/locator/text()}</locator>
         <source>{$scenefile/source/text()}</source>
         <mime_type>{$scenefile/mime_type/text()}</mime_type>
         <submit_date>{$scenefile/submit_date/text()}</submit_date>
         <submitted_by>
         {
          for $scenefilesubmitter in table("UserPerson")
          where data($scenefilesubmitter/oid) = data($scenefile/submitted_by)
          return
          <userperson oid="{$scenefilesubmitter/oid/text()}">
            <login>{$scenefilesubmitter/login/text()}</login>
            <first_name>{$scenefilesubmitter/first_name/text()}</first_name>
            <last_name>{$scenefilesubmitter/last_name/text()}</last_name>
            <email>{$scenefilesubmitter/email/text()}</email>
            <user_group>{$scenefilesubmitter/user_group/text()}</user_group>
          </userperson>
         }
         </submitted_by>
         <version>{$scenefile/version/text()}</version>
         <context>{$scenefile/context/text()}</context>
         <description>{$scenefile/description/text()}</description>
       </file>
      }
      </description_file>
      <preference>{$scene/preference/text()}</preference>
      <ismapscene>{$scene/ismapscene/text()}</ismapscene>
      {
       for $rendering in table("Rendering")
       where data($rendering/scene) = data($scene/oid)
       return
       <rendering oid="{$rendering/oid/text()}">
         <rendering_type>{$rendering/rendering_type/text()}</rendering_type>
         <preference>{$rendering/preference/text()}</preference>
         <image>
   {
    for $renderingfile in table("File")
    where data($renderingfile/oid) = data($rendering/image)
    return
    <file oid="{$renderingfile/oid/text()}">
      <label>{$renderingfile/label/text()}</label>
      <domain>{$renderingfile/domain/text()}</domain>
      <locator>{$renderingfile/locator/text()}</locator>
      <source>{$renderingfile/source/text()}</source>
      <mime_type>{$renderingfile/mime_type/text()}</mime_type>
      <submit_date>{$renderingfile/submit_date/text()}</submit_date>
      <submitted_by>
      {
       for $renderingfilesubmitter in table("UserPerson")
       where data($renderingfilesubmitter/oid) = data($renderingfile/submitted_by)
       return
       <userperson oid="{$renderingfilesubmitter/oid/text()}">
         <login>{$renderingfilesubmitter/login/text()}</login>
         <first_name>{$renderingfilesubmitter/first_name/text()}</first_name>
         <last_name>{$renderingfilesubmitter/last_name/text()}</last_name>
         <email>{$renderingfilesubmitter/email/text()}</email>
         <user_group>{$renderingfilesubmitter/user_group/text()}</user_group>
       </userperson>
      }
      </submitted_by>
      <version>{$renderingfile/version/text()}</version>
      <context>{$renderingfile/context/text()}</context>
      <description>{$renderingfile/description/text()}</description>
    </file>
   }
   </image>
   <image_pathname>{$rendering/image_pathname/text()}</image_pathname>
   <image_filename>{$rendering/image_filename/text()}</image_filename>
 </rendering>
}
{
 for $scenecomponent in table("SceneComponent")
 where data($scenecomponent/scene) = data($scene/oid)
 return
 <scenecomponent oid="{$scenecomponent/oid/text()}">
   <description>{$scenecomponent/description/text()}</description>
   {
    for $surfacemodel in table("SurfaceModel")
    where data($surfacemodel/oid) = data($scenecomponent/surface_model)
    return
    <surfacemodel oid="{$surfacemodel/oid/text()}">
      <format>{$surfacemodel/format/text()}</format>
      <model_file>
      {
       for $modelfile in table("File")
       where data($modelfile/oid) = data($surfacemodel/model_file)
       return
       <file oid="{$modelfile/oid/text()}">
         <label>{$modelfile/label/text()}</label>
         <domain>{$modelfile/domain/text()}</domain>
         <locator>{$modelfile/locator/text()}</locator>
         <source>{$modelfile/source/text()}</source>
         <mime_type>{$modelfile/mime_type/text()}</mime_type>
         <submit_date>{$modelfile/submit_date/text()}</submit_date>
         <submitted_by>
         {
          for $modelfilesubmitter in table("UserPerson")
          where data($modelfilesubmitter/oid) = data($modelfile/submitted_by)
          return
          <userperson oid="{$modelfilesubmitter/oid/text()}">
            <login>{$modelfilesubmitter/login/text()}</login>
            <first_name>{$modelfilesubmitter/first_name/text()}</first_name>
            <last_name>{$modelfilesubmitter/last_name/text()}</last_name>
            <email>{$modelfilesubmitter/email/text()}</email>
            <user_group>{$modelfilesubmitter/user_group/text()}</user_group>
          </userperson>
         }
         </submitted_by>
   <version>{$modelfile/version/text()}</version>
   <context>{$modelfile/context/text()}</context>
   <description>{$modelfile/description/text()}</description>
 </file>
 }
</model_file>
<model_pathname>{$surfacemodel/model_pathname/text()}</model_pathname>
<model_filename>{$surfacemodel/model_filename/text()}</model_filename>
<preference>{$surfacemodel/preference/text()}</preference>
{
 for $radialslicemodelinstance in table("RadialSliceModelInstance")
 where data($radialslicemodelinstance/oid) = data($surfacemodel/model_instance)
 return
 <radialslicemodelinstance oid="{$radialslicemodelinstance/oid/text()}">
   <landmarks_file>
   {
    for $landmarksfile in table("File")
    where data($landmarksfile/oid) = data($radialslicemodelinstance/landmarks_file)
    return
    <file oid="{$landmarksfile/oid/text()}">
      <label>{$landmarksfile/label/text()}</label>
      <domain>{$landmarksfile/domain/text()}</domain>
      <locator>{$landmarksfile/locator/text()}</locator>
      <source>{$landmarksfile/source/text()}</source>
      <mime_type>{$landmarksfile/mime_type/text()}</mime_type>
      <submit_date>{$landmarksfile/submit_date/text()}</submit_date>
      <submitted_by>
      {
       for $landmarksfilesubmitter in table("UserPerson")
       where data($landmarksfilesubmitter/oid) = data($landmarksfile/submitted_by)
       return
       <userperson oid="{$landmarksfilesubmitter/oid/text()}">
         <login>{$landmarksfilesubmitter/login/text()}</login>
         <first_name>{$landmarksfilesubmitter/first_name/text()}</first_name>
         <last_name>{$landmarksfilesubmitter/last_name/text()}</last_name>
         <email>{$landmarksfilesubmitter/email/text()}</email>
         <user_group>{$landmarksfilesubmitter/user_group/text()}</user_group>
       </userperson>
      }
      </submitted_by>
      <version>{$landmarksfile/version/text()}</version>
      <context>{$landmarksfile/context/text()}</context>
      <description>{$landmarksfile/description/text()}</description>
    </file>
   }
   </landmarks_file>
   <instance_file>
   {
    for $instancefile in table("File")
    where data($instancefile/oid) = data($radialslicemodelinstance/instance_file)
    return
    <file oid="{$instancefile/oid/text()}">
      <label>{$instancefile/label/text()}</label>
      <domain>{$instancefile/domain/text()}</domain>
      <locator>{$instancefile/locator/text()}</locator>
      <source>{$instancefile/source/text()}</source>
      <mime_type>{$instancefile/mime_type/text()}</mime_type>
      <submit_date>{$instancefile/submit_date/text()}</submit_date>
      <submitted_by>
      {
       for $instancefilesubmitter in table("UserPerson")
       where data($instancefilesubmitter/oid) = data($instancefile/submitted_by)
       return
       <userperson oid="{$instancefilesubmitter/oid/text()}">
         <login>{$instancefilesubmitter/login/text()}</login>
         <first_name>{$instancefilesubmitter/first_name/text()}</first_name>
         <last_name>{$instancefilesubmitter/last_name/text()}</last_name>
         <email>{$instancefilesubmitter/email/text()}</email>
         <user_group>{$instancefilesubmitter/user_group/text()}</user_group>
       </userperson>
      }
      </submitted_by>
      <version>{$instancefile/version/text()}</version>
```

