Updating XML Views Of Relational Data

by

Mukesh Mulchandani

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2003

APPROVED:

Professor Elke Rundensteiner, Major Thesis Advisor

Professor George Heineman, Thesis Reader

Professor Micha Hofri, Head of Department

Abstract

XML has emerged as the standard data format for Internet-based business applications. In many bussiness settings, a relational database management system(RDBMS) will serve as the storage manager for data from XML documents. In such a system, once the XML data is shredded and loaded into the storage system, XML queries posed against these (now virtual) XML documents are processed by translating them as much as possible into SQL queries against the underlying relational storage. Clearly, in order to support full database functionalities over XML data, we must allow users not only to query but also to specify updates on XML documents. Today while the XML query language XQuery is being standardized by W3C, no syntax for updating XML documents is included in this language proposal as of now.

In this thesis, we have developed techniques for supporting translation of XML updates on XML views of relational data into SQL updates on the underlying relations. These techniques are based on techniques for supporting translation of updates on object-based views of relational data into SQL updates on underlying relations [TBW91]. The system has been implemented as a part of XML Management System, called Rainbow, that is being developed at the Worcester Polytechnic Institute (WPI). We have used XQuery as XML query language and Oracle as the backend relational store for implementation of the system. Experimental studies show that incremental XML updates supported by our system is a better choice than complete reload of XML documents under a variety of system settings.

Acknowledgements

I would like to express my gratitute to my advisor Prof. Elke Rundensteiner for constantly guiding me during my work. I am thankful to her for giving enough consideration to my ideas and views and for her continous support in my research.

I am also thankful to Prof. George Heineman for being reader of my thesis and for his input on improving my work. I also extend my special thanks to Xin Zhang for his continous support on techincal issues and to other members of Database Systems Research Group for helping me in various way during my collaboration with them. I thank all my friends and colleagues at Worcester Polytechnic Institute for making my stay here a memorable moment of my life.

I dedicate this work to my parents and family members who made me capable of achieving such heights and to my girl friend whose unconditional love and moral support helped me throughout my stay at WPI.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Over the past few years, there has been a tremendous surge in the interest in XML as a universal queryable representation of data. This has in part been boosted by the growth of web and e-commerce applications in the context of which XML has emerged as the de-facto standard for information interchange [W3C98b]. Today nearly every major vendor of a data management tool, be it Oracle [Ora] or IBM [IBM], has added support for importing, storing and viewing XML data over their relational engine. XML publishing capabilities, that is, the use of XML for formatting the result of SQL queries, have been added to the latest releases of relational database systems by Oracle [Ora], IBM [IBM] and Microsoft [Mic]. At the same time, due to the maturity of query optimization techniques and the high query performance offered by Relational Database query engines, use of Relational Database Management Systems (RDBMS) as a store for XML data has been put forth as a promising direction for XML data management.

Concurrently, considerable research has been carried out to show different approaches for mapping XML data into the relational data model [DFS99, FK99, SHT+99, ZMLR01]. All of these approaches propose different algorithms for shred-

ding XML documents so to be able to store them in relational tables. The efficiency of the re-construction of the XML documents from data stored in relations as well as XML query processing in general depends in part upon the level of shredding of the XML documents introduced by the mapping. Thus the choice of an effective and flexible mapping to store XML documents in relations has been recognized as an important issue. The issue of translating XML queries into SQL statements and reconstructing XML query results is now beginning to also be tackled [CKS+00, FTS00]. Ultimately it is expected that the relational database engines will provide standardized and integrated support for querying and publishing XML views of databases using a unified model, namely XML views specified by a standard XML query language such as XQuery [W3C01]. The XML query language working group of The World Wide Web Consortium (W3C) is dedicated to standardizing this query language and its capabilities.

The next immediate step to offering full-fledged XML-based database support is to support not only queries but also updates over XML documents, irrespective of the underlying storage medium chosen to manage the XML data. This now is the focus of this thesis work.

1.2 State of the Art of XML Management Systems

While there has been a lot of work to solve the problem of mapping XML documents into and out of relational databases, less research has been done to date on supporting queries or updates on XML documents stored in a relational database. For XML views of relational data, work has appeared in the literature on querying XML views of relational data [SKS+01]. [SKS+01] proposes a method to translate XML queries to SQL in order to evaluate XML queries against XML data stored in

relational database but doesn't support any kind of updates that a user might want to express over the XML data.

Little work has been published on updating XML views of relational data [ZMLR01, TIHW01]. [ZMLR01] proposes update primitives and their implementation, which can be used to extend an XML query language, provided there is a way to break down the update queries into the update primitives. [TIHW01] specifies extensions to the XML query language XQuery to support data updates over XML documents. [TIHW01] proposes different trigger-based solutions to update the XML data stored in relational databases. They assume however that they already have the knowledge of which tables are to be updated and how and also how other tables that might be affected as a result of these updates. They do not focus on the issue of how to translate XML updates through XML views into the correct SQL updates, which is an essential part for executing XML updates against XML data stored in relational databases and which is addressed by this thesis work.

A few XML algebra have been proposed till date [ZR02, GVD+01]. Both [ZR02] and [GVD+01] introduce an algebraic framework for expressing and evaluating queries over XML data. They also define equivalence rules for algebra rewrites, which can be used for the optimization of queries expressed over XML data. [GVD+01] does not focus on the issue of querying XML views of the relational data whereas Rainbow system [ZMC+02] does.

This thesis work offers update support for XML (virtual) views of relational data. It exploits the knowledge available in the form of the mapping query, of how a particular XML view was constructed, to translate any XML updates on the view to updates on the underlying relations. For this translation, this thesis work employs XML algebra and various XML algebra rewrite rules along with other decomposition strategies to decompose an XML update into updates on individual

relations storing the XML data to be executed against the relational database. Our approach of updating XML views of relational data is based on Keller's approach [TBW91] of updating object-based views of relational data.

1.3 Research Issues

Since XML documents are semi-structured and have hierarchical relationships between the elements of the documents, updating of XML views is not straightforward. The complex structure of the XML views build on top of flat relational data poses several issues to be addressed for the translation of XML updates into updates on underlying relational tables. Below are the issues that need to addressed to solve the problem of updating XML views :

1. **XML query language to support XML updates:** The XML query language standard by World Wide Web Consortium(W3C) - XQuery [W3C01] does not have support for XML updates. Also, at the time we started this work, there was no other XML query language that can support XML updates yet. Hence as first step towards addressing the problem of XML updates, we have designed language support for issuing XML updates on XML documents. [TIHW01] has proposed extensions to XQuery for XML updates support but there is no prototype of a system available that supports updates on XML documents to date.

2. **Which XML views are updatable:** Due to many XML model characteristics of XML documents, XML views fall in a different category than traditional relational views. XML views can perhaps more closely be compared to object views of relational data [Bar90]. Not all XML views can be updated. A clear distinction needs to established between views that can be updated and those that cannot be updated un-ambiguously. Restrictions on

defining an XML view that would qualify this view as an updatable XML view need to be identified.

3. **Determining impact on underlying data storage:** Given a view definition in XQuery, how can we extract any information about the underlying relations that may be relevant for update propagation? Typically, this view query should be analyzed to determine the possible impact of any future update query on the underlying data relations. This knowledge of the underlying relations extracted from the view definition and stored once for each view then can help in the decomposition and translation of XML updates into SQL updates on the underlying relational data.

4. **Decomposing XML update query:** Elements in an XML view can be complex, possibly nested several levels deep. To carry out our update on these XML views having complex elements, each XML update needs to decomposed into updates against individual relations underneath the XML view. One needs to know what it takes to be able to decompose an XML update into SQL updates on the underlying relations and whether all the required information can be extracted from the XML view definition and the XML update, both expressed in the XML query language.

5. **Translation of relational updates:** Decomposition of an XML update into updates on the underlying relations will merely give rise to the same kind of updates on the underlying relations (e.g., a Delete on a view decomposes into Delete operations on the underlying relations). But carrying out the same update may not neccessarily result in the desired effect intended by the XML view update. An update on a relational table underneath the view is *extraneous* if the update on the view can be realised without the extraneous update. Minimum changes (without any extraneous updates) should be done to the underlying relations to achieve the effect intended by the view

update. Also taking global integrity of the database into consideration, depending upon the interrelationships between the underlying relations, an XML *Delete* operation might get translated into a relational *Replace* or *Insert* on certain relations. Thus correct translation of relational updates is important. Work has appeared in the literature on correct translation of updates [Kel85, Kel86, A. 86, DB82, Mas84, TBW91].

6. **Performance:** Performance of such an approach from the point of view of time, storage requirements and other factors should be accessed. Also a performance comparison of the proposed approach with other approaches such as the trigger based approach of carrying out updates on the relational system as discussed in [TIHW01] should be made.

1.4 Our Approach

The problem of updating XML views is similar to the traditional problem of updating relational views of relational data and can perhaps more closely be compared to the problem of updating object views of relational data. The problem of updating views was addressed in the relational context [DB82, Mas84, Kel85, Kel86, A. 86]. Updating of XML views comes with new challenges beyond those of relational views since it has to address the mismatch between the two distinct data models (the XML flexible hierarchical view model and the flat relational base model) and between the two query and update languages (XQuery FLWU updates on the XML view versus SQL queries on the base). Issues of updating object views of relational data have been addressed in the literature[TBW91]. This work also states limitations on the definition of views that are updatable. These limitations on views help in translating an update on a view, without ambiguity, into updates on the underlying relations. Though the solution to updating object views of relational data takes into consideration the hierarchy of complex objects defined on top of relational data, it

does neither deal with an inter-object hierarchy nor with the ordering of siblings as found in XML views.

Figure 1.1 shows the approach we have adopted for carrying out the updates on XML views of relational data. As shown in the figure there are 8 different steps that are performed to propagate a given XML update to the underlying relations. The grey boxes shown in Figure 1.1 are the steps that are performed individually on the view query and the update query except that *Analysis of View XAT* is only performed on the view query. Only during this step all semantic information about the underlying schema such as the names of the underlying relations, keys, relationships among the underlying relations, etc. is gathered and stored in our system for use in later steps of the process. For the view query, these steps are performed only once at the time of the definition of the view. As opposed to the view query that goes through the initial steps only at the time of view definition, each update query goes through all steps of the approach except for *Analysis of View XAT* step. Below we discuss in brief the steps of the approach.

During the first step, called *The Generation of XATs*, the XML algebra tree of the query is generated for the view query and update query. Then the two XATs are decorrelated and optimized. After decorrelation the two trees are merged during the *Merging of the Two XATs* step so that the update can be issued against the view constructed over the relational data. During the *Computation Pushdown* step the merged XAT is rewritten using several rewrite rules in order to push all SQL computation down to the bottom of the tree. The XML update is then decomposed into its relational update counterparts on the underlying relations. This is done during the *Decomposition of the XML Update* step. Then translation of those updates is carried out to give rise to the final updates that will be carried out on the corresponding relations. Semantic information is gathered during the analysis of the view step to be used later steps of update translation. Finally these translated

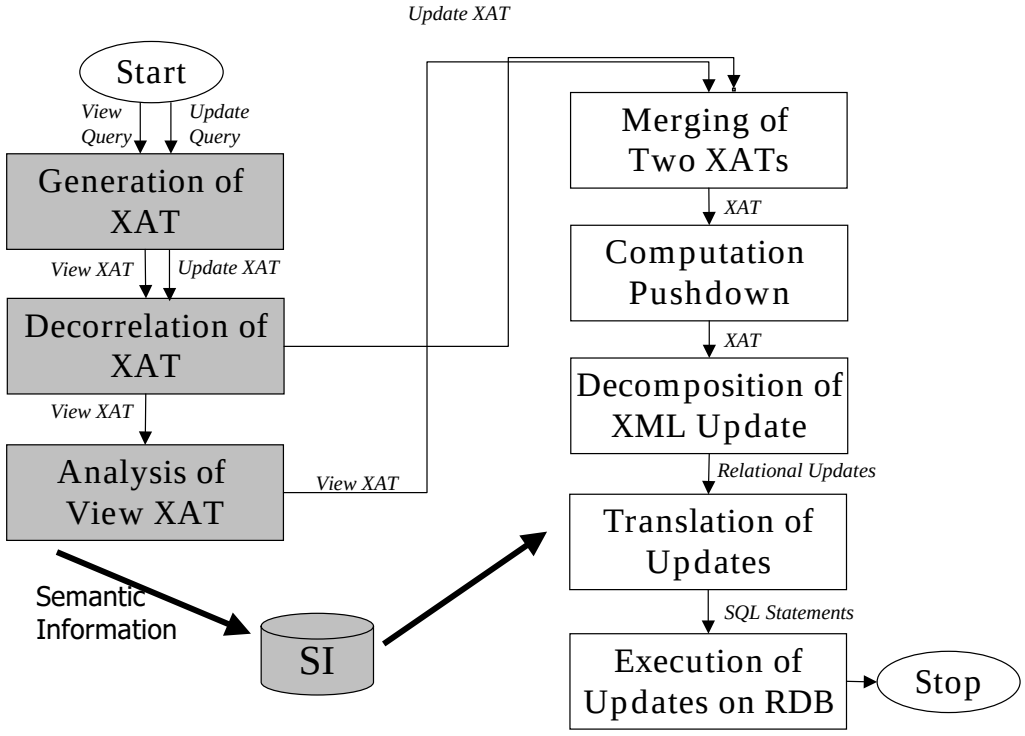

Figure 1.1: Approach

updates are executed on the underlying relations so that the intended effect of the update on the view has indeed been realised when the view is reconstructed from the modified relational data.

1.5 Contributions of this Thesis

Below is the list of contributions of my thesis work :

1. XQuery support for XML updates, including XQuery language extensions and Rainbow XML algebra extensions.

2. Framework for correct propagation of XML updates to the underlying relations through XML views expressed by XQuery.

    - Application of object-based view updating strategy to XML views.
    - First solution for updating XML views.

3. Implementation of the system as a proof of concept and incorporation into the XML Data Management System, called Rainbow.

4. Experimental evaluation to support my solution.

Chapter 2

Background

We now review the technical background needed for this work, in particular, the XML and DTD data model and XML query language. We also introduce a language extension for XQuery to support XML updates, and briefly introduce the concept of XML views of relational data.

2.1 XML and XML Schemas

XML (Extensible Markup Language) [W3C98b] is currently used both for defining document markup (and thus information modeling) and for data exchange. XML documents are composed of character data and nested tags used to document the semantics of the embedded text. Tags can be used freely in an XML document (as long as their use conforms to XML specification) or can be used in accordance with the *document type definitions* (DTDs) [W3C98a] or *XML Schema* [W3Cb] which define the types for a class of documents. An XML document that conforms to a DTD or XML Schema is called a valid XML document.

A DTD or XML Schema is used to define the allowable structure of elements (i.e., it defines allowable tags and tag structure) in a valid XML document. A DTD can

include four kinds of declarations: *element type, attribute-list, notation* and *entity*. An element type declaration is analogous to a data type definition; it names an element and defines the allowable content and structure. An element may contain only other elements (called element content) or may contain any mix of other elements and text, which is represented as PCDATA (called mixed content). An EMPTY element type declaration is used to name an element type without content (it can be used for example to define a placeholder for attributes). Finally an element type can be declared with content ANY meaning the type (content and structure) of the element is arbitrary.

Attribute-list declarations define the attributes of an element type. The declaration includes attribute names, default values and types, such as CDATA, NOTATION, ENTITY, etc. Two special types of attributes ID and IDREF are used to define references between elements. An ID attribute is used to uniquely identify an element; an IDREF attribute can be used to reference that element, and an IDREFS attribute can refer to multiple elements.

XML Schema [W3Cb], which is also used to define the allowable structure of elements for a given application or application domain in a valid XML document uses XML document syntax. Declarations in XML Schema can have richer and more complex internal structures than declarations in DTDs. Schema designers can take advantage of XML's containment hierarchies to add extra information where appropriate. XML Schemas also provide an improved data typing system. They provide *data-oriented* data types in addition to the more *document-oriented* data types that XML 1.0 DTDs [W3Ca] support, making XML more suitable for data interchange applications. Built-in data types include strings, booleans, and time values. The XML Schema draft [W3Cb] provides a mechanism for generating additional data types. Besides, XML Schema supports namespaces and the notion

of keys to uniquely identify elements in an XML document. More information on XML Schemas can be found in [W3Cb].

2.2 Running Example

Figure 2.1 depicts a running example of a DTD and a conforming XML document of a simple online book store application. As we can see, one *bib* element contains several *books* and each *book* in turn contains a *title*, one or more *authors* or *editors*, a *publisher* and a *price* element. Each book also has a *year* attribute which is required. The element *author* is composed of *first* and *last* elements which are of PCDATA type whereas each *editor* element is composed of *first, last* and *affiliation*. The later three again are of PCDATA type. The XML document in Figure 2.1 contains two books from the year 1994 and the year 2000, having one and three authors respectively.

2.3 The XQuery Language

XQuery [W3C01] uses the type system of XML Schema and is a W3C working draft by W3C XML: Query Working Group (http://www.w3.org). XQuery is a query language designed to query XML data and is derived from Quilt [CRF00] which is in turn derived from Xpath [W3C00], XML-QL [DFF+99], SQL, OQL, Lorel [AQM+97] and YATL [CJS99].

Example 1 *The query for getting the title of all the books published in or before the year 2000 and the total number of such books in the bibliograpy document bib.xml corresponds to the XQuery expression shown in Figure 2.2.*

From Figure 2.2, we can see that XQuery expressions are *FLWR expressions*. A FLWR expression is composed of FOR, LET, WHERE and RETURN clauses. A

```
<?xml version="1.0"?>
<!DOCTYPE bib [
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  (author+ | editor+ ), publisher, price )>
<!ATTLIST book  year CDATA  #REQUIRED >
<!ELEMENT author  (last, first )>
<!ELEMENT editor  (last, first, affiliation )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
]>


<bib>
   <book year="1994">
      <title>TCP/IP Illustrated</title>
      <author><last>Stevens</last><first>W.</first></author>
      <publisher>Addison-Wesley</publisher>
      <price> 65.95</price>
   </book>
   <book year="2000">
      <title>Data on the Web</title>
      <author><last>Abiteboul</last><first>Serge</first></author>
      <author><last>Buneman</last><first>Peter</first></author>
      <author><last>Suciu</last><first>Dan</first></author>
      <publisher>Morgan Kaufmann Publishers</publisher>
      <price> 39.95</price>
   </book>
</bib>
```

Figure 2.1: Example of DTD and XML Document

FOR-clause is used for iteration. Each FOR iteration will bind one or more variables, e.g., **$book**. A LET-clause will bind variables without iteration, e.g., **$titles** as shown in Figure 2.2. The variables bound in the FOR and LET clauses will be filtered by the predicates specified in the WHERE clause. For example, the WHERE clause **"$book/@year <= 2000"** in the XQuery expression in Figure 2.2 removes all the bindings from the **$book** set that do not have an attribute *year* with the value less than or equal to 2000. The RETURN clause generates the output of the FLWR expression. For example the xpath **$book/title** in our example will generate a *title* element for the title of each book.

A FLWR expression uses the abbreviated syntax of XPath [W3C00] for navi-

```
<books>
  FOR $book IN document("bib.xml")/book
  LET $titles = $book/title
  WHERE $book/@year <= 2000
  RETURN
       $book/title
       <total>count($titles)</total>
</books>
```

Figure 2.2: XQuery Expression over XML Document in Figure 2.1

```
<books>
    <title>TCP/IP Illustrated</title>
    <title>Data on the Web</title>
    <total>1</total>
</books>
```

Figure 2.3: Result of XQuery Expression in Figure 2.2

gating through the document. For example **$book/title** in Figure 2.2 is an XPath expression. An element constructor in a RETURN clause consists of a start tag and an end tag. For example, <**books**>**.....**</**books**> in Figure 2.2 will wrap the query result into a new XML element. The expected result when the XQuery expression in Figure 2.2 is executed over the XML document in Figure 2.1 is depicted in Figure 2.3.

2.4 XML Views Of Relational Data

One important use of an XML query language is to wrap data stored in relational databases using an XML wrapper. There can be several ways in which this access might be accomplished but the most obvious is allowing this access through XML views. XML views can be defined on top of relational data. These views can then be queried using some XML query language such as XQuery [W3C01]. In this section we discuss this mechanism of defining XML views on top of relational data as adapted in XPERANTO [CKS+00] and also in Rainbow [ZR01].

The basic XML view is called a *Default XML View* in which each table in the relational database is represented as one XML document. One way to represent a database table as an XML document is to let the table be represented as a docu-

ment element, and each row (tuple) inside the table to be represented by a nested sub-element also called a tuple-element. Inside the tuple-elements, each attribute value of this type is in turn represented by a nested sub-element labeled by the name of the respective column.

Representing each table as an XML document will give rise to several documents which, at times, might not be desirable. Hence several tables can be put together into a single XML document to represent the complete database as shown in Figure 2.5. As we can see from Figures 2.4 and 2.5 each table forms a child element of the main *DB* element rather than forming an XML document of its own. Figure 2.4 shows three relational tables :

- Books(bookid, title, year)
- Authors(first, last, bookid)
- Prices(title, source, price)

Figure 2.4 also shows the default XML view of the three tables. In real world applications the user might not be interested in knowing the source and particular storage structure of the XML documents (in this case the relational tables). Hence a default XML view which explicitly models the fact of how many tables exist in the database would be confusing to an end user. Hence in order to provide further levels of abstraction, it is advisable to define a user-specific view on top of this rigorous default XML view of relational data. The latter can in some sense be viewed as a physical XML Schema. In XML management systems such as Rainbow [ZR01] that provide a bridge between XML and relational technologies these XML views are called virtual views. Users interact with these views as if they were working with a pure XML system. Hence the fact that these views are XML views of relational data should be as transparent to the end user as possible. Such virtual views can

be defined on top of the *Default XML View* using XQuery expressions, then called *Mapping Queries* (queries that map relational data to XML views). Figure 2.7 illustrates one such virtual view. Figure 2.6 shows the mapping query that defines the virtual view on top of the default XML view from Figure 2.5.

Books

Bookid	Title	Year
1	Data Structures	2000
2	JAVA Complete Reference	1999

Authors

First	Last	BookId
Michael	Savitch	1
Peter	Naughton	2

Prices

Title	Source	Price
Data Structures	Amazon.com	54.00
JAVA Complete Reference	Ebay.com	65.00

Figure 2.4: Relational Tables in Database

```
<DB>
  <books>
        <row>
          <bookid>1</bookid>
          <title>Data Structures</title>
          <year>2000</year>
        </row>
        ……..
  </books>
  <authors>
        <row>
          <first>Michael</first>
          <last>Savitch</last>
          <bookid>1</bookid>
        </row>
        ……..
  </authors>
  <prices>
        <row>
          <title>Data Structures</title>
          <source>Amazon.com</source>
          <price>54.00</price>
        </row>
        ……..
  </prices>
</DB>
```

Figure 2.5: Default View of Relations Shown in Figure 2.4

2.5 XQuery Updates

In this section we describe an extension to the XQuery language syntax to support XML updates. We follow the basic set of operations proposed in [TIHW01]. Below we discuss language extensions using the grammar in general. We also give specific examples to show use of language extensions for specifying updates. As proposed in [TIHW01]. we extend XQuery with a **FOR...LET....WHERE.....UPDATE**

Mapping Query

```
<books>
FOR $book IN document("default.xml")/books/Row,
    $author IN document("default.xml")/authors/Row,
    $price IN document("default.xml")/prices/Row
WHERE $book/bookid = $author/bookid
        AND $price/book_title = $book/title
RETURN
        <book>
          $book/bookid,
          $book/title,
          $book/year,
          $price/source,
          $price/price,
          <author>
              $author/first,
              $author/last
          </author>
        </book>
</books>
```

Figure 2.6: Mapping Query Issued on XML View Shown in Figure 2.5

Virtual View

```
<books>
  <book>
        <bookid>1</bookid>
        <title>Data Structures</title>
        <year>2000</year>
        <source>Amazon.com</source>
        <price>54.00</price>
        <author>
          <first>Michael</first>
          <last>Savitch</last>
        </author>
  </book>
  <book>
        <bookid>2</bookid>
        <title>Java Complete Reference</title>
        <year>1999</year>
        <source>Ebay.com</source>
        <price>65.00</price>
        <author>
          <first>Peter</first>
          <last>Naughton</last>
        </author>
  </book>
</books>
```

Figure 2.7: Mapping View as a Result of Mapping Query Shown in Figure 2.6

structure of updates as shown in Figure 2.8. Within the **UPDATE** clause, a sequence of sub-operations are specified.

As shown in Figure 2.8, the nested **FOR....WHERE** clause allows one to specify an XPath expression to be matched and to be bound to *$binding*, as well as a set of predicates that may restrict this binding. A nested update operation may be performed over any of the bindings from the outer scope or the **FOR** clause. If multiple **updateOps** are specified, they are performed consecutively for each iteration of the variable binding. Figure 2.9 shows an example of the *Insert* update.

The **FLWR** expression in Figure 2.9 inserts a new *book* element into the (virtual) view shown in Figure 2.7. The **FOR** clause of the expression binds $root to the root of the document which is the *books* element. The **UPDATE** clause of the expression

FOR $*binding*1 **IN** *Xpath-expr*,…..
LET $*binding* := *Xpath-expr*,…
WHERE *predicate1*,…..
updateOp,……

Where **updateOp** is defined in EBNF as :

UPDATE $*binding* {**subOp** {, **subOp**}* } and **subOp** is :

DELETE $*child* |
RENAME $*child* To new_name |
INSERT ($*bind* [**BEFORE** | **AFTER** $*child*]
| **new_attribute**(name, value)
| **new_ref**(name, value)
| content [**BEFORE** | **AFTER** $*child*]) |
REPLACE $*child* **WITH** (**new_attribute**(name, value)
| **new_ref**(name, value)
| content) |
FOR $*sub_binding* **IN** *Xpath-subexpr*,…..
WHERE *predicate1*,……. **updateOp**.

Figure 2.8: XQuery Language Extensions to Support XML Updates.

specifies that the **$root** is to be updated by inserting a new **book** element that is created within the **INSERT** clause. The result we expect to see when this XQuery update is issued on the virtual XML view in Figure 2.7 is shown in Figure 2.10.

Insert Update

```
FOR $root IN document("books.xml")
UPDATE $root{
  INSERT <book>
            <bookid>"3"</bookid>,
            <title>"My New Book"</title>,
            <year>"2003"</year>,
            <source>"Ebay.com"</source>,
            <price>"54.50"</price>,
            <author>
              <first>"John"</first>,
              <last>"Doe"</last>
            </author>
          </book>
}
```

Figure 2.9: Insert Update Issued on XML View Shown in Figure 2.7

Virtual View

```
<books>
  <book>
        <bookid>1</bookid>
        <title>Data Structures</title>
        <year>2000</year>
        <source>Amazon.com</source>
        <price>54.00</price>
        <author>
          <first>Michael</first>
          <last>Savitch</last>
        </author>
  </book>
  <book>
        <bookid>2</bookid>
        <title>Java Complete Reference</title>
        <year>1999</year>
        <source>Ebay.com</source>
        <price>65.00</price>
        <author>
          <first>Peter</first>
          <last>Naughton</last>
        </author>
  </book>
  <book>
        <bookid>3</bookid>,
        <title>My New Book</title>,
        <year>2003</year>,
        <source>Ebay.com</source>,
        <price>54.50</price>,
        <author>
          <first>John</first>,
          <last>Doe</last>
        </author>
  </book>
</books>
```

Figure 2.10: Result of Insert Update Shown in Figure 2.9

Chapter 3

Rainbow System

Rainbow is an XML Management System being developed at Worcester Polytechnic Institute (WPI). *Rainbow* has been designed for storage, retrieval, and querying of XML documents based on relational database technology. The goal is to equip *Rainbow* with a flexible mapping mechanism that not only allows for a wide variety of mapping XML documents into appropriate relational schemata, but also explicitly models and manages the chosen mapping via a metadata model [CR02]. This metadata driven mapping model then in turn can be exploited by *Rainbow* for accomplishing database management services such as loading, extracting, updating and querying. Figure 3.1 shows the complete architecture of the *Rainbow* system. The system uses strategy designed and implemented for XQuery to SQL query translation. It is being fully implemented using XQuery as XML query language and Oracle as backend relational store.

3.1 Rainbow Query Engine

The *Rainbow Query Engine* is the core part of the *Rainbow* system. All XML queries or XML updates are submitted to this module for execution. Below we discuss the architecture of the query engine, the XAT algebra used by the engine

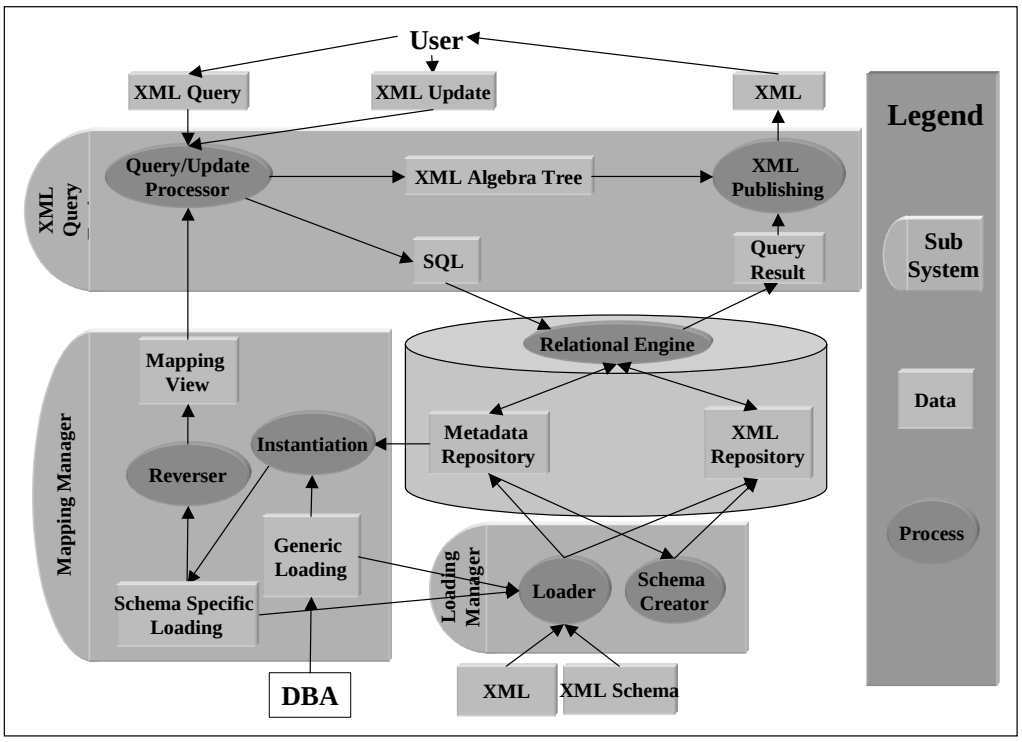

Figure 3.1: Architecture of Rainbow System.

and the XQuery-to-SQL translation mechanism adopted by the query engine for the execution of XQueries but not XML updates.

3.1.1 Architecture.

Figure 3.2 depicts the architecture of the *Rainbow Query Engine*. The architecture shown depicts the engine capable of only executing XQueries issued on the virtual views of relational data but not XML updates. The architecture of the Rainbow engine enhanced to support XML updates will be shown in following sections.

As shown in Figure 3.2, the query engine uses the *Kweelt Parser* [A. 00] to parse XQuery expressions. The parsed tree generated by the *Kweelt Parser* is fed to *XAT Generator* where the parsed tree is analysed and the *XML Algebra Tree* (XAT) for the XQuery expression is generated. XAT is generated for both the user query as well as the mapping query. The generated XATs are then given to *XAT Decorrelator* which decorrelates the given XATs to unnest XQuery expressions.

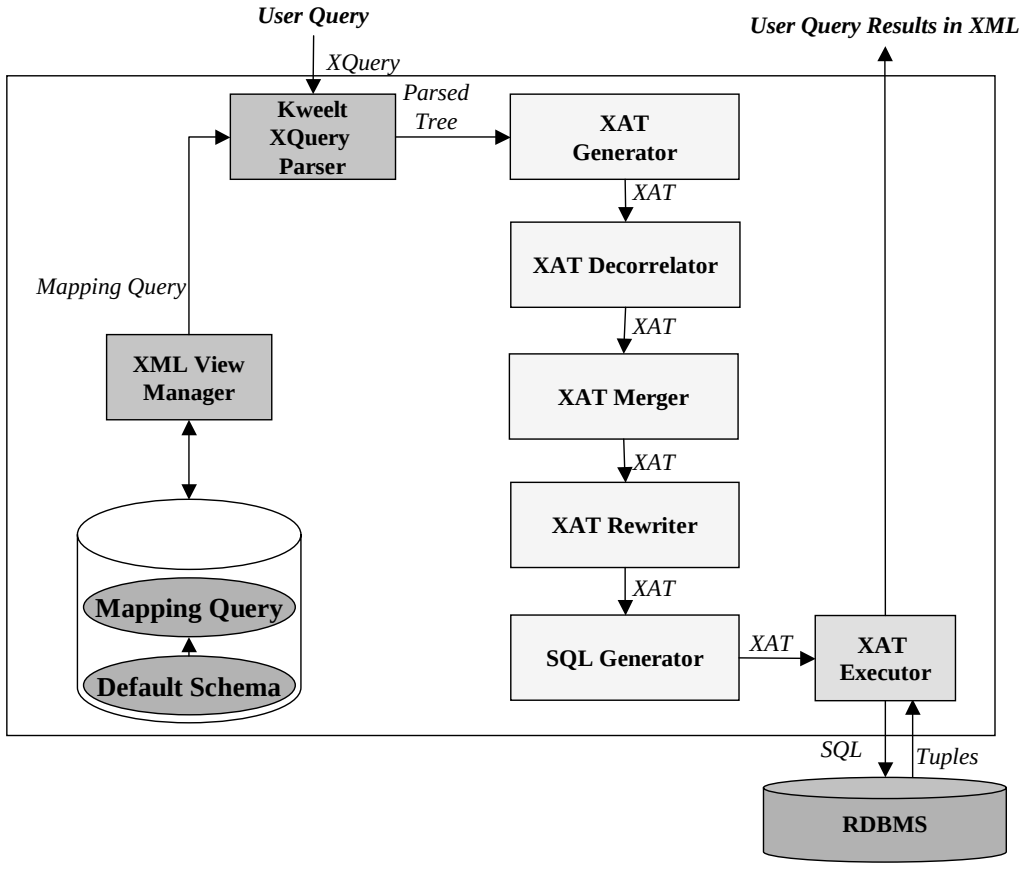

Figure 3.2: Architecture of Rainbow Query Engine.