```
              <context>{$instancefile/context/text()}</context>
              <description>{$instancefile/description/text()}</description>
            </file>
           }
           </instance_file>
           <expansion_factor>{$radialslicemodelinstance/expansion_factor/text()}</expansion_factor>
           <instance_pathname>{$radialslicemodelinstance/instance_pathname/text()}</instance_pathname>
           <instance_filename>{$radialslicemodelinstance/instance_filename/text()}</instance_filename>
           <preference>{$radialslicemodelinstance/preference/text()}</preference>
           <landmarks_pathname>{$radialslicemodelinstance/landmarks_pathname/text()}
              </landmarks_pathname>
           <landmarks_filename>{$radialslicemodelinstance/landmarks_filename/text()}
              </landmarks_filename>
           {
            for $radialslicemodel in table("RadialSliceModel")
            where data($radialslicemodelinstance/model) = data($radialslicemodel/oid)
            return
            <radialslicemodel oid="{$radialslicemodel/oid/text()}">
              <pathname>{$radialslicemodel/pathname/text()}</pathname>
              <filename>{$radialslicemodel/filename/text()}</filename>
              <comment>{$radialslicemodel/comment/text()}</comment>
              <theta_radials>{$radialslicemodel/theta_radials/text()}</theta_radials>
              <slices>{$radialslicemodel/slices/text()}</slices>
              <training_set>{$radialslicemodel/training_set/text()}</training_set>
              <model_file>
              {
               for $modelfile in table("File")
               where data($modelfile/oid) = data($radialslicemodel/model_file)
               return
               <file oid="{$modelfile/oid/text()}">
                 <label>{$modelfile/label/text()}</label>
                 <domain>{$modelfile/domain/text()}</domain>
                 <locator>{$modelfile/locator/text()}</locator>
                 <source>{$modelfile/source/text()}</source>
                 <mime_type>{$modelfile/mime_type/text()}</mime_type>
                 <submit_date>{$modelfile/submit_date/text()}</submit_date>
                 <submitted_by>
                 {
                  for $modelfilesubmitter in table("UserPerson")
                  where data($modelfilesubmitter/oid) = data($modelfile/submitted_by)
                  return
                  <userperson oid="{$modelfilesubmitter/oid/text()}">
                    <login>{$modelfilesubmitter/login/text()}</login>
                    <first_name>{$modelfilesubmitter/first_name/text()}</first_name>
                    <last_name>{$modelfilesubmitter/last_name/text()}</last_name>
                    <email>{$modelfilesubmitter/email/text()}</email>
                    <user_group>{$modelfilesubmitter/user_group/text()}</user_group>
                  </userperson>
                 }
                 </submitted_by>
                 <version>{$modelfile/version/text()}</version>
                 <context>{$modelfile/context/text()}</context>
                 <description>{$modelfile/description/text()}</description>
               </file>
              }
              </model_file>
              <preference>{$radialslicemodel/preference/text()}</preference>
            </radialslicemodel>
           }
         </radialslicemodelinstance>
        }
      </surfacemodel>
    }
   </scenecomponent>
  }
</scene>
}
{
 for $mrexam in table("MRExam")
 where data($mrexam/imaging_study) = data($imagingstudy/oid)
 return
 <mrexam oid="{$mrexam/oid/text()}">
   <exam_num>{$mrexam/exam_num/text()}</exam_num>
```