After decorrelation, the two XATs are merged by the *XAT Merger* and the output is a final merged XAT that specifies the user XML query on top of the XML view construction. The merged tree is then sent to the *XAT Rewriter* for rewriting the XML Algebra Tree so as to push as much computation down to the bottom of the tree. This allows most of the computation to be pushed down into SQL while the top part of the XAT then contains XML specific construction operators. The *SQL Generator* generates SQL from the bottom portion of XAT. This SQL is then executed against the underlying relational database by the *XAT Executor*. The tuples returned by the SQL engine are then tagged into XML elements using construction information from XML specific operators at the top in XAT. The results in form of an XML document is then returned to the user.

3.1.2 XAT Operators

XML operators (as shown in Table 3.1) are used to represent the XML document related operations, e.g., navigation and construction. SQL operators (as shown in Table 3.3) correspond to the relational complete portion of the algebra. Special operators (as shown in Table 3.2) include operators used temporarily in different phases of optimization and operators shared by the class of XML and of SQL operators. Each table describes the operator's name, its symbol, its parameters, the notion of the output columns, and its sources of data and subqueries, with their description. We show only the relevant operators in this section. For the complete list of the XAT operators, please refer to [ZR02].

3.1.3 XQuery Translation

Rainbow's query engine uses the XML algebra, introduced in Section 3.1.2, for optimization and execution of queries. Below we briefly describe some of the steps invoved in the XQuery to SQL translation procedure which include generation,decorrelation, merging and rewriting of XATs. For more general discussion and walk through steps of XQuery translation please refer to [ZPR02a].

```
<prices>
    FOR $book IN document("dxv.xml")/book/row,
        $store IN document("dxv.xml")/store/row,
        $prices IN document("dxv.xml")/prices/row
    WHERE $book/bid = $prices/bid AND
          $source/sid = $prices/sid
    RETURN
        <book>
            $book/title,
            $store/source,
            $prices/price
        </book>
</prices>
```

Figure 3.3: XQuery for Constructing a Virtual View.

```
<result> {  FOR $t IN
            distinct( document("prices.xml")/book/title)
    RETURN
        <booktitle>
            $t/text ( )
        </booktitle>
</result>
```

Figure 3.4: XQuery to Query over View Shown in Figure 3.3.

Generation, Decorrelation and Merging of XATs Figure 3.3 depicts the view

query that is used to construct the XML view using the default XML view constructed from relational data. Whereas Figure 3.4 depicts the user query issued to query XML view for distinct book titles. A separate XAT is generated for each query using the XML algebra discussed in section 3.1.2. Figure 3.5 and figure 3.6 show the XATs generated from the view query and user query respectively. The two trees after decorrelation are shown in Figure 3.7 and Figure 3.8 and XAT after merging view XAT and query XAT is shown in Figure 3.9.

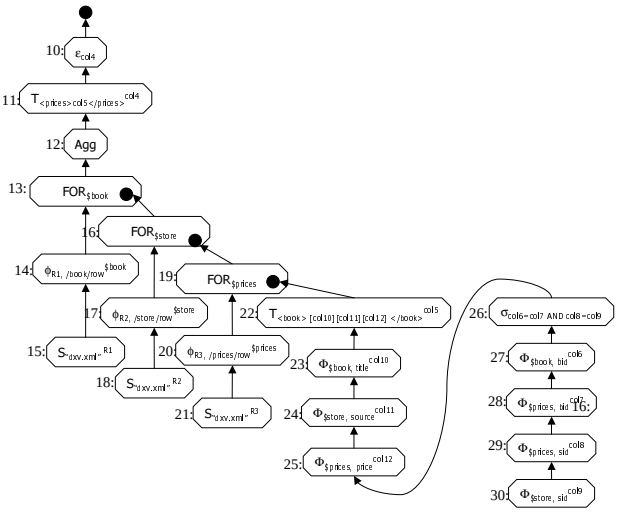

Figure 3.5: XAT for View Query Shown in figure 3.3.

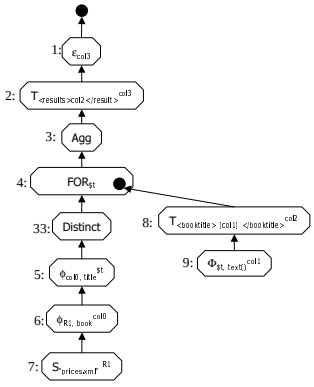

Figure 3.6: XAT for User Query Shown in Figure 3.4.

Rewriting of merged XAT Rewriting in Rainbow refers to the process of evaluating equivalence rules on an XML algebra tree to optimize the tree for efficiency and create a version of the tree where SQL queries can easily be generated. The administrator can choose from several possible rule application heuristics to optimize the tree, otherwise default heuristics are chosen. Below several commonly used rewrite rules in Rainbow are listed.

ϕ(x1, path):x2[ϕ(y1, path):y2[T]] $\rightarrow$ ϕ(y1, path):y2[ϕ(x1, path):x2[T]] (x1 != y2)
ϕ(x1, path):x2[σ(c)[T]]$\rightarrow$ σ(c)[ϕ(x1, path):x2[T]]
ϕ(x1, path):x2[$\times$[T1, T2]] $\rightarrow$ $\times$[T1, ϕ(x1, path):x2[t2]] (if x1 in DM(T2))
ϕ(x1, path):x2[$\times$[T1, T2]] $\rightarrow$ $\times$[ϕ(x1, path):x2[T1], T2] (if x1 in DM(T1))

These rules state that Navigate operators can be pushed through all operators until a dependency on its child operator arises.

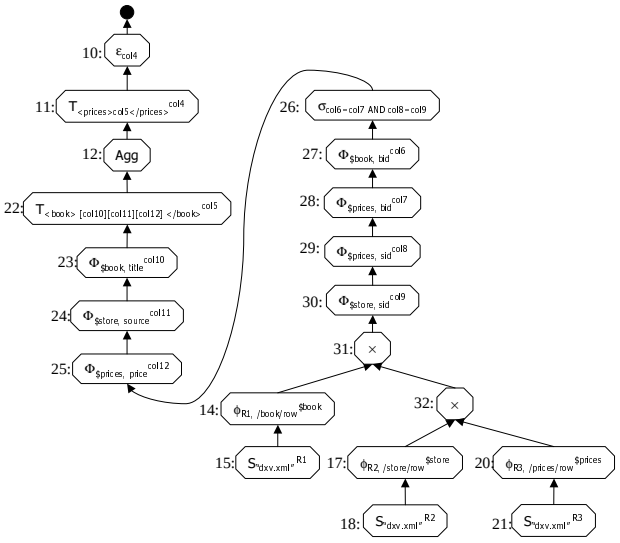

Figure 3.7: View XAT After Decorrelation.

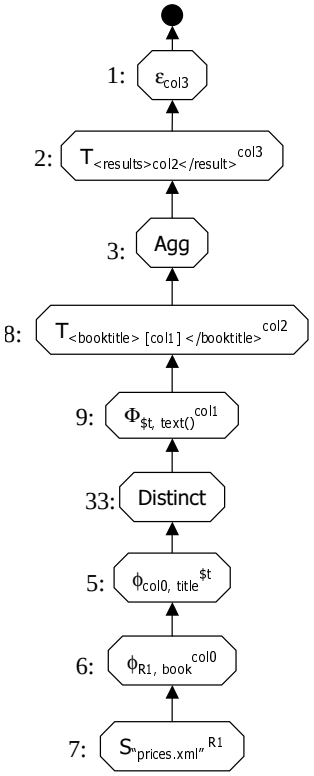

Figure 3.8: User XAT After Decorrelation.

Furthermore, during the push down process, redundant operators may be canceled out. As discussed earlier, several heuristics can be chosen to rewrite XAT into the normalised form where all XML operators are on the top and all SQL-doable are at the bottom. Rewrite heuristics include removal of the construction of intermediate XML fragments, canceling of duplicate navigate operators, computation pushdown, and propagating renames. The order in which the heuristics are applied can affect the resulting tree. Figure 3.10 shows rewritten XAT for our example. For detailed discussion of the rewrite heuristics of Rainbow please refer [ZPR02b].

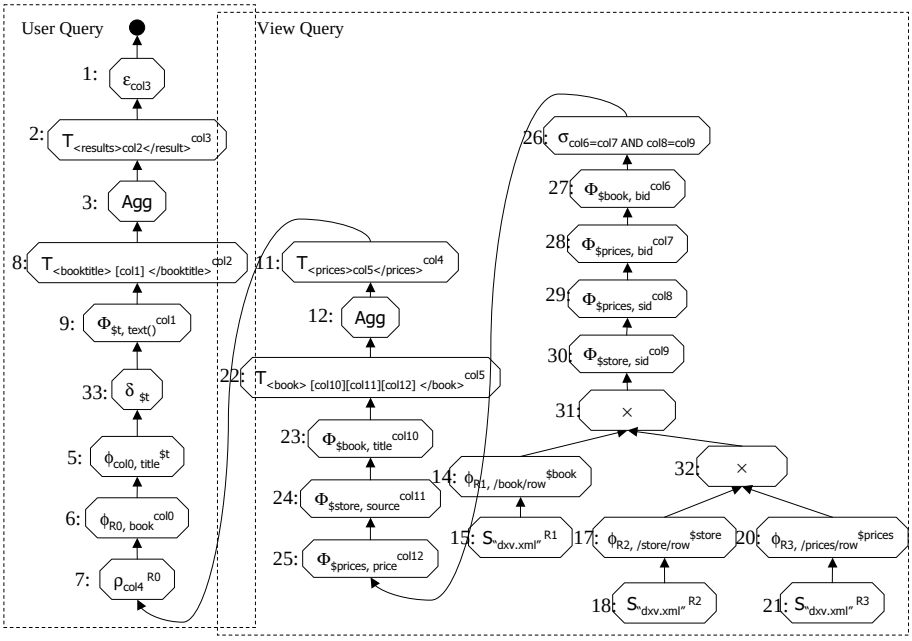

Figure 3.9: XAT as a Result of Merging XATs Shown in Figure 3.7 and Figure 3.8.

3.2 Rainbow Update Manager

As pointed out in earlier sections, the *Rainbow Update Manager* is the part of the *Rainbow Query Engine* responsible for handling any XML updates entering the system. The *Rainbow Update Manager* has been closely integrated with the *Rainbow Query Engine* by extending several modules of the query engine and adding other modules to extend query engines functionality to handle XML updates as well. Below we discuss the architecture of the modified *Rainbow Query Engine* with emphasis on modules specifically added to handle XML updates by this thesis. In the following subsections, we also discuss the update-specific XAT algebra for the *Rainbow* system and briefly introduce the overall update translation process.

3.2.1 Architecture of Enhanced Rainbow Query Engine

Figure 3.11 depicts the architecture of the modified *Rainbow Query Engine*. The architecture shown is of the engine now capable of executing an XQuery expression used for querying and also updating virtual views of relational data.

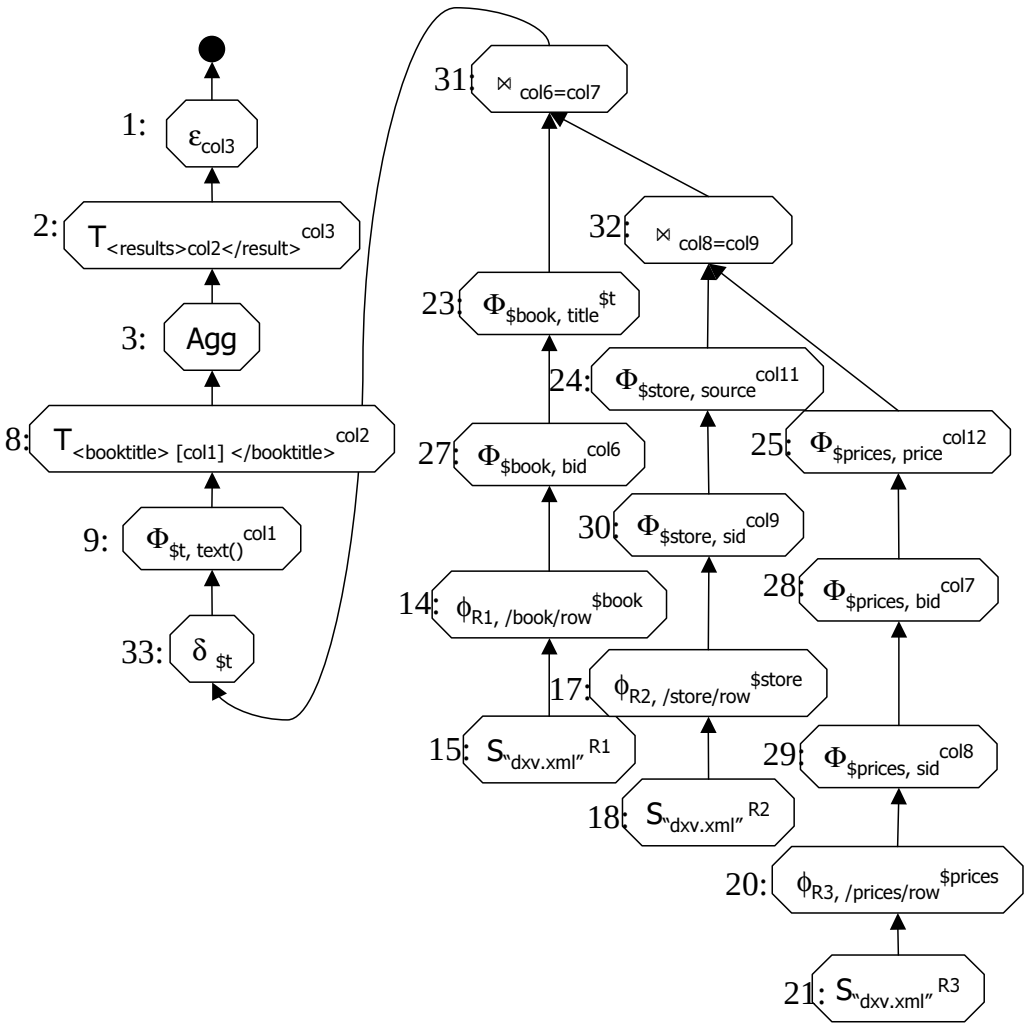

Figure 3.10: XAT as a Result of Rewriting of XAT Shown in Figure 3.9 using Rewrite Rules Discussed in [ZPR02b].