```
<description>{$mrexam/description/text()}</description>
<import_date>{$mrexam/import_date/text()}</import_date>
<location>{$mrexam/location/text()}</location>
<import_info>{$mrexam/import_info/text()}</import_info>
{
 for $mrseries in table("MRSeries")
 where data($mrseries/mrexam) = data($mrexam/oid)
 return
 <mrseries oid="{$mrseries/oid/text()}">
   <location>{$mrseries/location/text()}</location>
   <showing>{$mrseries/showing/text()}</showing>
   <total_images>{$mrseries/total_images/text()}</total_images>
   <plane>{$mrseries/plane/text()}</plane>
   <scan_start>{$mrseries/scan_start/text()}</scan_start>
   <scan_end>{$mrseries/scan_end/text()}</scan_end>
   <psd>{$mrseries/psd/text()}</psd>
   <type>{$mrseries/:type/text()}</type>
   <description>{$mrseries/description/text()}</description>
   <fov_x>{$mrseries/fov_x/text()}</fov_x>
   <fov_y>{$mrseries/fov_y/text()}</fov_y>
   <height>{$mrseries/height/text()}</height>
   <width>{$mrseries/width/text()}</width>
   <bytes_per_pixel>{$mrseries/bytes_per_pixel/text()}</bytes_per_pixel>
   <bits_per_pixel>{$mrseries/bits_per_pixel/text()}</bits_per_pixel>
   <optical_disk>{$mrseries/optical_disk/text()}</optical_disk>
   <start_img>{$mrseries/start_img/text()}</start_img>
   <stop_img>{$mrseries/stop_img/text()}</stop_img>
   <threshold>{$mrseries/threshold/text()}</threshold>
   <tissue>{$mrseries/tissue/text()}</tissue>
   <first>{$mrseries/first/text()}</first>
   <last>{$mrseries/last/text()}</last>
   <label>{$mrseries/label/text()}</label>
   <thickness>{$mrseries/thickness/text()}</thickness>
   <spacing>{$mrseries/spacing/text()}</spacing>
   {
    for $mrslice in table("MRSlice")
    where data($mrslice/mrseries) = data($mrseries/oid)
    return
    <mrslice oid="{$mrslice/oid/text()}">
      <sequence_num>{$mrslice/sequence_num/text()}</sequence_num>
      <image_file>
      {
       for $imagefile in table("File")
       where data($imagefile/oid) = data($mrslice/image_file)
       return
       <file oid="{$imagefile/oid/text()}">
         <label>{$imagefile/label/text()}</label>
         <domain>{$imagefile/domain/text()}</domain>
         <locator>{$imagefile/locator/text()}</locator>
         <source>{$imagefile/source/text()}</source>
         <mime_type>{$imagefile/mime_type/text()}</mime_type>
         <submit_date>{$imagefile/submit_date/text()}</submit_date>
         <submitted_by>
         {
          for $imagefilesubmitter in table("UserPerson")
          where data($imagefilesubmitter/oid) = data($imagefile/submitted_by)
          return
          <userperson oid="{$imagefilesubmitter/oid/text()}">
            <login>{$imagefilesubmitter/login/text()}</login>
            <first_name>{$imagefilesubmitter/first_name/text()}</first_name>
            <last_name>{$imagefilesubmitter/last_name/text()}</last_name>
            <email>{$imagefilesubmitter/email/text()}</email>
            <user_group>{$imagefilesubmitter/user_group/text()}</user_group>
          </userperson>
         }
         </submitted_by>
         <version>{$imagefile/version/text()}</version>
         <context>{$imagefile/context/text()}</context>
         <description>{$imagefile/description/text()}</description>
       </file>
       }
      </image_file>
    </mrslice>
```

```
}
{
 for $alignedvolume in table("AlignedVolume")
 where data($alignedvolume/series) = data($mrseries/oid)
 return
 <alignedvolume oid="{$alignedvolume/oid/text()}">
   <format>{$alignedvolume/format/text()}</format>
   <volume_file>
   {
    for $volumefile in table("File")
    where data($volumefile/oid) = data($alignedvolume/volume_file)
    return
    <file oid="{$volumefile/oid/text()}">
      <label>{$volumefile/label/text()}</label>
      <domain>{$volumefile/domain/text()}</domain>
      <locator>{$volumefile/locator/text()}</locator>
      <source>{$volumefile/source/text()}</source>
      <mime_type>{$volumefile/mime_type/text()}</mime_type>
      <submit_date>{$volumefile/submit_date/text()}</submit_date>
      <submitted_by>
      {
       for $volumefilesubmitter in table("UserPerson")
       where data($volumefilesubmitter/oid) = data($volumefile/submitted_by)
       return
       <userperson oid="{$volumefilesubmitter/oid/text()}">
         <login>{$volumefilesubmitter/login/text()}</login>
         <first_name>{$volumefilesubmitter/first_name/text()}</first_name>
         <last_name>{$volumefilesubmitter/last_name/text()}</last_name>
         <email>{$volumefilesubmitter/email/text()}</email>
         <user_group>{$volumefilesubmitter/user_group/text()}</user_group>
       </userperson>
      }
      </submitted_by>
      <version>{$volumefile/version/text()}</version>
      <context>{$volumefile/context/text()}</context>
      <description>{$volumefile/description/text()}</description>
    </file>
   }
 </volume_file>
 <filename>{$alignedvolume/filename/text()}</filename>
 <tissue>{$alignedvolume/tissue/text()}</tissue>
 <patient>{$alignedvolume/patient/text()}</patient>
 {
  for $surfacemodel in table("SurfaceModel")
  where data($surfacemodel/volume) = data($alignedvolume/oid)
  return
  <surfacemodel oid="{$surfacemodel/oid/text()}">
    <format>{$surfacemodel/format/text()}</format>
    <model_file>
    {
     for $modelfile in table("File")
     where data($modelfile/oid) = data($surfacemodel/model_file)
     return
     <file oid="{$modelfile/oid/text()}">
       <label>{$modelfile/label/text()}</label>
       <domain>{$modelfile/domain/text()}</domain>
       <locator>{$modelfile/locator/text()}</locator>
       <source>{$modelfile/source/text()}</source>
       <mime_type>{$modelfile/mime_type/text()}</mime_type>
       <submit_date>{$modelfile/submit_date/text()}</submit_date>
       <submitted_by>
       {
        for $modelfilesubmitter in table("UserPerson")
        where data($modelfilesubmitter/oid) = data($modelfile/submitted_by)
        return
        <userperson oid="{$modelfilesubmitter/oid/text()}">
          <login>{$modelfilesubmitter/login/text()}</login>
          <first_name>{$modelfilesubmitter/first_name/text()}</first_name>
          <last_name>{$modelfilesubmitter/last_name/text()}</last_name>
          <email>{$modelfilesubmitter/email/text()}</email>
          <user_group>{$modelfilesubmitter/user_group/text()}</user_group>
        </userperson>
       }
   </submitted_by>
   <version>{$modelfile/version/text()}</version>
   <context>{$modelfile/context/text()}</context>
   <description>{$modelfile/description/text()}</description>
 </file>
}
</model_file>
<model_pathname>{$surfacemodel/model_pathname/text()}</model_pathname>
<model_filename>{$surfacemodel/model_filename/text()}</model_filename>
<preference>{$surfacemodel/preference/text()}</preference>
{
 for $radialslicemodelinstance in table("RadialSliceModelInstance")
 where data($radialslicemodelinstance/oid) = data($surfacemodel/model_instance)
 return
 <radialslicemodelinstance oid="{$radialslicemodelinstance/oid/text()}">
    <landmarks_file>
    {
     for $landmarksfile in table("File")
     where data($landmarksfile/oid) = data($radialslicemodelinstance/landmarks_file)
     return
     <file oid="{$landmarksfile/oid/text()}">
       <label>{$landmarksfile/label/text()}</label>
       <domain>{$landmarksfile/domain/text()}</domain>
       <locator>{$landmarksfile/locator/text()}</locator>
       <source>{$landmarksfile/source/text()}</source>
       <mime_type>{$landmarksfile/mime_type/text()}</mime_type>
       <submit_date>{$landmarksfile/submit_date/text()}</submit_date>
       <submitted_by>
       {
        for $landmarksfilesubmitter in table("UserPerson")
        where data($landmarksfilesubmitter/oid) = data($landmarksfile/submitted_by)
        return
        <userperson oid="{$landmarksfilesubmitter/oid/text()}">
          <login>{$landmarksfilesubmitter/login/text()}</login>
          <first_name>{$landmarksfilesubmitter/first_name/text()}</first_name>
          <last_name>{$landmarksfilesubmitter/last_name/text()}</last_name>
          <email>{$landmarksfilesubmitter/email/text()}</email>
          <user_group>{$landmarksfilesubmitter/user_group/text()}</user_group>
        </userperson>
       }
       </submitted_by>
       <version>{$landmarksfile/version/text()}</version>
       <context>{$landmarksfile/context/text()}</context>
       <description>{$landmarksfile/description/text()}</description>
      </file>
     }
    </landmarks_file>
    <instance_file>
    {
     for $instancefile in table("File")
     where data($instancefile/oid) = data($radialslicemodelinstance/instance_file)
     return
     <file oid="{$instancefile/oid/text()}">
       <label>{$instancefile/label/text()}</label>
       <domain>{$instancefile/domain/text()}</domain>
       <locator>{$instancefile/locator/text()}</locator>
       <source>{$instancefile/source/text()}</source>
       <mime_type>{$instancefile/mime_type/text()}</mime_type>
       <submit_date>{$instancefile/submit_date/text()}</submit_date>
       <submitted_by>
       {
        for $instancefilesubmitter in table("UserPerson")
        where data($instancefilesubmitter/oid) = data($instancefile/submitted_by)
        return
        <userperson oid="{$instancefilesubmitter/oid/text()}">
          <login>{$instancefilesubmitter/login/text()}</login>
          <first_name>{$instancefilesubmitter/first_name/text()}</first_name>
          <last_name>{$instancefilesubmitter/last_name/text()}</last_name>
          <email>{$instancefilesubmitter/email/text()}</email>
          <user_group>{$instancefilesubmitter/user_group/text()}</user_group>
        </userperson>
       }
       </submitted_by>
```