As shown in Figure 3.11, the *Update Manager* still uses most of the previous components of the query engine for generation, decorrelation and merging of XATs as well as for rewriting of the merged XAT. The *Kweelt Parser* has been extended to parse XQuery update expressions and the output now is a parsed query tree or parsed update tree depending on whether the input is a query or an update expression. At the same time, the *XAT Generator* has been extended to generate XAT update nodes which will be discussed in following subsections. Thus an XQuery update expression will eventually lead to an XAT with update operators as the output of the *XAT Generator*. The XAT representing the mapping query, also called the

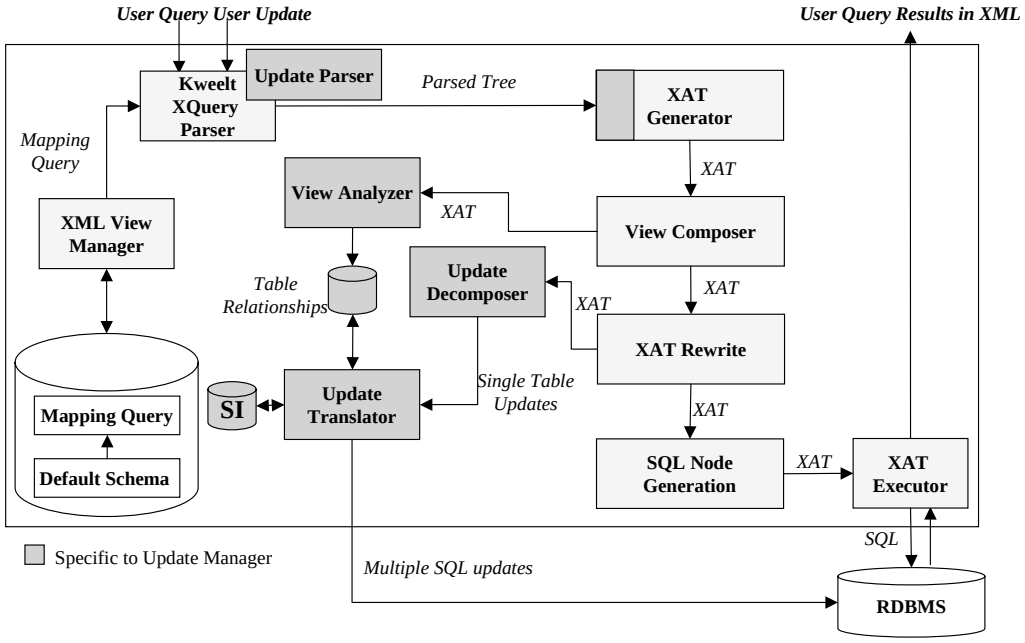

Figure 3.11: Architecture of Rainbow Query Engine with Update Manager.

view XAT, is generated as discussed earlier. This view XAT is then fed to the *View Analyzer* which processes this XAT to extract information about the underlying relations and relationships between the relations. The *View Analyser* is discussed in detail in Chapter 4. This kind of analysis is done only once for each view at view definition time and information gathered is stored in the system as shown in Figure 3.11. Once the XQuery update comes into the system, it goes through the process of generation, decorrelation and merging of XATs after which the merged tree is submitted to the *XAT Rewriter* for computation pushdown. After computation pushdown, the XAT is passed to the *Update Decomposer* which decomposes the XML update represented by the update tree into relational updates against individual base relations. These updates are then sent to the *Update Translator* which also uses the information gathered at view definition time and any other semantic information available to translate these updates using algorithms designed for translation of updates on view-objects of relational data [TBW91]. The *SQL Generator and Executor* then generates SQL updates and executes them against underlying relations in the relational database.

3.2.2 XAT Update Algebra

XML update operators as shown in Table 3.4 are used to represent the XML document related updates using Delete, Insert, Replace and Rename update primitives. As described earlier for operators in Table 3.1, the table describes the operator's name, its symbol, its parameters, the notion of output columns, and its sources of data and subqueries along with their description. The table shows the complete list of update operators supported by the *Update Manager* in the *Rainbow* system. *Delete* and *Rename* operations are simple and self explanatory with only one option of parameters whereas *Insert* and *Replace* operations are complex operations with several parameter options as indicated in Table 3.4. The effect of the update operations is based on the chosen input parameters to the operators.

Operator	*Sym*	*Prms.*	*Output*	*Data*	*Description*
Expose	ϵ	*col*	N/A	*s*	Expose column *col* as XML documents or fragments.
Tagger	T	*p*	*col*	*s*	Taggering *s* according to list pattern *p*.
Navigate	ϕ/Φ	$col1, path$	$col2$	*s*	Navigate into column *col* of *s* based on path *path*.
Aggregate	*Agg*	*col*	N/A	*s*	Make a collection for each column. *col* is the focus column name.
XML Union	$\overset{x}{\cup}$	*col*+	*col*	*s*	Union multiple columns into one.
XML Intersect	$\overset{x}{\cap}$	*col*+	*col*	*s*	Intersect multiple columns into one.
XML Difference	$\overset{x}{-}$	*col*+	*col*	*s*	Compute the difference between two columns.

Table 3.1: XML XAT Operators.

Operator	*Sym*	*Prms.*	*Output*	*Data*	*Description*
SQLstmt	*SQL*	*stmt*	*col*+	N/A	Execute a SQL query statement *stmt* to underlying database.
Function	$\{F\}$	*param*+	*col*	*s*?	XML or user defined function over optional input table with given parameters.
Source	*S*	*desc*	*col*+	N/A	Identify a data source by description *desc*. It could be a piece of XML fragment, an XML document, or a relational table.
Name	$\rho, \rightarrow$	$col1, col2$	N/A	*s*	Rename column $col1$ of source *s* into $col2$.
Name	$\rho, \rightarrow$	*ns*	N/A	*s*	Rename table *s* into new name *ns*.
FOR	*FOR*	*col*	N/A	s, sq	FOR operator iterate over *s* and execute subquery *sq* with the variable binding column.
IF_THEN_ELSE	*IF*	*c*	N/A	sq_1, sq_2	If condition *c* is true, then execute subquery sq_1, else execute subquery sq_2.
Merge	M	N/A	N/A	*s*+	Merge multiple tables into one table based on tuple order.

Table 3.2: Special XAT Operators.

Operator	*Sym*	*Prms.*	*Output*	*Data*	*Description*
Project	π	*col*+	N/A	*s*	Project out multiple columns from input table *s*.
Select	σ	*c*	N/A	*s*	filter input table *s* by condition *c*.
Cartesian Product	$\times$	N/A	N/A	s_1, s_2	Cartesian product of the results of two input tables, s_1 and s_2.
Theta Join	$\bowtie$	*c*	*col*+	ls, rs	Join two input tables ls and rs under condition *c*, and output selected columns.
Outer Join	$\overset{\circ}{\bowtie}_L, \overset{\circ}{\bowtie}_R$	*c*	*col*+	ls, rs	Left (right) outer join two input tables ls and rs, and output selected columns.
Distinct	δ	*col*+	N/A	*s*	Eliminates the duplicates in the columns *col*+ of input table *s* by grouping.
Groupby	γ	*col*+	N/A	s, sq_g	Making temporary groups by multiple columns from input table *s*, then evaluate subquery sq_g for each group, then merge the evaluated results back.
Orderby	τ	*col*+	N/A	*s*	Sort input table *s* by multiple columns.
Union	$\cup$	N/A	N/A	*s*+	Union multiple sources together.
Outer Union	$\overset{\circ}{\cup}$	N/A	N/A	*s*+	Outer union multiple sources together.
Difference	$-$	N/A	N/A	ls, rs	Difference between two sources.
Intersect	$\cap$	N/A	N/A	*s*+	Intersect multiple sources.

Table 3.3: SQL XAT Operators.

Operator	*Sym*	*Prms.*	*Output*	*Data*	*Description*
Delete	D	$col1, col2$	N/A	s	Delete child element $col2$ of element $col1$.
Insert	I	$col1, Tag$	N/A	s, sq	Insert newly constructed element Tag into element identified by $col1$.
		$col1, Tag, B/A, col2$	N/A	s, sq	Insert newly constructed element Tag into element identified by $col1$ before/after element identified by $col2$.
		$col1, fn$	N/A	s, sq	Insert new attribute or ref. defined by function fn into element identified by $col1$.
		$col1, cont, B/A, col2$	N/A	s, sq	Insert content $cont$ into element identified by $col1$ before or after element identified by $col2$.
Replace	R	$col1, col2, Tag$	N/A	s, sq	Within element identified by $col1$, replace element identified by $col2$ with newly constructed element Tag.
		$col1, col2, fn$	N/A	s, sq	Within element identified by $col1$, replace attribute/ref. identified by $col2$ with attribute/ref. defined by function fn.
		$col1, col2, cont$	N/A	s, sq	Within element identified by $col1$, replace content identified by $col2$ with content $cont$.
Rename	Re	$col1, col2, name$	N/A	s, sq	Within element identified by $col1$, rename element identified by $col2$ to $name$.

Table 3.4: XML Update Operators.

Chapter 4

View Analyzer

As described in Section 1.4, the *View Analyser* is the module in the *Update Manager* that processes the view query (view definition) to gather semantic information about the underlying relations. Each view is processed in this manner only once at the time of the view definition and information gathered during this analysis is stored in the system henceforth to aid in propagating XML updates to the underlying relational database. Before we talk about the working of this module let's first take a look at the formal concepts that we shall use to describe the module.

4.1 Assumptions And Restrictions

Below we discuss the assumptions and restrictions underlying the definition of views that enable unambiguous translation of updates. We also describe our assumtions on the type of updates. Relational views are defined to be stored database queries. Significant restrictions are imposed on views in order to be updatable. Similarly, for defining XML views of relational data, significant restrictions must also be imposed on such XML views so as to be updatable.

4.1.1 Restrictions on the View Definition

In this section we discuss the restrictions that should be imposed on XML views of relational data to have these views updatable. These restrictions have been discussed in the context of relational views by [Kel86].

1. **No aggregations in view definition:** It is undesirable to update views that include *aggregations*. For example if an element in an XML view represents the sum of books by a particular author as shown in Figure 4.2 then a view update request to increment the number of books by the author will require to add an unknown (better chosen by a human) book to the **Books relation and also an extra tuple into the Authors relation with the same** *bookid*. **A request to decrement the number of books for a particular author would require the computer to choose a book to be deleted - a decision best done by a human. Thus updating views that include aggregations is not meaningful in a completely automated fashion. Here, in this work we thus do not allow aggregations to appear in the view definition.**

2. **No explicit functional dependencies in view definition: In relational view context, in an** *explicit functional dependency* **[SSC84], one or more of the attributes in a tuple can be computed based on the values of some other attributes. In the context of XML views over relational data this would mean that a particular element in the XML view can have values computed from values of one or more other attributes of a relation. One such explicit functional dependency is shown in Figure 4.4. Element** *sale_price* **has its value computed from the relational attribute** *price* **using a user-specified function. An update request to change the value of** *sale_price* **will not convey any clear strategy of how the underlying attribute** *price* **should be**

Books

Bookid	Title	Year
1	Data Structures	2000
2	JAVA Complete Reference	1999
3	Algorithms in C++	2001

Authors

First	Last	BookId
Michael	Savitch	1
Peter	Naughton	2
Michael	Savitch	3

Figure 4.1: Relational Tables in Database

XML view

```
<authors>
  <author>
        <first>Michael</first>
        <last>Savitch</last>
        <no_of_books>2</no_of_books>
  </author>
 <author>
        <first>Peter</first>
        <last>Naughton</last>
        <no_of_books>1</no_of_books>
  </author>
</authors>
```

Figure 4.2: XML View (with Aggregation) of Relations Shown in Figure 4.1

updated. Thus there is no good way of handling explicit functional dependencies and should those be avoided for updatable views.

Books

Bookid	Title	Year
1	Data Structures	2000
2	JAVA Complete Reference	1999

Prices

Title	Source	Price
Data Structures	Amazon.com	50.00
JAVA Complete Reference	Ebay.com	60.00

Figure 4.3: Relational Tables in Database

XML view

```
<books>
  <book>
        <bookid>1</bookid>
        <title>Data Structures</title>
        <sale_price>45.00</sale_price>
  </book>
  <book>
        <bookid>2</bookid>
        <title>JAVA Complete Reference</title>
        <sale_price>54</sale_price>
  </book>
</books>
```

Figure 4.4: XML View (with External Dependency) of Relations Shown in Figure 4.3

3. **Operators : As pointed out in [Kel86], the relational operators Select, Project, and Join are more suitable than some of the other relational operators (for example, Divide and Set Difference) for defining updatable relational views. The Select operator extracts tuples from a relation that satisfy a selection condition. The Project**

operator extracts the desired attributes from each tuple in the given input. The Join operator combines two relations and in doing so, combines tuples with matching attribute values. To define updatable XML views of relational data the XML algebra counterparts of those relational operators still continue to make more sense as pointed out in the description of XAT operators in section 3.1.2. We have Select and Join which achieve the similar functionality of Select and Join in the relational algebra. One way to project elements in an XML view is to navigate into parent elements and expose desired sub-elements as they are and with the same name. The other way is to extract data values of elements, tag them and then expose them. Thus Navigate, Tagger and Expose together achieve the functionality of a Project operator in relational algebra. XML operators discussed so far are sufficient for defining XML views that do not include any aggregations or explicit functional dependencies. Hence operators used to define XML views should be restricted to Navigate, Select, Join, Tagger and Expose and these are the operator that we will consider as well.

4.1.2 Assumptions on Type of Updates

Having discussed the constraints we impose on defining views in the previous section, below we now state our assumptions on the type of update requests we assume will be issued on our XML views of relational data.

1. We do not validate any updates issued on XML views, rather we assume that the updates issued are valid updates. Valid here means that the XML document expected as result of the execution of the issued updates will still conform to the XML schema of the original documents assuming one had

been given. This means that issued updates do not change the schema of the original view. For example, an update request on the XML view shown in Figure 4.2 to add the *age* element within the *author* element is not valid as the resulting view will not conform to the XML schema that does not have the *age* element appearing as part of the *author* element.

2. Since we are dealing with XML views of relational data, the structure of the XML view constructed of the relational data (modeled by the flat relational data) is quite different from its relational counterpart. Hence, an update issued on an XML view might require a schema change at the relational database even if we follow the assumption 1 from above. One such example is renaming an element in the XML view that represents an attribute column in the underlying relation. Renaming this element will require renaming this attribute of the relation and hence will result in a schema change at the relational end. We now assume for simplicity that only those updates that result in a data change at the relational end are issued on XML views. We do not check to verify that indeed a particular update results only in data change at the relational end, rather we assume that we get only *"good"* update requests.

3. We consider an XML view to be defined as a collection of elements and assume that updates issued are *Complete Updates* [TBW91] issued on the said elements. For example as shown in Figure 4.6, an update request for the deletion of a complete **book** element, replacement of a complete **book** element with another **book** element, and insertion of a newly constructed complete **book** element are expected. Partial updates that will update only a part of a **book** element are not yet supported by our approach.

XML View

```
<books>
  <book>
        <bookid>1</bookid>
        <title>Data Structures</title>
        <year>2000</year>
        <source>Amazon.com</source>
        <price>54.00</price>
        <author>
          <first>Michael</first>
          <last>Savitch</last>
        </author>
  </book>
  <book>
        <bookid>2</bookid>
        <title>Java Complete Reference</title>
        <year>1999</year>
        <source>Ebay.com</source>
        <price>65.00</price>
        <author>
          <first>Peter</first>
          <last>Naughton</last>
        </author>
  </book>
</books>
```

Figure 4.5: XML Virtual View

Complete Delete Update

```
FOR $root IN document("books.xml")/books
LET $book := $root/book
WHERE $book/author/first = "Peter"
  AND $book/author/last = "Naughton"
UPDATE $root{
 DELETE $book
}
```

Complete Insert Update

```
FOR $root IN document("books.xml")/books
UPDATE $root{
      INSERT <book>
                <bookid>"3"</bookid>,
                <title>"My New Book"</title>,
                <year>"2003"</year>,
                <source>"Ebay.com"</source>,
                <price>"54.50"</price>,
                <author>
                   <first>"John"</first>,
                   <last>"Doe"</last>
                </author>
             </book>
}
```

Figure 4.6: Example of Complete Update Queries Issued on XML View Shown in Figure 4.5

4.2 The Structural Model

The structural model of a relational database is a formal semantic data model constructed from relations that express entity classes and form relationships, or connections, among those classes [WE80]. The structural model defines a directed-graph representation of the database, where vertices correspond to relations and edges to connections (relationships).

Definition 1 [TBW91] **A Connection** *is defined by the two relations R_1 and R_2 being connected, and by the two subsets of attributes X_1 of R_1 and X_2 of R_2 such that X_1 and X_2 have identical number of attributes and domains. R_1 and R_2 are then connected through the ordered pair(X_1, X_2).*

For two connected relations R_1 and R_2, two tuples $t_1 \in R_1$ and $t_2 \in R_2$ are connected if and only if the values of the connecting attributes in t_1 and t_2 match.

The structural model defines three types of connections, according to the semantics of the relationships between the two relations. Most importantly for our purpose, the connection types carry precise integrity rules. Let K(R) and NK(R) be the key and nonkey attributes of relation R respectively.

Definition 2 [TBW91] **An ownership connection** *from R_1 to R_2 is specified by the following criteria :*

1. *Every tuple in R_2 must be connected to an owning tuple in R_1.*
2. *Deletion of an owning tuple in R_1 requires deletion of all tuples connected to that tuple in R_2.*
3. *Modification of X_1 in an owning tupple of R_1 requires either propagation of the modification to attributes X_2 of all owned tuples in R_2 or deletion of those tuples.*

The ownership connection embodies the concept of dependency, where owned tuples are specifically related to a single owner tuple. As a result, we must have X_1 = K(R_1), and $X_2 \subset$ K(R_2). The cardinality of the ownership connection is 1:n.