```
          <version>{$instancefile/version/text()}</version>
          <context>{$instancefile/context/text()}</context>
          <description>{$instancefile/description/text()}</description>
        </file>
       }
       </instance_file>
       <expansion_factor>{$radialslicemodelinstance/expansion_factor/text()}</expansion_factor>
       <instance_pathname>{$radialslicemodelinstance/instance_pathname/text()}
          </instance_pathname>
       <instance_filename>{$radialslicemodelinstance/instance_filename/text()}
          </instance_filename>
       <preference>{$radialslicemodelinstance/preference/text()}</preference>
       <landmarks_pathname>{$radialslicemodelinstance/landmarks_pathname/text()}
          </landmarks_pathname>
       <landmarks_filename>{$radialslicemodelinstance/landmarks_filename/text()}
          </landmarks_filename>
       {
        for $radialslicemodel in table("RadialSliceModel")
        where data($radialslicemodelinstance/model) = data($radialslicemodel/oid)
        return
        <radialslicemodel oid="{$radialslicemodel/oid/text()}">
          <pathname>{$radialslicemodel/pathname/text()}</pathname>
          <filename>{$radialslicemodel/filename/text()}</filename>
          <comment>{$radialslicemodel/comment/text()}</comment>
          <theta_radials>{$radialslicemodel/theta_radials/text()}</theta_radials>
          <slices>{$radialslicemodel/slices/text()}</slices>
          <training_set>{$radialslicemodel/training_set/text()}</training_set>
          <model_file>
          {
           for $modelfile in table("File")
           where data($modelfile/oid) = data($radialslicemodel/model_file)
           return
           <file oid="{$modelfile/oid/text()}">
             <label>{$modelfile/label/text()}</label>
             <domain>{$modelfile/domain/text()}</domain>
             <locator>{$modelfile/locator/text()}</locator>
             <source>{$modelfile/source/text()}</source>
             <mime_type>{$modelfile/mime_type/text()}</mime_type>
             <submit_date>{$modelfile/submit_date/text()}</submit_date>
             <submitted_by>
             {
              for $modelfilesubmitter in table("UserPerson")
              where data($modelfilesubmitter/oid) = data($modelfile/submitted_by)
              return
              <userperson oid="{$modelfilesubmitter/oid/text()}">
                <login>{$modelfilesubmitter/login/text()}</login>
                <first_name>{$modelfilesubmitter/first_name/text()}</first_name>
                <last_name>{$modelfilesubmitter/last_name/text()}</last_name>
                <email>{$modelfilesubmitter/email/text()}</email>
                <user_group>{$modelfilesubmitter/user_group/text()}</user_group>
              </userperson>
             }
             </submitted_by>
             <version>{$modelfile/version/text()}</version>
             <context>{$modelfile/context/text()}</context>
             <description>{$modelfile/description/text()}</description>
           </file>
          }
          </model_file>
          <preference>{$radialslicemodel/preference/text()}</preference>
        </radialslicemodel>
      }
    </radialslicemodelinstance>
   }
 </surfacemodel>
}
{
 for $radialslicemodelinstance in table("RadialSliceModelInstance")
 where data($radialslicemodelinstance/volume) = data($alignedvolume/oid)
 return
 <radialslicemodelinstance oid="{$radialslicemodelinstance/oid/text()}">
   <landmarks_file>
   {
```