Definition 3 [TBW91] **A reference connection** *from R_1 to R_2 is specified by the following criteria:*

1. *Every tuple in R_1 must either be connected to a referenced tuple in R_2 or have null values for X_1.*
2. *Deletion of a tuple in R_2 requires either deletion of its referencing tuples in R_1 or assignment of valid or null values to attributes X_1 of all the referencing tuples in R_1.*
3. *Modification of X_2 in a referenced tuple of R_2 requires any one of several possible propagations namely, either propagation of the modification to attributes X_1 of all referencing tuples in R_1, assignment of null values to attributes X_1 or all referencing tuples in R_1, or deletion of those tuples.*

The reference connection relates one entity (the referencing relation) to another more abstract entity (the referenced relation). As a result, we must have $X_1 \subset \mathrm{K}(R_1)$ or $X_1 \subset \mathrm{NK}(R_1)$, and $X_2 = \mathrm{K}(R_2)$. The cardinality of the reference connection is *n:1*.

Definition 4 [TBW91] **A subset connection** *from R_1 to R_2 is specified by the following criteria:*

1. *Every tuple in R_2 must be connected to one tuple in R_1.*
2. *Deletion of a tuple in R_1 requires deletion of the connected tuple in R_2 (if the latter exists).*
3. *Modification of X_1 in a tuple of R_1 requires either propagation of the modification to attributes X_2 of its connected tuple in R_2 or deletion of the R_2 tuple.*

Specialization of the general entity can be implemented by defining more specific entities connected to the main one through subset relationships. As a result, we must have $X_1 = \mathrm{K}(R_1)$ and $X_2 = \mathrm{K}(R_2)$. The cardinality of the subset connection is *0:1*.

Note that *m:n* relationships are not modeled directly in the structural model but can be represented using combinations of connections. Finally, if there is a connection $\mathcal{C}$ (one of the above three connections) from relation R_1 to relation R_2, then there is an inverse connection $\mathcal{C}^{-\infty}$ from relation R_2 to relation R_1.

Below we give a few more definitions of the terms borrowed from the literature [TBW91] that we shall use later to describe our approach to updating a relational database through views which is based on Keller's approach [TBW91].

As discussed in Section 4.1.2, we consider the XML view defined as a collection of elements and each of these elements is analogous to an object that is constructed from projections on one or more relations. Each element of the collection has the same XML schema. Below we quote the formal definition of an object as defined in [TBW91]. Let R be the domain of all relations for a given relational database. Let

Π denote the domain of all projections π defined on R. Let Set(Πe) designate the domain of all finite sets of projections. In addition, a function d : $\Pi \rightarrow$ R such that d(π) is the relation on which π is defined.

Definition 5 An object ω *is a non-empty element of Set(π) (set of projections), denoted by* $\omega = \{\pi_1, \pi_2, ..., \pi_n\}$ *where* πs *are projections defined on R - the set of relations. The complexity of* ω *is defined to be the number of projections included in the object.*

Definition 6 *For each object* ω*, we further define a* **pivot** *relation* $R_1 \in R$ *such that*

- $\exists$ *only one* $\pi_j \in \omega | [d(\pi_j) = R_1] \wedge [K(R_1) \subseteq \pi_j]$
- $K(\omega) = K(R_1)$
- $\forall k, k = 1.....i, k \neq j, d(\pi_k) \neq R_1$

The notion of *pivot relation* is central to the formalism. Each object is "anchored" on one base relation, which constitutes its core component. We extend the notion of a relational key to an object key, such that the key of an object ω is isomorphic to the key of its pivot relation. Hence the requirement that the key attributes be included in the projection defined on the *pivot relation*. Thus each element of the collection in the XML view has a unique identifier and like its relational counterpart, this key permits unique identification of any instance of the given object type (XML element). For our running example as shown in Figure **??**, the relation **Books** qualifies as pivot relation as its key is visible in a view element and also the key of the **Books** relation is a unique identifier for the view elements.

Definition 7 *The* **dependency island** D_ω *of a view object* ω *is the maximal subtree of the tree of projections such that (1) the root of the subtree is the pivot relation, and (2) all directed paths starting at the pivot relation must be composed only of*

Underlying Relations

Books

Bookid	Title	Year
1	Data Structures	2000
2	JAVA Complete Reference	1999

Authors

First	Last	BookId
Michael	Savitch	1
Peter	Naughton	2

Prices

Title	Source	Price
Data Structures	Amazon.com	54.00
JAVA Complete Reference	Ebay.com	65.00

Figure 4.7: Relational Tables in Database

XML View

```
<books>
  <book>
      <bookid>1</bookid>
      <title>Data Structures</title>
      <year>2000</year>
      <source>Amazon.com</source>
      <price>54.00</price>
      <author>
        <first>Michael</first>
        <last>Savitch</last>
      </author>
  </book>
  <book>
      <bookid>2</bookid>
      <title>Java Complete Reference</title>
      <year>1999</year>
      <source>Ebay.com</source>
      <price>65.00</price>
      <author>
        <first>Peter</first>
        <last>Naughton</last>
      </author>
  </book>
</books>
```

Figure 4.8: Virtual View of Relations Shown in Figure 4.7

ownership and subset connections.

Definition 8 *A* **referencing peninsula** *is a relation $R_j \in d(\omega)$ that is directly connected to any relation Rk of the dependency island by a reference connection.*

The rationale behind the dependency island is that all the relations in the dependency island belong to the same entity - the entity that is centered on the pivot relation. As a result, any update operation on the view element should have consistent repercussions throughout the components of that object's dependency island. Referencing peninsulas, on the other hand, must be identified because of the constraints of referential integrity. For elements of a view shown in Figure 4.7 relations **Books** and **Authors** form the dependency island with the relation **Books** also as the *pivot* relation whereas the relation **Prices** is a referencing peninsulas whose attribute **title** refers to the unique attribute **title** of relation **Books** in the dependency

island.

4.3 Analysis of the View XAT

The view is analyzed only once when it is defined. Analysis of the view construction to gather semantic information about the underlying relations helps to analyze the possible impact of any update on the underlying database. It gives us a priori knowledge of how a particular relation involved in the construction of a view may be affected by an update issued on the view. The data collected during analysis of the view is stored later in the system and is accessed for interpreting all updates issued on this view. In our approach analysis of the view is done by processing the view XAT generated from the mapping or view query. The following steps are typically carried out while analyzing a view :

- **Finding names of the underlying relations:** The first and foremost step taken is to process the view XAT to find out how many and which relations are involved in the construction of a view. Since the view is constructed on top of the default XML view which has attribute values for each tuple of a relation within the *Row* element, the view query will always navigate into the *Row* element to extract attribute values for the construction of the virtual view. Hence the view query will navigate to the *Row* element via the parent of the *Row* element which is named after the name of the respective table. This navigation which is expressed as an XPath expression in an XQuery statement is then captured by the *Navigate* operator in the view XAT. Hence these navigate operators also carry knowledge of the underlying relations. Each navigate operator that has a *Row* element as a step in its destination also carries the name of the underlying relation. Thus knowledge of all underlying relations can be easily extracted from the navigate nodes. Figure 4.9 depicts the XAT of the view query for our running example. One can see that the navigate

operators having a *Row* step in their destination telling us about the underlying relation too. In our example there are three underlying relations namely **Books, Authors** and **Prices**. The algorithm for processing the navigate operator to extract the relation names is shown in Figure 4.10.

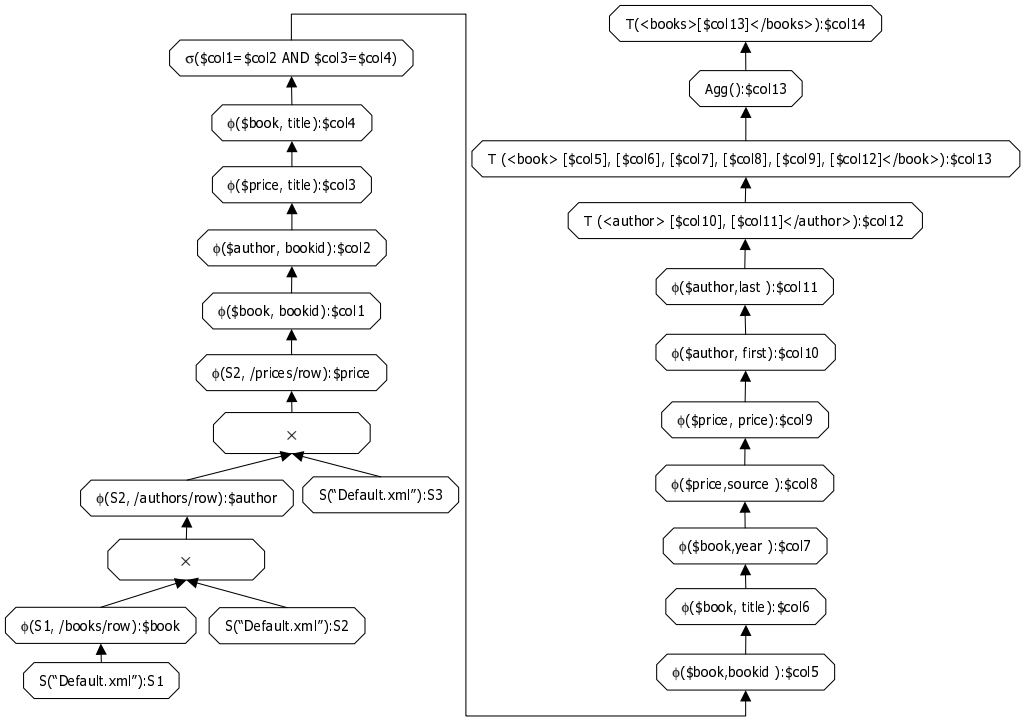

Figure 4.9: XML Algebra Tree for View Query

- **Collecting meta information about the underlying relations:** After discovering names of the relations underneath the given view, next step is to collect meta data such as keys, foreign keys, names and data types of attributes as well as other constraints on the attributes of the relations. Relational database is queried for this set of information for each underlying relation and information is stored in a data structure, in our case a relational table. The reason for collecting all this information is multi-folded. First, the knowledge of keys and foreign keys of a particular relation will help as to determine its relationship with other underlying relation which in turn will help in identifying the dependency island and referencing peninsulas. Secondly

```
findTable(NavigateOperator nav){
    If nav has destination then
        If destination has 'Row' step then
            Case 1: 'Row' is first step of destination
                // Last step of entry point has table name
                tablename = last step of entry point
            Case 2: 'Row' is last step of destination
                // Last but one step of destination has table name
                tablename = last but one step of destination
            Case 3: 'Row' is intermediate step of destination
                // previous step of destination has table name
                tablename = a step prior to 'Row' step of destination

            If tablename is a variable-binding then
                tablename = get actual-name from binding table.

    return tablename
}
```

Figure 4.10: Algorithm to Search for a Relation in Navigate Operator

the knowledge of attributes and their data types will help in extending tuples with attributes that do not appear the in view. Extending tuples is required as partial tuples cannot be inserted into relations. Note that the decision of values of attributes for extending tuples is application dependent.

- **Identifying Pivot Relation:** As discussed earlier while formally defining a pivot relation, the intuition behind having a pivot relation for view elements is that each element is "anchored" to one relation. Any update on view elements will have a direct effect on the pivot relation and other relations will be affected indirectly depending upon their relationship with the pivot relation. As stated in the definition, the key of the pivot relation is also the unique identifier for view elements and hence the key of the pivot relation must be visible in the view. Starting with this fact, now that we have knowledge of all underlying relations and their keys, we traverse the algebra tree to find a relation whose key is exposed in the view. Note that there might be multiple such relations but we are looking for atleast one which is a sufficient condition for the view to be updatable. Figure 4.11 shows the algorithm we followed to search for

a pivot relation. For our running example, as shown in Figure 4.9, **$col5** is exposed in the view which happens to be the **bookid** attribute of the table **Books**. Since **bookid** is the key of the table **Books** it qualifies to be the *pivot* relation for our view and table **Books** being the only candidate for pivot relation becomes the *pivot* relation for the XML view in our example.

```
searchPivotRelation( ){
    for each underlying relation
        data-structureVector = createDataStructureVector(Keys of relation)
        for each leaf of the view tree
            start traversing tree bottom-up from the leaf
                if current node has Tagger Operator then
                    tag_contents = contents of Tagger
                    for each column name in tag_contents
                        if this column name is from current relation then
                            attribute_name = get actual name for column name from binding table
                            for each data-structure ds in data-structureVector
                                if attribute of ds is not in view and attribute_name = attribute then
                                    mark this attribute to be in view.
                                    if all attribute are marked to be in view then
                                        current relation is pivot relation
                                        return
}

class specialDataStructure{
        String attribute;
        boolean isInView;
        String getAttribute();
        void setAttribute(String);
        boolean getIsInView();
        void setIsInView(boolean);
}
```

Figure 4.11: Algorithm to Search for Pivot Relation

As shown in Figure 4.11, the view tree is traversed bottom-up starting at each leaf. We traverse the tree bottom-up once for each underlying relation until we find a pivot relation. While traversing we keep a vector of special data structures (also shown in Figure 4.11) created from keys of the relation in question. *Tagger* operators are of prime interest as they hold the information of elements visible in the view. So we process each tagger operator marking which key attributes of the relation in question are exposed in the view. Once we find that all the key attributes of the current table are in the view, we stop the search and record this relation as pivot relation. During this whole

process, we do need to decipher the names of bound columns. We also need to trace their paths in order to know the actual names of attributes exposed and whether the attribute belongs to the relation in question in order for it to be considered to be the pivot relation. For this we make use of the binding table - a hashtable which stores bindings and the XPath associated with each binding.

- **Identifying Dependency Island and Referencing Peninsulas:** Relationships among the underlying relations namely ownership connections, subset connections and reference connections can be identified using information about keys and foreign keys of the relations. Definitions of these connections as stated earlier in the section can be restated using notion of keys and foreign keys as below:

  1. **Ownership Connection** from relation R_1 to R_2 is said to exist if the foreign key (also part of key) of relation R_2 which may not be unique is refering to the key of relation R_1.
  2. **Subset Connection** from relation R_1 to R_2 is said to exist if the foreign key (also key) of the relation R_2, also unique, is refering to the unique key of relation R_1.
  3. **Reference Connection** from relation R_1 to R_2 is said to exist if the foreign key of relation R_1 with null values allowed for foreign key attributes and the foreign key not necessarily unique in R_1 is refering to a key or a unique attribute of relation R_2.

  Having identified the set of the underlying relations and the pivot relation for the given XML view, each relation from the set of our underlying relations is analyzed, using the above given defintions, for having a **Ownership** or **Subset** connection with the pivot relation. If a **Ownership** or **Subset**

connection exists from the pivot relation to a particular relation, then the relation is marked to be a part of the *Dependency Island*. Thus the pivot relation along with any other relations having **Ownership** or **Subset** connections with the pivot relation form the *Dependency Island* for a given XML view. For our running example a **Subset Connection** from the relation **Books** to the relation **Authors** exists. Hence the relation **Authors** falls in *Dependency Island*. Relation **Books** also, being a *pivot* relation, is a part of the *Dependency Island*.

Relations that do not qualify to be part of the *Dependency Island* are then analyzed, using the above given definition of reference connection, for having reference connection with any relation falling into the *Dependency Island* (i.e., a reference connection from a particular relation to any relation in the dependency island). If such a connection exists then the relation under question qualifies as one of many possible *Referencing Peninsulas* for the given XML view. The algorithm for identifying the *Dependency Island* and *Referencing Peninsulas*, being relatively simple, has not been discusssed here. There exists **Reference Connections** between the *pivot* relation **Books** and the relation **Prices** and so the relation **Prices** is a *Referencing Peninsula* in our running example.

- **Storing into the system the knowledge of Dependency Island, Referencing Peninsulas and other underlying relations:** All the knowledge about the underlying relations and their relationships, gathered in the form of the *Dependency Island* and *Referencing Peninsulas*, is stored on persisitent storage. This knowledge is then used for the translation of updates as discussed in later chapters.

Chapter 5

Update Decomposer

The *Update Decomposer* is yet another component specific to the *Update Manager* in Rainbow. This component decomposes the XML update given to it in the form of XAT into multiple updates that are relation specific. Each of these updates is an update on a single relation that belongs to the pool of relations that consitute the foundation of the XML view on which original XML update is issued. The *Update Decomposer* decomposes the update using the information that is available from the XAT. This component does not know anything about the underlying relations and relationships among them if any. It simply extracts information about the relation names and column names available in the XAT. Thus it cannot differentiate between relation and hence may generate multiple but different updates against one particular relation. This kind of situation will arise in the case where the XML view was constructed using a join of a particular relation with itself. The decision about whether or not execution of any of these updates is necessary to reflect the effect of the XML update on the XML view depends on rules that take into consideration the existence of the *Pivot Table, Dependency Island* and *Referencing Peninsulas* as discussed in earlier chapters. The *Update Translator* component is aware of the rules and is the component that takes such decisions.