```
   for $landmarksfile in table("File")
   where data($landmarksfile/oid) = data($radialslicemodelinstance/landmarks_file)
   return
   <file oid="{$landmarksfile/oid/text()}">
     <label>{$landmarksfile/label/text()}</label>
     <domain>{$landmarksfile/domain/text()}</domain>
     <locator>{$landmarksfile/locator/text()}</locator>
     <source>{$landmarksfile/source/text()}</source>
     <mime_type>{$landmarksfile/mime_type/text()}</mime_type>
     <submit_date>{$landmarksfile/submit_date/text()}</submit_date>
     <submitted_by>
     {
      for $landmarksfilesubmitter in table("UserPerson")
      where data($landmarksfilesubmitter/oid) = data($landmarksfile/submitted_by)
      return
      <userperson oid="{$landmarksfilesubmitter/oid/text()}">
        <login>{$landmarksfilesubmitter/login/text()}</login>
        <first_name>{$landmarksfilesubmitter/first_name/text()}</first_name>
        <last_name>{$landmarksfilesubmitter/last_name/text()}</last_name>
        <email>{$landmarksfilesubmitter/email/text()}</email>
        <user_group>{$landmarksfilesubmitter/user_group/text()}</user_group>
      </userperson>
     }
     </submitted_by>
     <version>{$landmarksfile/version/text()}</version>
     <context>{$landmarksfile/context/text()}</context>
     <description>{$landmarksfile/description/text()}</description>
   </file>
  }
  </landmarks_file>
  <instance_file>
  {
   for $instancefile in table("File")
   where data($instancefile/oid) = data($radialslicemodelinstance/instance_file)
   return
   <file oid="{$instancefile/oid/text()}">
     <label>{$instancefile/label/text()}</label>
     <domain>{$instancefile/domain/text()}</domain>
     <locator>{$instancefile/locator/text()}</locator>
     <source>{$instancefile/source/text()}</source>
     <mime_type>{$instancefile/mime_type/text()}</mime_type>
     <submit_date>{$instancefile/submit_date/text()}</submit_date>
     <submitted_by>
     {
      for $instancefilesubmitter in table("UserPerson")
      where data($instancefilesubmitter/oid) = data($instancefile/submitted_by)
      return
      <userperson oid="{$instancefilesubmitter/oid/text()}">
        <login>{$instancefilesubmitter/login/text()}</login>
        <first_name>{$instancefilesubmitter/first_name/text()}</first_name>
        <last_name>{$instancefilesubmitter/last_name/text()}</last_name>
        <email>{$instancefilesubmitter/email/text()}</email>
        <user_group>{$instancefilesubmitter/user_group/text()}</user_group>
      </userperson>
     }
     </submitted_by>
     <version>{$instancefile/version/text()}</version>
     <context>{$instancefile/context/text()}</context>
     <description>{$instancefile/description/text()}</description>
   </file>
  }
</instance_file>
<expansion_factor>{$radialslicemodelinstance/expansion_factor/text()}</expansion_factor>
<instance_pathname>{$radialslicemodelinstance/instance_pathname/text()}</instance_pathname>
<instance_filename>{$radialslicemodelinstance/instance_filename/text()}</instance_filename>
<preference>{$radialslicemodelinstance/preference/text()}</preference>
<landmarks_pathname>{$radialslicemodelinstance/landmarks_pathname/text()}</landmarks_pathname>
<landmarks_filename>{$radialslicemodelinstance/landmarks_filename/text()}</landmarks_filename>
{
 for $radialslicemodel in table("RadialSliceModel")
 where data($radialslicemodelinstance/model) = data($radialslicemodel/oid)
 return
 <radialslicemodel oid="{$radialslicemodel/oid/text()}">
```

```
                   <pathname>{$radialslicemodel/pathname/text()}</pathname>
                   <filename>{$radialslicemodel/filename/text()}</filename>
                   <comment>{$radialslicemodel/comment/text()}</comment>
                   <theta_radials>{$radialslicemodel/theta_radials/text()}</theta_radials>
                   <slices>{$radialslicemodel/slices/text()}</slices>
                   <training_set>{$radialslicemodel/training_set/text()}</training_set>
                   <model_file>
                   {
                    for $modelfile in table("File")
                    where data($modelfile/oid) = data($radialslicemodel/model_file)
                    return
                    <file oid="{$modelfile/oid/text()}">
                      <label>{$modelfile/label/text()}</label>
                      <domain>{$modelfile/domain/text()}</domain>
                      <locator>{$modelfile/locator/text()}</locator>
                      <source>{$modelfile/source/text()}</source>
                      <mime_type>{$modelfile/mime_type/text()}</mime_type>
                      <submit_date>{$modelfile/submit_date/text()}</submit_date>
                      <submitted_by>
                      {
                       for $modelfilesubmitter in table("UserPerson")
                       where data($modelfilesubmitter/oid) = data($modelfile/submitted_by)
                       return
                       <userperson oid="{$modelfilesubmitter/oid/text()}">
                         <login>{$modelfilesubmitter/login/text()}</login>
                         <first_name>{$modelfilesubmitter/first_name/text()}</first_name>
                         <last_name>{$modelfilesubmitter/last_name/text()}</last_name>
                         <email>{$modelfilesubmitter/email/text()}</email>
                         <user_group>{$modelfilesubmitter/user_group/text()}</user_group>
                       </userperson>
                      }
                      </submitted_by>
                      <version>{$modelfile/version/text()}</version>
                      <context>{$modelfile/context/text()}</context>
                      <description>{$modelfile/description/text()}</description>
                    </file>
                   }
                   </model_file>
                   <preference>{$radialslicemodel/preference/text()}</preference>
                 </radialslicemodel>
                }
                </radialslicemodelinstance>
               }
               </alignedvolume>
              }
            </mrseries>
           }
      </mrexam>
        }
    </imagingstudy>
      }
  </patient>
    }
</root>
```

C.3 The Mondial Database

The Mondial database is a case study for information extraction and integration. The database schema is as follows:

```
CREATE TABLE Country
   (Name VARCHAR2(32) NOT NULL UNIQUE,
    Code VARCHAR2(4) CONSTRAINT CountryKey PRIMARY KEY,
    Capital VARCHAR2(35),
    Province VARCHAR2(32)
    Area NUMBER,
    Population NUMBER);

CREATE TABLE City
   (Name VARCHAR2(35),
    Country VARCHAR2(4),
    Province VARCHAR2(32),
    Population NUMBER,
    Longitude NUMBER,
    Latitude NUMBER,
    CONSTRAINT CityKey PRIMARY KEY (Name, Country, Province));

CREATE TABLE Province
   (Name VARCHAR2(32) CONSTRAINT PrName NOT NULL ,
    Country  VARCHAR2(4) CONSTRAINT PrCountry NOT NULL ,
    Population NUMBER,
    Area NUMBER,
    Capital VARCHAR2(35),
    CapProv VARCHAR2(32),
    CONSTRAINT PrKey PRIMARY KEY (Name, Country));

CREATE TABLE Economy
   (Country VARCHAR2(4) CONSTRAINT EconomyKey PRIMARY KEY,
    GDP NUMBER,
    Agriculture NUMBER,
    Service NUMBER,
    Industry NUMBER,
    Inflation NUMBER);

CREATE TABLE Population
   (Country VARCHAR2(4) CONSTRAINT PopKey PRIMARY KEY,
    Population_Growth NUMBER,
    Infant_Mortality NUMBER);

CREATE TABLE Politics
   (Country VARCHAR2(4) CONSTRAINT PoliticsKey PRIMARY KEY,
    Independence DATE,
    Government VARCHAR2(120));

CREATE TABLE Language
   (Country VARCHAR2(4),
    Name VARCHAR2(50),
    Percentage NUMBER,
    CONSTRAINT LanguageKey PRIMARY KEY (Name, Country));

CREATE TABLE Religion
   (Country VARCHAR2(4),
    Name VARCHAR2(50),
    Percentage NUMBER,
    CONSTRAINT ReligionKey PRIMARY KEY (Name, Country));

CREATE TABLE Ethnic_Group
   (Country VARCHAR2(4),
    Name VARCHAR2(50),
    Percentage NUMBER,
    CONSTRAINT EthnicKey PRIMARY KEY (Name, Country));

CREATE TABLE Continent
   (Name VARCHAR2(20) CONSTRAINT ContinentKey PRIMARY KEY,
    Area NUMBER(10));

CREATE TABLE borders
   (Country1 VARCHAR2(4),
    Country2 VARCHAR2(4),
    Length NUMBER,
    CONSTRAINT BorderKey PRIMARY KEY (Country1,Country2) );

CREATE TABLE encompasses
   (Country VARCHAR2(4) NOT NULL,
    Continent VARCHAR2(20) NOT NULL,
    Percentage NUMBER,
    CONSTRAINT EncompassesKey PRIMARY KEY (Country,Continent));

CREATE TABLE Organization
   (Abbreviation VARCHAR2(12) PRIMARY KEY,
    Name VARCHAR2(80) NOT NULL,
    City VARCHAR2(35) ,
    Country VARCHAR2(4) ,
    Province VARCHAR2(32) ,
    Established DATE,
    CONSTRAINT OrgNameUnique UNIQUE (Name));

CREATE TABLE is_member
   (Country VARCHAR2(4),
    Organization VARCHAR2(12),
    Type VARCHAR2(30) DEFAULT 'member',
    CONSTRAINT MemberKey PRIMARY KEY (Country,Organization));

CREATE OR REPLACE TYPE GeoCoord AS OBJECT
   (Longitude NUMBER,
    Latitude NUMBER);

CREATE TABLE Mountain
   (Name VARCHAR2(20) CONSTRAINT MountainKey PRIMARY KEY,
    Height NUMBER,
    Coordinates GeoCoord);

CREATE TABLE Desert
   (Name VARCHAR2(25) CONSTRAINT DesertKey PRIMARY KEY,
    Area NUMBER);

CREATE TABLE Island
   (Name VARCHAR2(25) CONSTRAINT IslandKey PRIMARY KEY,
    Islands VARCHAR2(25),
    Area NUMBER,
    Coordinates GeoCoord);

CREATE TABLE Lake
   (Name VARCHAR2(25) CONSTRAINT LakeKey PRIMARY KEY,
    Area NUMBER);

CREATE TABLE Sea
   (Name VARCHAR2(25) CONSTRAINT SeaKey PRIMARY KEY,
    Depth NUMBER);

CREATE TABLE River
   (Name VARCHAR2(20) CONSTRAINT RiverKey PRIMARY KEY,
    River VARCHAR2(20),
    Lake VARCHAR2(20),
    Sea VARCHAR2(25),
    Length NUMBER);

CREATE TABLE geo_Mountain
   (Mountain VARCHAR2(20),
    Country VARCHAR2(4),
    Province VARCHAR2(32),
    CONSTRAINT GMountainKey PRIMARY KEY (Province,Country,Mountain));

CREATE TABLE geo_Desert
   (Desert VARCHAR2(25),
    Country VARCHAR2(4),
    Province VARCHAR2(32),
    CONSTRAINT GDesertKey PRIMARY KEY (Province, Country, Desert));

CREATE TABLE geo_Island
   (Island VARCHAR2(25),
    Country VARCHAR2(4),
    Province VARCHAR2(32),
    CONSTRAINT GIslandKey PRIMARY KEY (Province, Country, Island));

CREATE TABLE geo_River
   (River VARCHAR2(20),
    Country VARCHAR2(4),
    Province VARCHAR2(32),
    CONSTRAINT GRiverKey PRIMARY KEY (Province ,Country, River));

CREATE TABLE geo_Sea
   (Sea VARCHAR2(25),
    Country VARCHAR2(4),
    Province VARCHAR2(32),
    CONSTRAINT GSeaKey PRIMARY KEY (Province, Country, Sea));
```

```
CREATE TABLE geo_Lake
   (Lake VARCHAR2(25),
    Country VARCHAR2(4),
    Province VARCHAR2(32),
    CONSTRAINT GLakeKey PRIMARY KEY (Province, Country, Lake));

CREATE TABLE merges_with
   (Sea1 VARCHAR2(25),
    Sea2 VARCHAR2(25),
    CONSTRAINT MergesWithKey PRIMARY KEY (Sea1,Sea2));

CREATE TABLE located
   (City VARCHAR2(35) ,
    Province VARCHAR2(32) ,
    Country VARCHAR2(4) ,
    River VARCHAR2(20),
    Lake VARCHAR2(25),
    Sea VARCHAR2(25));
```

We analyzed Mondial 2.0 XML. We present a fragment of this view below. The entire view is available at `http://www.informatik.uni-freiburg.de/~may/Mondial/mondial-2.0.xml`.