5.1 Decomposition of Updates

For the decomposition of updates, we follow a traversal strategy that traverses the input XAT bottom up. The intuition behind the bottom up traversal is that each distinct leaf at the bottom of the tree contains information about one relation that was involved in the formation of the XML view. This relation, hence, is also a candidate for being updated as a result of any update intended on the XML view. The XML update represented by the update XAT will be decomposed into a number of relational updates equal to the number of distinct leaves of this XAT. By traversing bottom up we are guaranteed to start with knowledge of the relation names which we extract from a leaf and its immediate parent node. Besides, we also collect knowledge of relational attributes to be updated within a relation and filters, if any, on values of attributes that may qualify them to be a candidate for the update. In the case of *Insert* and *Replace* XML updates, the new values for attributes that will be updated are also extracted and put into the data structure representing the relational update being constructed. Thus, a relational update is incrementally constructed while traversing the update XAT bottom up starting at a leaf. Also, knowledge of join conditions for joining two tables is extracted from the *Join* nodes to be applied to the relational update being constructed.

Shown in Figure 5.1 is the algorithm for the traversal strategy that we follow to construct relational updates. Figure 5.1 also shows the data structure that represents a relational update, known as *Relational Update*. *Relational Update* is designed to keep information about the type of the update, the relation name for which the update is being constructed and knowledge of the attributes to be updated within the relation. Information about the attributes is stored in the form of a vector of *Update Column*. *Update Column* is yet another data structure designed to store the name of the attribute and its values. Of course, in case of a *Delete* update, the

vector of *Update Column* will be empty as complete records will be deleted from relations and we do not care for any attributes in particular. But in case of an *Insert* update each item in the vector of *Update Column* will store the name of the attribute and the value (new value) that is to be inserted. Besides, the *WhereClause* of the *Relational Update* will store all conditions applicable to the relational udpate. If a *Replace* update is being constructed then each item in the vector of the *Relational Column* will also contain old values which are to be replaced by the new ones. *Relational Update*, though being a very simple data structure, is sufficient to store all the information required to construct a single relational update.

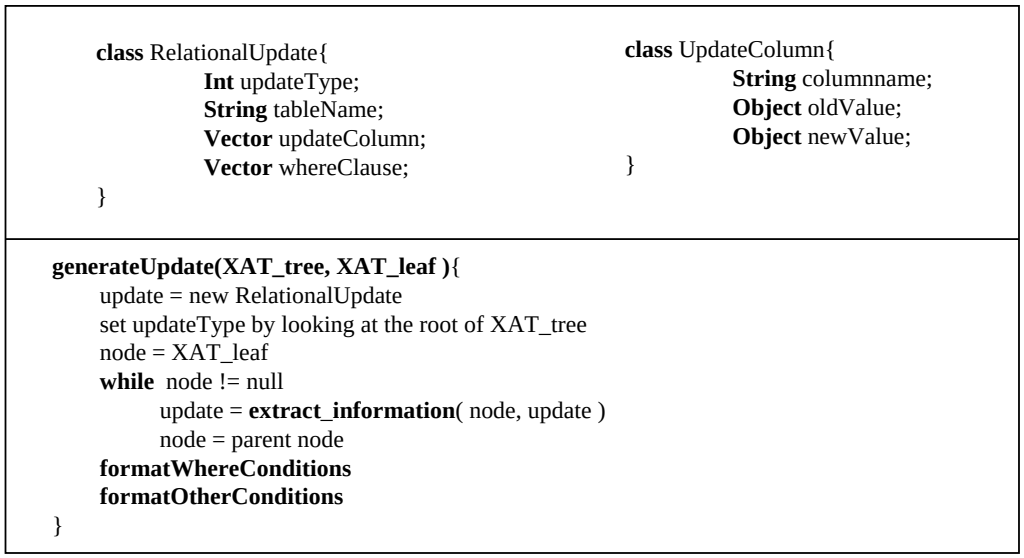

Figure 5.1: Algroithm for Decomposition of XML Update.

As stated in the algorithm in Figure 5.1, the update type is decided by looking at the root node of the tree which, essentially, is the update node. Then we start with a given leaf node and traverse up in the tree untill we reach the top of the tree. During our traversal we keep collecting information from nodes to construct the relational update. Again, we only care for nodes that have information required to help us construct the relational update. We ignore all other types of nodes. In particular, we care for Navigate, Select, Join, and Tagger nodes. Table 5.1 summarises what

```
extract_information( XAT_node, update){
    if XAT_node is a Navigate node
        update.tableName = get table name from node if it has one
    elseif XAT_node is a Select node or a Join node
        add the condition into whereClause of update
        break any complex binary conditions into simple binary conditions and store the conditions
                to be referred while extending tuples in case of insert and replace updates
    elseif XAT_node is a Tagger node
        if type of update is Insert
          if the DOM pattern of element represented by the tagger matches DOM pattern of the
            element to be inserted
                extract names of attributes from tagger pattern
                extract values of attributes from pattern of element to be inserted
                fill in the updateColumn vector of update.
        if type of update is Replace
          if the DOM pattern of element represented by the tagger matches DOM pattern of the
            replacing element
                extract names of attributes from tagger pattern
                extract old values by querying the relational database
                extract new values from pattern of the replacing element
                fill in the updateColumn vector of update
    else do nothing
    return update
}
```

Figure 5.2: Algorithm for Extracting Information from a Node.

kind of information is gathered for different types of updates from different types of nodes. Similar details are also explained by the algorithm shown in Figure 5.2.

As shown in the algorithm for extracting information from a node in Figure 5.2, the table name can be extracted from the navigate nodes. Refer to chapter *View Analyser* for discussion on extracting the table name from a Navigate node. There will be only one such Navigate node for each branch of the tree. If the node is a Select or a Join node then we extract the conditions from these nodes to go as *where* clauses with the relational update. These conditions are stored in raw form, separate from relational update to aid later in extending tuples for *Insert* or *Replace* updates. [1] Tagger nodes carry essential information for *Insert* and *Replace* updates. If the tagger node represents the element same as the one being inserted

[1]XML view might not always expose all the attributes from a relation. In such cases, when an update is issued on an XML view we get information about only the exposed attributes of a relation. When this XML update is translated into a relational update, each record is only partially consisting of attributes exposed in the XML view. Relational database do not accept partial records and so the update tuples need to be extended with values for attributes that were not exposed in the view. How these records are extended and how the conditions from Select and Join nodes help in extension is discussed in later chapters.

then we get attribute names from the pattern of the element represented by this tagger. At the same time, corresponding values for attributes are gathered from the pattern of the element in the update (Insert and Replace) nodes. Our assumption that updates issued are complete and correct gives us the liberty to assume that the pattern of topmost tagger node and the update (Insert and Replace) node exactly match. Once we have traversed the tree from the bottom (leaf) of a branch to the top (root) we are assured of having gathered sufficient information required for the construction of relational update.

In following sections we discuss the decomposition of an XML update of every kind (Delete, Insert, Replace) through examples.

5.2 Delete Update Decomposition

Decomposition of a *Delete* update is the simplest decomposition among the three kinds of updates viz. *Delete, Insert and Replace*. As discussed earlier, the *Update Decomposer* decomposes the given XML update into relational updates on relational tables underneath the XML view. The *Update Decomposer* does not care about whether or not a relational update on a particular relational table is required or necessary to bring the intended update on the XML view. It simply decomposes the given XML update into the corresponding relational updates, one per underlying relational table. Figure 2.6 shows the view definition for our running example and Figure 4.9 shows the XAT for the given definition. This view is analyzed as discussed in Chapter 4 and meta information (names and data types of attributes, information about keys and foreign keys etc.) about underlying relations viz **Books, Authors** and **Prices** is stored. Also information about the *Dependency Island* and *Referencing Peninsulas* is extracted during the view analysis is available during the decomposition and translation of updates.

Figure 5.3 shows the XAT of the complete delete update shown in Figure 4.6 which is issued on the XML view defined by the query in Figure 2.6. The update is to delete *books* that have an *author* with first name as *'Peter'* and last name as *'Naughton'*. XAT shown in Figure 5.3 is a decorrelated XAT. For the sake of simplicity to visualize the query, the tree has been simplified to not show irrelevant nodes.

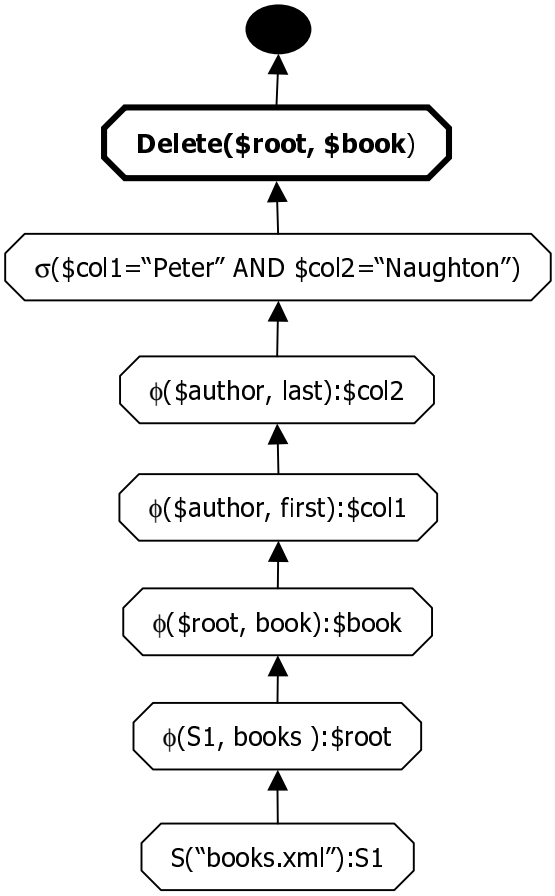

Figure 5.3: XAT for Delete Query Shown in Figure 4.6.

Following steps of our approach as shown in Figure 1.1 the two XATs are then merged and sent for computation pushdown, also known as rewriting of XATs. Figure 5.4 shows the XAT generated as result of rewriting. Again, the XAT has been simplified for the sake of understandability. The lower part of the tree (below the topmost tagger node) gives the understanding of the construction of our XML view whereas the top part of the tree (above the topmost tagger node) represents

the update part. The bottom part of tree tells us that there are three distinct Source nodes. Hence, as discussed in previous sections, the number of relational updates that will be generated will be three. One relational update will be generated for each branch of the tree and the relational update will be on the relational table referenced in that particular branch. In this example, one *Delete* update will be generated for each of the relational tables namely **Books, Authors, Prices**.

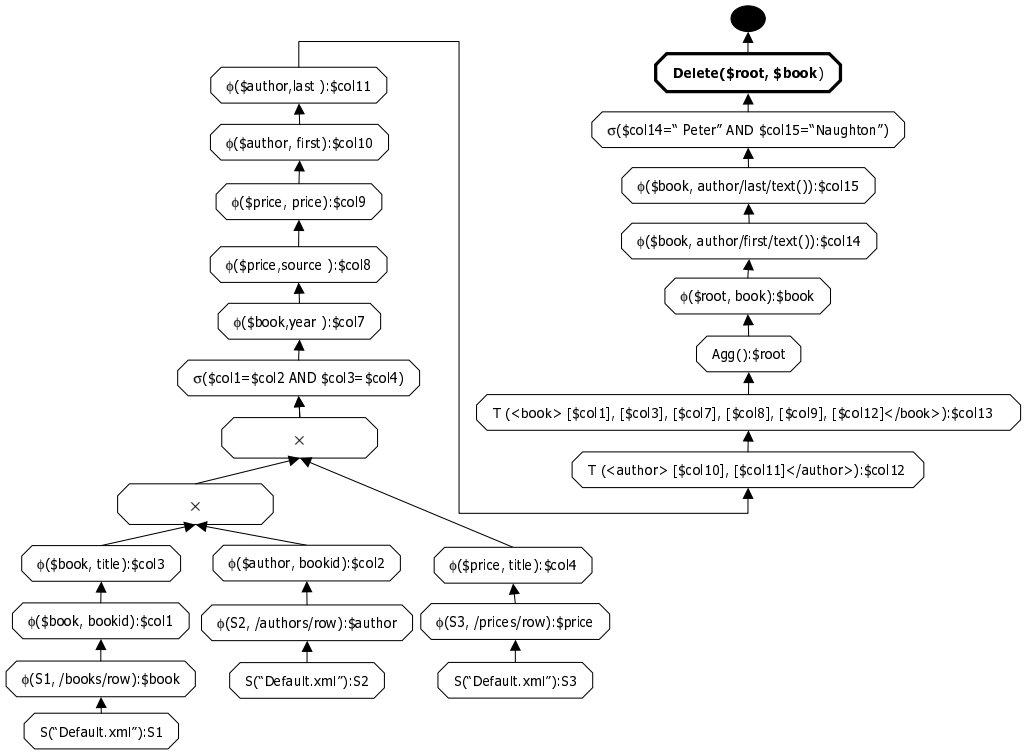

Figure 5.4: XAT as Result of Rewriting XAT Obtained After Merging XATs Shown in Figure 4.9, Figure 5.3.

First and foremost, the Delete node on top of the XAT tells us that the relational updates generated for all underlying tables will be *Delete* updates. We start at each leaf of the tree one by one and traverse up the tree following parent links and gathering information from each node as explained in Table 5.1. For this example, first we traverse the XAT shown in Figure 5.4 bottom up starting at the left most leaf which is a Source node **S1**. The Navigate node right above it gives us the name of the relational table which is **Books**. The two Select nodes in the path while

traversing up the XAT give relevant conditions which will be used in the *Where* clause of the *Delete* update that is being generated. At the end of the traversal of the XAT starting at the left most leaf of XAT we will have created the following *Relational Update* data structure.

```
Relational Update{
    updateType = Delete
    tableName = Books
    updateColumns = null
    whereClause = [$col1==$col2 AND $col3==$col4,
                   $col14=='Peter' AND $col14=='Naughton']
   }
```

Following the algorithm to generate the update shown in Figure 5.1 the next step is to format conditions listed in the *whereClause* of our data structure representing the relational update. By formatting we mean to translate the conditions into a format appropriate for SQL. Formatting the conditions in *whereClause* of the data structure shown above results the in following data structure:

```
Relational Update{
    updateType = Delete
    tableName = Books
    updateColumns = null
    whereClause = [books.bookid=authors.bookid AND
                   books.title=prices.title,
                   authors.first='Peter' AND authors.last='Naughton']
   }
```

Similarly, starting at the other two leaves of the XAT shown in Figure 5.4 we get two more data structures representing a relational update, each on tables **Authors** and **Prices** as shown below.

```
Relational Update{
    updateType = Delete
    tableName = Authors
    updateColumns = null
    whereClause = [books.bookid=authors.bookid AND
                   books.title=prices.title,
                   authors.first='Peter' AND authors.last='Naughton']
   }
```

```
Relational Update{
    updateType = Delete
    tableName = Prices
    updateColumns = null
    whereClause = [books.bookid=authors.bookid AND
                   books.title=prices.title,
                   author.first='Peter' AND authors.last='Naughton']
   }
```

5.3 Insert Update Decompostion

Figure 5.5 shows the XAT generated from the complete insert update shown in Figure 4.6. This insert update is issued on the XML view having a definition shown in Figure 2.6. The XAT generated for the view is shown in Figure 4.9. Again, as explained in Section 5.2 the XAT for the user update and the XAT for the view query are merged and then sent for computation pushdown, also known as rewriting. For the *Insert* update the XAT as a result of rewriting of the merged XAT is shown in figure 5.6. This rewritten XAT is then sent for update decomposition. Below is a discussion on how an *Insert* XML update is decomposed into corresponding *Insert* relational updates on the underlying relations.