```
<mondial xmlns:redirect="org.apache.xalan.xslt.extensions.Redirect">
  <country car_code="AL" area="28750" capital="cty-cid-cia-Albania-Tirane"
           memberships="org-BSEC org-CE org-CCC org-ECE org-EBRD org-EU org-FAO org-IAEA
           org-IBRD org-ICAO org-Interpol org-IDA org-IFRCS org-IFC org-IFAD org-ILO org-IMO
           org-IMF org-IOC org-IOM org-ISO org-ICRM org-ITU org-Intelsat org-IDB org-ANC
           org-OSCE org-OIC org-PFP org-UN org-UNESCO org-UNIDO org-UNOMIG org-UPU org-WFTU
           org-WHO org-WIPO org-WMO org-WToO org-WTrO">
    <name>Albania</name>
    <population>3249136</population>
    <population_growth>1.34</population_growth>
    <infant_mortality>49.2</infant_mortality>
    <gdp_total>4100</gdp_total>
    <gdp_agri>55</gdp_agri>
    <inflation>16</inflation>
    <government>emerging democracy</government>
    <encompassed continent="europe" percentage="100" />
    <ethnicgroups percentage="3">Greeks</ethnicgroups>
    <ethnicgroups percentage="95">Albanian</ethnicgroups>
    <religions percentage="70">Muslim</religions>
    <religions percentage="10">Roman Catholic</religions>
    <religions percentage="20">Albanian Orthodox</religions>
    <border country="GR" length="282" justice="org-UN" />
    <border country="MK" length="151" justice="org-UN" />
    <border country="YU" length="287" justice="org-UN" />
    <city id="cty-cid-cia-Albania-Tirane" is_country_cap="yes" country="AL">
      <name>Tirane</name>
      <longitude>10.7</longitude>
      <latitude>46.2</latitude>
      <population year="87">192000</population>
    </city>
    <city id="stadt-Shkoder-AL-AL" country="AL">
      <name>Shkoder</name>
      <longitude>19.2</longitude>
      <latitude>42.2</latitude>
      <population year="87">62000</population>
      <located_at watertype="lake" lake="lake-Skutarisee" />
    </city>
    <city>
    ...
    </city>
  </country>
  <country>
  ...
  </country>
  ...
</mondial>
```

The view definition query expressed in UXQuery is as follows:

```
<mondial>
  {for $c in table("country"),
       $pop in table("population"),
       $eco in table("economy"),
       $pol in table("politics")
   where $c/code=$pop/country and
         $c/code=$eco/country and
         $c/code=$pol/country
   return
   <country car_code="{$c/code/text()}" area="{$c/area/text()}" capital="{$c/capital/text()}"
            memberships="">
      <name>{$c/name/text()}</name>
      <population>{$c/population/text()}</population>
      <population_growth>{$pop/population_growth/text()}</population_growth>
      <infant_mortality>{$pop/infant_mortality/text()}</infant_mortality>
      <gdp_total>{$eco/gdp/text()}</gdp_total>
      <gdp_agri>{$eco/ariculture/text()}</gdp_agri>
      <inflation>{$eco/inflation/text()}</inflation>
      <government>{$pol/government/text()}</government>
      {
         for $enc in table("encompasses")
         where $c/code=$emcompasses/country
         return
         <encompassed continent="{enc/continent/text()}" percentage="{enc/percentage/text()}"/>
      }
      (: subelements not expressible - ethnicgroups and religions  :)
      {
         for $border in table("borders")
         where $c/code=$border/country1
         return
         <border country="{border/country2/text()}" length="{border/length/text()}" justice=""/>
      }
      {
         for $border in table("borders")
         where $c/code=$border/country2
         return
         <border country="{border/country1/text()}" length="{border/length/text()}" justice=""/>
      }
      {
         for $province in table("province")
         where $c/code=$province/country
         return
         <province id="" capital="{$province/capital/text()}" country="$province/country/text()">
            <name>{$province/name/text()}</name>
            <population>{$province/population/text()}</population>
            {
               for $cityp in table("city")
               where $c/code=$cityp/country and $province/name=$cityp/province
               return
               <city id="" is_country_capital="" country="{$cityp/country/text()}">
                  <name>{$cityp/name/text()}</name>
                  <longitude>{$cityp/longitude/text()}</longitude>
                  <latitude>{$cityp/latitude/text()}</latitude>
                  <population year="">{$cityp/population/text()}</population>
                  {
                     for $loc in table("located")
                     where $c/code=$loc/country and $province/name=$loc/province
                     and $cityp/name=$loc/city
                     return
                     <located_at watertype="" river="{$loc/river/text()}"
                                              sea="{$loc/sea/text()}"
                                              lake="$loc/lake/text()"/>
                  }
               </city>
            }
         </province>
      }
      {
         for $city in table("city")
         where $c/code=$city/country and $city/province=""
         return
         <city id="" is_country_capital="" country="{$city/country/text()}">
            <name>{$city/name/text()}</name>
            <longitude>{$city/longitude/text()}</longitude>
            <latitude>{$city/latitude/text()}</latitude>
            <population year="">{$city/population/text()}</population>
            {
                for $loc in table("located")
                where $c/code=$loc/country and $province/name=""
                and $city/name=$loc/city
                return
                <located_at watertype="" river="{$loc/river/text()}"
                                          sea="{$loc/sea/text()}"
                                          lake="$loc/lake/text()"/>
            }
         </city>
      }
   </country>
   {
      for $continent in table("continent")
      return
      <continent id="{$continent/name/text()}">
         <name>$continent/name/text()</name>
         <area>$continent/area/text()</area>
      </continent>
   }
   {
      for $organization in table("organization")
      return
      <organization id="{}" headq="{$organization/city/text()}">
         <name>{$organization/name/text()}</name>
         <abbrev>{$organization/abbreviation/text()}</abbrev>
         <established>{$organization/established/text()}</established>
         {
            for $ismember in table("is_member")
            where $ismember/organization=$organization/abbreviation
            return
            (: this will produce one members element for each country that is member of the current organization :)
            (: the actual view groups all countries in a single element :)
            <members type="{$ismember/type/text()}" country="{$ismember/country}"/>
         }
      </organization>
   }
   {
      for $mountain in table("mountain")
      return
      <mountain id="{$mountain/name/text()}" country="">
          {
            for $geomountain in table("geo_mountain")
            where $geomountain/mountain=$mountain/name
            return
            (: this will produce one located element for each province :)
            (: the actual view groups all provinces in a single element :)
            <located country="{$geomountain/country/text()}"  province="{$geomountain/province/text()}"/>
          }
          <name>{$mountain/name/text()}</name>
          (: the source of these attributes is object-relational :)
          <longitude></longitude>
          <latitude></latitude>
          (: -- :)
          <height>{$mountain/height/text()}</height>
      </mountain>
   }
   {
      for $desert in table("desert")
      where $desert/name=$geodesert/desert
      return
      <desert id="{$desert/name/text()}" country=""
              climate="" temperature="" ground="">
         {
            for $geodesert in table("geo_desert")
            where $geodesert/desert=$desert/name
            return
            (: this will produce one located element for each province :)
            (: the actual view groups all provinces in a single element :)
            <located country="{$geodesert/country/text()}"  province="{$geodesert/province/text()}"/>
         }
         <name>{$desert/name/text()}</name>
         <area>{$desert/area/text()}</area>
      </desert>
   }
   {
      for $island in table("island")
      return
      <island id="{$island/name/text()}" country="" province="">
         <name>{$island/name/text()}</name>
         <area>{$island/area/text()}</area>
         (: the source of these attributes is object-relational :)
         <longitude></longitude>
         <latitude></latitude>
      </island>
   }
   {
      for $river in table("river")
      return
      <river id="{$river/name/text()}">
         <to watertype="" water="" />
         {
            for $georiver in table("geo_river")
            where $river/name=$georiver/river
            return
            (: this will produce one located element for each province :)
            (: the actual view groups all provinces in a single element :)
            <located country="{$georiver/country/text()}" province="{$georiver/province/text()}"/>
         }
         <length>{$river/length/text()}</length>
         <name>{$river/name/text()}e</name>
      </river>
   }
   {
      for $sea in table("sea")
      return
      <sea id="{$sea/name/text()}" country="">
         {
            for $geosea in table("geo_sea")
            where $sea/name=$geosea/sea
            return
            (: this will produce one located element for each province :)
            (: the actual view groups all provinces in a single element :)
            <located country="{$geosea/country/text()}" province="{$geosea/province/text()}"/>
         }
         <name>{$sea/name/text()}</name>
         <depth>{$sea/depth/text()}</depth>
         {
            $for $mergew1 in table("merges_with")
            where $mergew1/sea1=$sea/name
            return
            (: this will produce one bordering element for each border sea :)
            (: the actual view groups all seas in a single element :)
            <bordering>{$mergew1/sea2/text()}</bordering>
         }
         {
            $for $mergew2 in table("merges_with")
            where $mergew2/sea2=$sea/name
            return
            (: this will produce one bordering element for each border sea :)
            (: the actual view groups all seas in a single element :)
            <bordering>{$mergew2/sea1/text()}</bordering>
         }
      </sea>
   }
   {
      for $lake in table("lake")
      return
      <lake id="{$lake/name/text()}" country="">
         {
            for $geolake in table("geo_lake")
            where $lake/name=$geolake/lake
            return
```