To start decomposing the XML update we start at leaves and traverse the XAT bottom up gathering information for *Insert* update from each node as per table 5.1. Extraction of information from Navigate and Select nodes is done in a similar way as done in the case of the *Delete* update. But, unlike the *Delete* update, the Tagger nodes do carry important information for the *Insert* update. As briefed in Table 5.1, names and values of relational attributes are extracted from the Tagger nodes. For example, if we start at the left most leaf of the XAT shown in Figure 5.6, we come across the topmost tagger which represents the element to be inserted into the XML view. Since we identify it to be the element to be inserted in the XML view, we know that this is the element that carries values for the relational attributes. Since we know we are building an update on the relation **Books** we extract from the Tagger

node values for attributes belonging to the relation **Books**. So we get values for attributes **bookid, title** and **year** as *'3', 'My New Book'* and *'2003'* respectively. Figure 5.1 shows the data structure representing an update column. One such data structure is created for each of the attributes namely **bookid, title** and **year** with their respective names, new values as values extracted from Tagger node and old values as NULL. Old values are relevant to the *Replace* update as explained in the following section. Thus, at the end of the bottom up traversal of the XAT starting at left most leaf and after formatting the where clause conditions we will have the following data structure created for the branch where * suggests that the entry is of type *UpdateColumn*.

```
Relational Update{
    updateType = Insert
    tableName = Books
    updateColumns = [bookid*, title*, year*]
    whereClause = null
     }
```

In the case of the *Insert* and *Replace* update types all the conditions that fall in the path while traversing the rewritten XAT bottom up are broken into non-nested binary conditions and are stored separate from the relational update. These conditions are later referred to while extending the relational tuples to make sure that the join conditions between the underlying relations are preserved. The last statement of the algorithm in Figure 5.1 refers to these conditions which are to be formatted as where clause conditions. Similar to the generation of a data structure for relation **Books** as explained above, data structures representing a relational update are generated for relations **Authors** and **Prices** as well. They are as below:

```
Relational Update{
    updateType = Insert
    tableName = Authors
    updateColumns = [first*, last*]
    whereClause = null
     }
```

```
Relational Update{
    updateType = Insert
    tableName = Prices
    updateColumns = [source*, price*]
    whereClause = null
     }
```

5.4 Replace Update Decomposition

Figure 5.8 shows the XAT generated from the complete replace update shown in Figure 5.7. This replace update is issued on an XML view having its definition shown in Figure 2.6. The XAT generated for the view is shown in Figure 4.9. For the *Replace* update the XAT as a result of rewriting of the merged XAT is shown in Figure 5.9. This rewritten XAT is then sent for update decomposition.

Decomposition of the *Replace* update is done in a similar fashion as in the case of the *Insert* update the only difference being that we also need old values for the relational attributes. Old values for some of the attributes are available from the select conditions that appear in the XAT. For example, as seen in XAT shown in Figure 5.9, the *Select* node with the condition **'$col14=='JAVA Complete Reference'** gives us the old value for attribute **title** of the relation **Books**. Thus an *UpdateColumn* will be created for attribute *bookid* with its name, old value as **'JAVA Complete Reference'** and new value as **'JAVA Complete Reference Revised'**. All other attributes will have old values as NULL at the time of creation of *UpdateColumns*. Thus a relational update generated by the traversal of the XAT shown in Figure 5.9 will be as shown below where * suggests that the entry is of type *UpdateColumn* as defined in Figure 5.1 and + suggests that the entry also has an old value for the attribute.

```
Relational Update{
    updateType = Replace
    tableName = Books
    updateColumns = [bookid*, title*+, year+]
```

```
    whereClause = [books.bookid=authors.bookid AND
                   books.title=prices.title,
                   books.title='JAVA Complete Reference']
     }
```

Similarly relational updates generated by starting at the other two leaves of the XAT shown in Figure 5.9 will look as shown below:

```
Relational Update{
    updateType = Replace
    tableName = Authors
    updateColumns = [first*, last*]
    whereClause = [books.bookid=authors.bookid AND
                   books.title=prices.title,
                   books.title='JAVA Complete Reference']
     }

Relational Update{
    updateType = Replace
    tableName = Prices
    updateColumns = [source*, price*]
    whereClause = [books.bookid=authors.bookid AND
                   books.title=prices.title,
                   books.title='JAVA Complete Reference']
     }
```

Once relational updates have been generated, using information from all the update data structures we form an SQL query to query the relational database for old values for the attributes. The SQL query for this particular example will look like

```
SELECT books.bookid, books.title, books.year, authors.first,
       authors.last, prices.source, prices.price
  FROM books, authors, prices
 WHERE books.bookid=authors.bookid AND books.title=prices.title
   AND books.title='JAVA Complete Reference'
```

Old values for relational attributes for each relational update are then filled in from the result set of the SQL query.

Since all attributes of any particular relation may not be exposed in a constructed XML view, *Insert* and *Replace* relational updates generated by the *Update Decomposer* may not represent complete tuples of a relation. And since *Insert* and *Update* SQL statements require complete tuples being mentioned we need to extend tuples of updates generated by *Update Decomposer*. How do we extend these tuples is discussed in Chapter 6.

Node Type	*Update Type*	*Information*
Navigate	Delete	One *Navigate* node in the branch carries name of a relational table involved in formation of XML view. Current branch of the tree will be used to gneerate relational update on the table name extracted from this node.
	Insert	”
	Replace	”
Tagger	Delete	*Tagger* nodes carry information about subelements of XML element being updated. In other words, they carry information about attributes of relational tables involved in formation of XML view. Since relational delete update does not require attributes to be mentioned, this information is irrelevant.
	Insert	If output this node is used by the update node of the tree then this node carries values for subelements of the XML element being inserted in the view. In other words, it carries values for attributes of relational tuples. Hence, values of attributes for table on which the update is being generated are extracted from this node.
	Replace	If output of this node is used by the update node of the tree then this node carried values for subelements of the replacing XML element. In other words, it carries replacing values for attributes of relational tuples. Hence, values of attributes for table on which current update is being generated are extracted from this node.
Select/Join	Delete	These nodes carry conditions imposed on XML elements of the view to qualify for the update. In other words, when translated, these are the conditions to be imposed on relational tuples to qualify for the update. Also, join conditions, if any, for joining tables involved in formation of view are available from these nodes. Hence, these conditions are gathered to be used in *Where* clause of the relational update being generated.
	Insert	*Join* conditions are gathered and used later during translation of updates (as exlpained in following chapters) to make sure the *Join* conditions between relational tables are preserved for the tuples being inserted into different relational tables.
	Replace	*Join* conditions are gathered and used later during translation of updates (as explained in following chapters) to make sure the *Join* conditions between relational tables are preserved for the replacing tuples in different relational tables. Also, we extract from these conditions old values for attributes taking part in conditions. Old values for other attributes are collected via SQL queries to the underlying database. Old values for attributes are used during translation of updates (also explained in following chapters).

Table 5.1: Node Types and Information they carry.

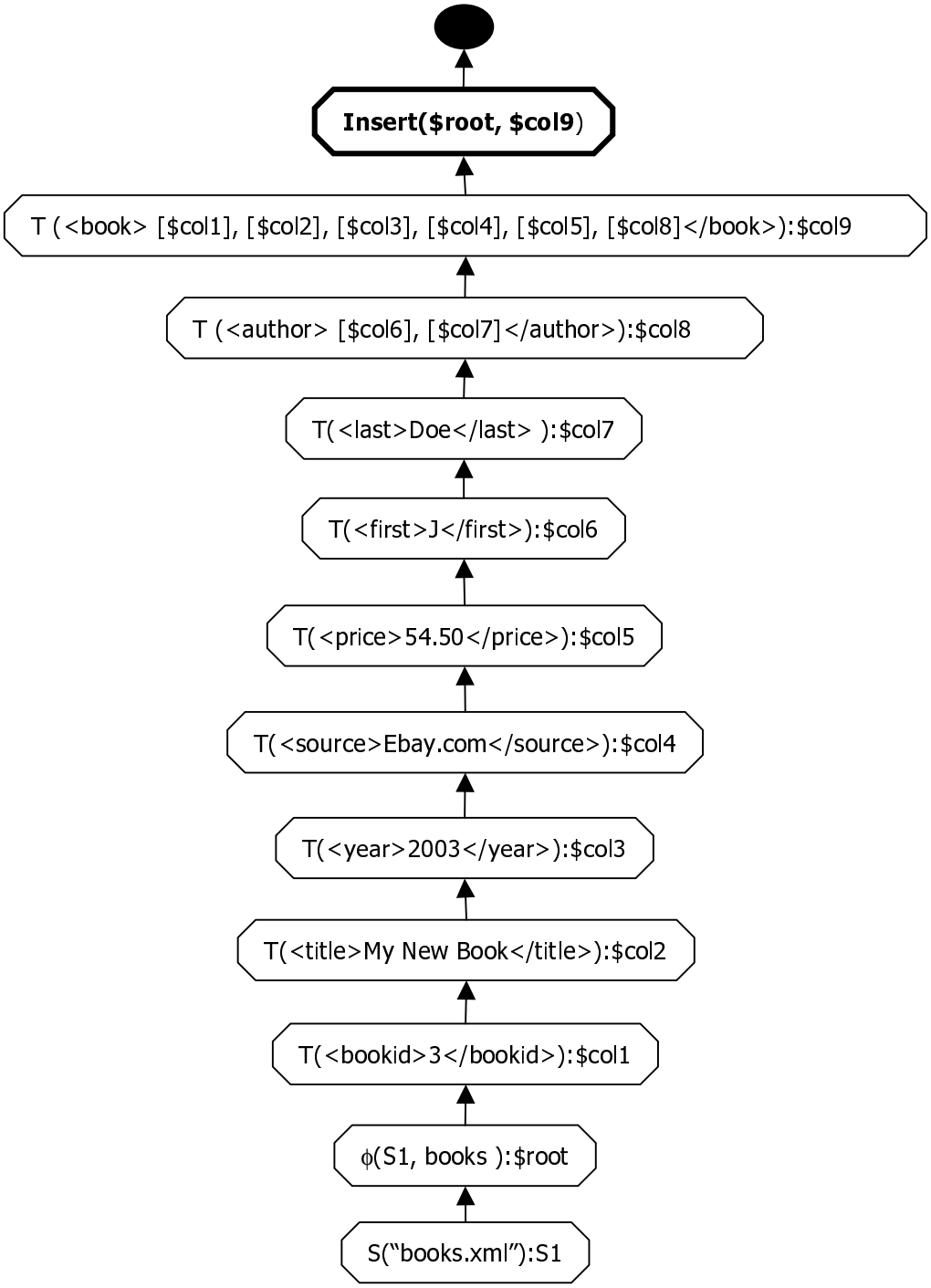

Figure 5.5: XAT for Complete Insert Update Shown in Figure 4.6.

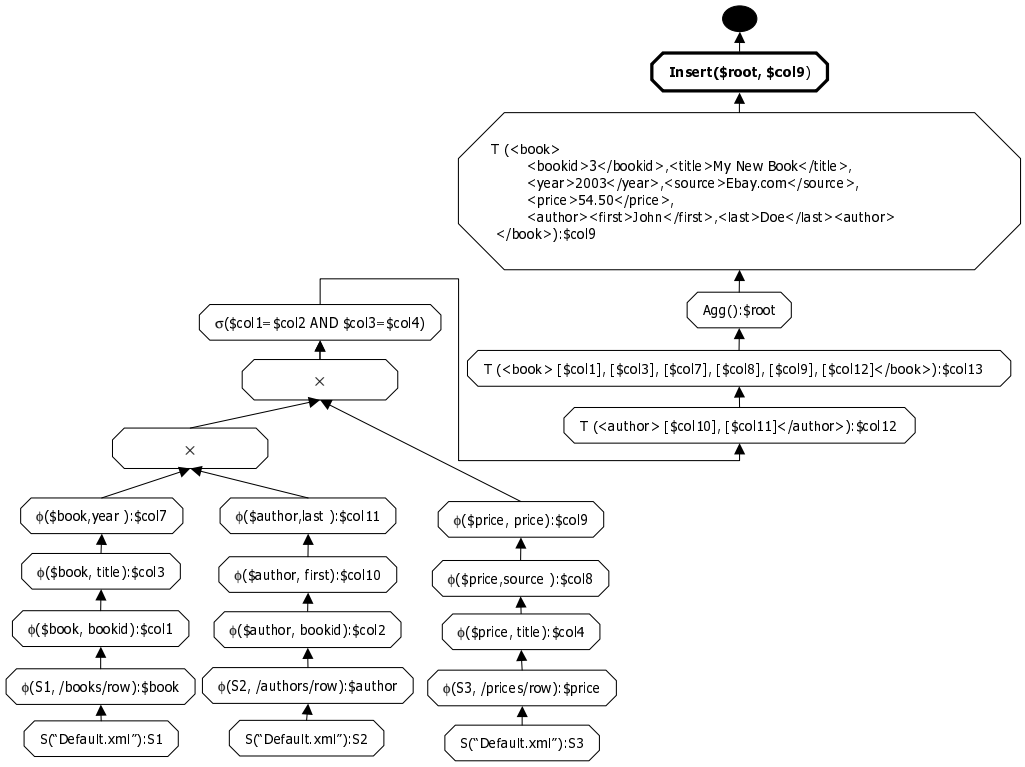

Figure 5.6: XAT as a result of rewriting XAT obtained after merging XATs shown in Figure 4.9 and Figure 5.5.

Complete Replace Update

```
FOR $root IN document("view.xml")/books
LET $book := $root/book
WHERE $book/title = "JAVA Complete Reference"
UPDATE $root{
   REPLACE $book WITH <book>
                        <bookid>"2"</bookid>,
                        <title>"JAVA Complete Reference Revised"</title>,
                        <year>"2001"</year>,
                        <source>"Ebay.com"</source>,
                        <price>"65.50"</price>,
                          <author>
                             <first>"Peter"</first>,
                             <last>"Naughton"</last>
                          </author>
                        </book>
}
```

Figure 5.7: Example of Complete Replace Update Query Issued on the XML View Shown in Figure 4.5.

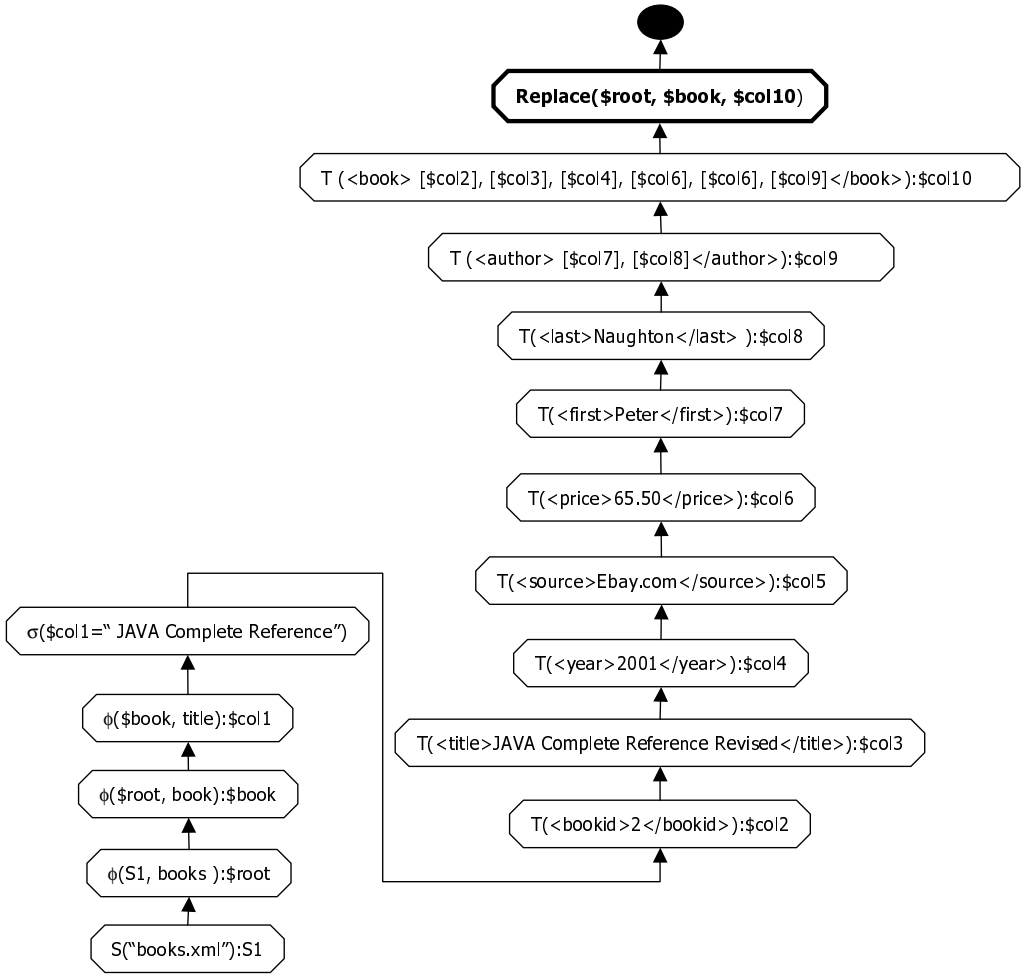

Figure 5.8: XAT for Complete Replace Update Shown in Figure 5.7.