```
            (: this will produce one located element for each province :)
            (: the actual view groups all provinces in a single element :)
            <located country="{$geolake/country/text()}" province="{$geolake/province/text()}"/>
         }
         <name>{$lake/name/text()}e</name>
         <area>{$lake/area/text()}e</name>
      </lake>

   }
</mondial>
```

APPENDIX D CONTRIBUIÇÕES

Este trabalho apresenta uma proposta para atualização de bancos de dados relacionais através de visões XML. A proposta utiliza um formalismo de definição de visões que denominamos *query trees.* As query trees capturaram noções de seleção, projeção, aninhamento, agrupamento e conjuntos heterogêneos, presentes na maioria das linguagens de consulta XML. Para permitir a atualização das visões XML, o trabalho demonstra como tais visões XML podem ser mapeadas para um conjunto de visões relacionais correspondentes. Consequentemente, esta tese transforma o problema de atualização de bancos de dados relacionais através de visões XML em um problema clássico de atualização de bancos de dados através de visões relacionais.

A partir daí, este trabalho mostra como atualizações na visão XML são mapeadas para atualizações sobre as visões relacionais correspondentes. Trabalhos existentes em atualização de visões relacionais podem então ser aplicados para determinar se as visões são atualizáveis com relação àquelas atualizações relacionais, e em caso afirmativo, traduzir as atualizações para o banco de dados relacional.

Como *query trees* são uma caracterização formal de consultas de definição de visões, elas não são adequadas para usuários finais. Diante disso, esta tese investiga como um subconjunto de XQuery pode ser usado como uma linguagem de definição das visões, e como as *query trees* podem ser usadas como uma representação intermediária para consultas definidas nesse subconjunto.

As principais contribuições deste trabalho são:

Um formalismo para especificação de visões XML. As *query trees* podem ser usadas como uma forma intermediária de representação de linguagens de definição de visões XML sobre bancos de dados relacionais (BRAGANHOLO; DAVIDSON; HEUSER, 2004a), o que torna a abordagem apresentada neste trabalho independente de sintaxe. Qualquer linguagem que possa ser mapeada para *query trees* pode ser utilizada para especificar as visões.

O trabalho apresenta uma avaliação do poder de expressão das *query tree*, onde se mostra que as query trees são suficientemente expressivas para ser utilizadas na prática.

Mapeamento de visões XML para visões relacionais. Dada uma visão XML especificada por uma *query tree*, este trabalho propõe algoritmos para mapeá-la para um conjunto de visões relacionais correspondentes. Também são apresentados algoritmos para mapear as atualizações sobre as visões XML para atualizações correspondentes sobre as visões relacionais (BRAGANHOLO; DAVIDSON; HEUSER, 2004a). Portanto, este trabalho transforma um problema em aberto – o da atualização através de visões XML – em um problema já bastante estudado – o de atualizações através de visões relacionais.

Um estudo da atualizabilidade de visões XML. Foi feito um estudo da atualizabilidade de visões XML construídas com as *query trees*, baseado em um estudo preliminar (BRAGANHOLO; DAVIDSON; HEUSER, 2003a) que utilizava a álgebra relacional aninhada para definir as visões. O estudo apresentado neste trabalho se baseia na idéia de ausência de efeitos colaterais de (DAYAL; BERNSTEIN, 1982a), e utiliza a teoria de tal trabalho para identificar classes de visões atualizáveis. Foram identificadas três classes de visões: (i) uma que é atualizável para todas as possíveis inserções; (ii) uma que é atualizável para todas as possíveis inserções, exclusões e modificações; e (iii) uma cuja atualizabilidade pode ser estudada utilizando o Teorema 6.1.

Um subconjunto de XQuery para especificar visões XML. Este trabalho propõe e implementa um subconjunto de XQuery que é capaz de construir visões XML sobre bancos de dados relacionais (BRAGANHOLO; DAVIDSON; HEUSER, 2003b). A UXQuery utiliza as *query trees* como uma representação intermediária para mapear a visão XML resultante para visões relacionais.

PATAXÓ. As idéias deste trabalho foram implementadas em um sistema denominado Pataxó. O Pataxó utiliza a UXQuery como a linguagem de definição das visões, e a abordagem de (DAYAL; BERNSTEIN, 1982a) para traduzir as atualizações das visões relacionais para o banco de dados.