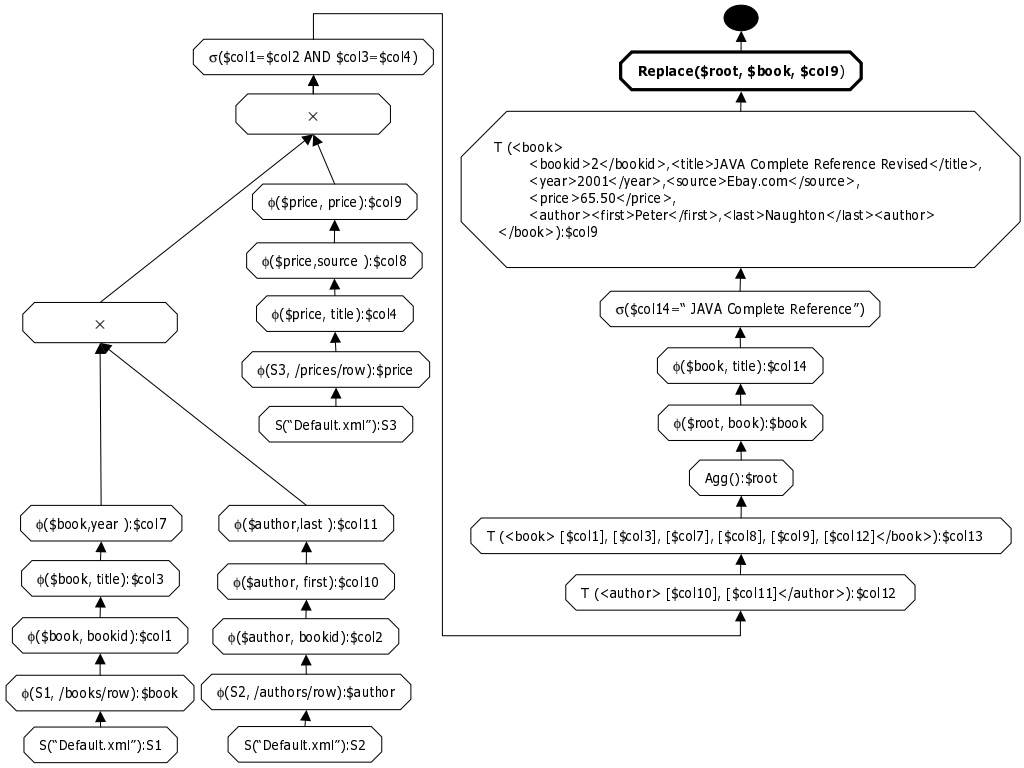

Figure 5.9: XAT as Result of Rewriting XAT Obtained After Merging XATs Shown in Figure 4.9 and Figure 5.8.

Chapter 6

Update Translator

Update Translator is the core part of the *Update Manager* and that applies translation rules to the updates generated by the *Update Decomposer*. *Update Decomposer* with its simplest logic decomposes the given XML update into corresponding relational updates on the underlying relations. Such decomposition is unrealistic and may not always bring the right change as desired by the update. Also, this sort of decomposition, if right, will do more changes to the underlying relational database than the minimal changes required to be done to bring the intended effect of the original XML update. For example, an XML update to delete a particular element from an XML view will be decomposed into *Delete* updates on each of the underlying relations. Since XML views in question are build around a *pivot* relation deleting from relations that are connected to the pivot relation through ownership or subset connection will do the minimal change to the underlying relational database. On the other hand, deleting from every relation taking part in the formation of the view will still bring the intended change in the XML view (when reconstructed from the updated relations). However unnecessary changes may be done to the underlying database. Similarly, inserting an element into the XML view will not necessarily require inserting a tuple into every relation taking part in the formation of the view, nor will replacing an existing element with a new one require a replacement of tuples

in each and every relation that contributes to the formation of the XML view. Such rules which guarantee the right and minimal changes to the underlying database form the translation algorithms for object views as designed by Barsalou, Keller and others in [TBW91]. We now discuss these translation algorithms in the next section and then nearly borrow from them to solve our problem.

6.1 Update Translation Algorithms

In their work on *Updating Relational Databases through Object-Based Views* [TBW91], Keller and others designed algorithms that can be used to translate complete updates on object views of relational data. Objects of views supported by their algorithms are *"anchored"* to one base relation which consitutes its core part. Thus a view object is a set of projections on base relations, where one of those is the *pivot* relation for the object. Their notion of objects view is identical to our notion of XML views in that each element of an XML view is made from a set of projections on base relations and is *"anchored"* to one base relation. This base relation is the *pivot* relation of XML elements or in general of the XML view. Also, similar to our goal of translating a complete XML update on a given XML view into relational updates on base relations, their goal also is to translate a given complete update on an object view into relational updates on the underlying relations. These similarities in the types of views being supported and the problem being tackled support the argument that we can re-use their algorithms to suit our needs. The algorithms discussed in the following subsections are taken from [TBW91] and modified to suit the problem of translating XML updates on XML views into relational updates on relational tables underneath.

6.1.1 Translation of Complete Delete Updates

The following algorithm is for the translation of complete delete requests issued over a view-object instance which in our case is a complete XML element.

Algorithm VO-CD: The input is an request for deleting a XML element. The output is the set of database operations that implement the request.

- Isolate the dependency island
- For each projection in the island, delete all matching tuples from the underlying relation
- Identify the referencing peninsulas
- For each peninsula, delete matching tuples or perform a replacement on the foreign key of each matching tuple.

To implement the fourth step in the algorithm we set **BEFORE DELETE** triggers that delete referring tuples from the peninsulas before deleting tuples from relations in the dependency island.

This algorithm has very controlled effects on the database; the process of global integrity maintenance can therefore be simplified to require only two operations. First, for relations in the dependency island that have an outgoing ownership or subset connections, the deletions must be propagated (repeatedly, if necessary) to those owned and subset relations. Second, in addition to the referencing peninsulas already handled, foreign-key replacements of deletions must be performed on any relation referencing one of the relations involved in a deletion. Note that no further propagation is needed outside of the referencing peninsulas and the referencing relations. Maintaining global consistency becomes more complicated with multiple views defined on the common base relations. Our system does not support global consistency at this time.

6.1.2 Translation of Complete Insert Updates

The transaction for complete-insert update is rejected only if identical relational tuples exists in relations that belong to dependency island. Inserting an XML element instance involves adding the tuples to each of element's projections to the underlying base relations. The fact that each tuple inserted in the database needs to be extended with some values for the attributes that have been projected out is not included in the translation algorithm that follows. How this operation is handled is dependent on the application. The following sections will only discuss how we have chosen to extend such tuples in our implementation.

Algorithm VO-CI: The input specifies a new XML element instance to be added to the view. The output is the set of database operations that implement the request.

- Isolate the dependency island
- For each tuple in each projection of the XML element, there are three possible cases:

  CASE 1: An identical tuple exists in the database. If the current relation belongs to the dependency island, reject the update; otherwise, do nothing.

  CASE 2: The new tuple does not match the key of any tuple in the underlying database relation. Perform an insertion.

  CASE 3: The new tuple matches the key of an existing tuple, but some values for nonkey attributes differ. If the current relation belongs to the dependency island, reject the update; otherwise, perform replacement of the existing tuple with the new tuple.

Following the insertion of the new XML element, we need to run a number of checks to preserve the global consistency of the database. For all the relations where tuples have been inserted by the algorithm VO-CI, the outside relations along

inverse ownership, inverse subset, and reference connections must be verified for proper dependencies. If no tuple satisfying the suitable dependency is found in any of those relations, one such tuple must be inserted, and the process must be applied recursively to that new insertion. Finally, for all the relations where tuples have been replaced, if referencing attributes are involved in the replacement, then the referenced relations must be checked for referential integrity.

6.1.3 Translation of Complete Replace Updates

As with replacements in relational views, replacements on XML elements are more difficult to handle than are complete insertions or deletions. Below we discuss the algorithm for translating complete replace updates.

Algorithm VO-CR: For each base relation, go to state REPLACING if the relation belongs to dependency island and go to state INSERTING otherwise.

- **STATE REPLACING**: Compare the old and new tuples in this projection.

  CASE R-1: The projections match exactly. Do nothing.

  CASE R-2: The projections differ but the keys match. Perform a replacement in this projection.

  CASE R-3: The projections differ and the keys differ. This case can happen only for those projections that are part of the dependency island. Perform a replacement in this projection.

- **STATE INSERTING**: Compare the old and the new tuples in this projection. There are four cases:

  CASE I-1: The keys match. Go to state REPLACING.

CASE I-2: The keys differ and the new tuple does not exist in the database relation. Insert new tuple in the database.

CASE I-3: The keys differ and the new tuple exists in the database. Do nothing.

CASE I-4: The keys differ and the new tuple is in the database but some attributes have conflicting values. Perform a replacement in this projection.

To maintain global consistency of the database, for referencing peninsulas, we must replace the foreign key of all the tuples that were refering to any of the modified tuples in the dependency island. Similarly, if a relation outside of the XML view is attached to the dependency island by an ownership or subset connection, the replacement has to be propagated to it. In all other cases, propagation of the change outside of the island will produce only checking and insertion operations to maintain global consistency.

6.2 Example Walkthrough

Let us try to explain the algorithms discussed in the previous sections by going over our running example. As discussed in Chapter 4, relations **Books** and **Authors** form a dependency island for the XML view of our running example whereas the relation **Prices** is a referencing peninsula having a reference connection with the relation **Books** in dependency the island. Having identified the dependency island and referencing peninsulas for our view, the application of our translation algorithms to the relational updates as a result of the XML update decomposition discussed in Chapter 5 becomes straightforward. Let us take a look at the *Delete* case through our example. The translation algorithms for *Insert* and *Replace* are dependent on actual data in the base relations and hence are not discussed here.

As discussed in chapter 5 one *Relational Update* is generated for each of relations **Books, Authors** and **Prices**. Following **Algorithm VO-CD** stated in Section

6.1 we know that since **Books** and **Authors** are in the dependency island we need to delete matching tuples from these relations. Also, since **Prices** is a referencing peninsula we need to delete any matching tuples from this relation or perform replacement of the keys. Since we choose to delete matching tuples from referencing peninsulas using triggers in our implementation, we know that we have taken care of deleting tuples from **Prices** relation. Thus applying the translation algorithm to our Delete relational updates results in explicit Delete relational updates as illustrated below on the relations **Books** and **Authors** and some implicit Delete relational update on the relation **Prices** via triggers.

```
Relational Update{
    updateType = Delete
    tableName = Books
    updateColumns = null
    whereClause = [books.bookid=authors.bookid AND
                   books.title=prices.title,
                   authors.first='Peter' AND authors.last='Naughton']
   }

Relational Update{
    updateType = Delete
    tableName = Authors
    updateColumns = null
    whereClause = [books.bookid=authors.bookid AND
                   books.title=prices.title,
                   authors.first='Peter' AND authors.last='Naughton']
   }
```

6.3 Extending Update Tuples

As mentioned while discussing the algorithms for translating complete XML updates in previous section, tuples to be inserted into the base relations should be extended with values for attributes that were projected out. Values of these attributes can be chosen arbitrarily from their respective sets of selecting values (which is the domain for non-selecting attributes). Each combination of values will represent a different algorithm for choosing values. There will be a unique algorithm to choose values

iff each attribute projected out (appearing in the database but not in the view) has a set of selecting values that is a singleton (has only one element). Thus since selection of values for attributes projected out depends on their application-specific value-domains, extending tuples for insertion into the base relation is application-dependent. For our implementation we chose a simple way (discussed next in the section) of extending tuples that serves the purpose while still maintaining the database integrity.

As discussed in Chapter 4 the relational database is queried for meta data about the base relations. During this process, information such as names of attributes, their data types, whether or not null values are allowed and if any attribute has the contraint to be unique is collected for each base table. This information about base tables is stored in a data structure called *Relational Table* shown below.

```
Relational Table{
    String tableName
    String[] attributeNames
    Integer[] attributeTypes  // Data types stored as constants
    Integer[] nullsAllowed    // Yes or No stored as constants
    String[] keys
    String[] foreignKeys
 }
```

When it comes to extending tuples, we already have the *Relational Update* data structure per base relation as defined in Chapter 5 which we compare with this *Relational Table* data structure of the corresponding relation. Any attributes present in the *Relational Table* but not in the *Relational Update* are added to the *Relational Update* with their new values generated following rules shown in Table 6.1.

Data Type	*Null Allowed*	*Generated Value*
Character	Yes	NULL
Character	No	X
String	Yes	NULL
String	No	NULL
Number	Yes	NULL
Number	No	99999
Date	Yes	NULL
Date	No	1/1/2000

Table 6.1: Data Type of Attributes and Values Generated for them.

While extending tuples for any base relation, join and select conditions from view definitions and select conditions from the XML update are taken into consideration to make sure that these conditions still hold. For example, if two base relations of XML view were joined on an attribute that was projected out then it is important to make sure that corresponding attributes in both base relations get the same value.

Chapter 7

SQL Update Generator and SQL Execution

7.1 SQL Generation

SQL generator is the simplest module of the *Update Manager* that generates SQL statements from the data structures representing relational updates as introduced in Chapter 5. Generated SQL updates are then executed in the underlying database engine. In our implementation, the *SQL Update Generator* generates SQL updates suitable for execution in the Oracle database engine. Let us look at SQL update generation through examples. Below, we depict a data structure representing a *Delete* update on the relation **Books** of our running example.

```
Relational Update{
    updateType = Delete
    tableName = Books
    updateColumns = null
    whereClause = [books.bookid=authors.bookid AND
                   books.title=prices.title,
                   authors.first='Peter' AND authors.last='Naughton']
   }
```

It is evident from the data structure shown above that all the information needed for generating the updates is available. It is simply a matter of generating strings representing SQL statements. These generated SQL statements should be suitable for being executed as is in the underlying database engine. The *SQL Update Generator* will generate the following SQL *Delete* statement from the data structure shown above.

```
DELETE FROM Books
       WHERE books.ROWID IN
       (SELECT books.ROWID FROM books, authors, prices
        WHERE books.bookid=authors.bookid AND
              books.title=prices.title AND
              authors.first='Peter' AND authors.last='Naughton')
```

The SQL statement shown above can now directly be submitted to the Oracle database engine. For the *Insert* and *Replace* relational updates corresponding *Insert* and *Update* SQL statements will be generated by the *SQL Update Generator*. For example, shown below are the data structure representing an *Insert* update on the relation **Books** and the corresponding SQL statement generated from the data structure. Let us assume that the attributes **bookid, title** and **year** have the values *3, 'My New Book'* and *2003* respectively.

```
Relational Update{
    updateType = Insert
    tableName = Books
    updateColumns = [bookid, title, year]
    whereClause = null
     }


INSERT INTO Books(bookid, title, year)
            VALUES(3, 'My New Book', '2003')
```

Similarly shown below is the data structure and SQL statement pair for a *Replace Update*. Let us assume that the following are old and new value pairs for the attributes **bookid, title** and **year** respectively : *(1,3), ('My Old Book', 'My New*

Book') and *('2000', '2003)*.

```
Relational Update{
    updateType = Replace
    tableName = Books
    updateColumns = [bookid, title, year]
    whereClause = [books.bookid=authors.bookid AND
                   books.title=prices.title,
                   books.title='My Old Book']
     }

UPDATE Books SET bookid=3, title='My New Book', year='2003'
       WHERE books.ROWID IN
       (SELECT books.ROWID from books, authors, prices
               WHERE books.bookid=authors.bookid AND
                     books.title=prices.title AND
                     books.title='My Old Book')
```

7.2 Update Execution

SQL statements generated by the *SQL Update Generator* are one by one sent to the Oracle database engine using a JDBC connection. Thus the time we measure for executing updates includes the overhead of client-server once per each update statement. In our implementation, the execution of updates is a three step process that includes initializing the database, executing update statements and restoring the database, as explained below:

1. **Initializing the database**: Initializing the database includes disabling any user constraints in the database. This is done because we do not know the order of execution of updates in the sense that whether or not child tuples will be inserted only after parent tuples have been inserted and such. To avoid being tripped over by such constraints, we first disable all the constraints on tables and then execute our updates. Initializing the database also includes creating a temporary table with row-ids and filling it up with the result of subquery that returns row-ids as seen in the case of *Delete* and *Update* SQL

statements above. And then the SQL statements are modified to select row-id from this temporary table rather than executing the subquery multiple times.

2. **Executing updates**: Once the database has been initialized, the SQL update statements are sent one by one to database engine using JDBC connection.

3. **Restoring the database**: Restoring the database includes enabling all the constraints that had been disabled during the initialization of the database. Also, the temporary table created while initializing the database is deleted from the database.

After execution of updates in the database engine we expect to have made all required changes to the relational database to bring the effect of the XML update that was originally issued on our virtual XML view. The effect will be seen when the XML view is recreated from the database using the same view definition.

Chapter 8

Experimental Evaluation

A comprehensive series of experiments was done to measure and compare the performance of individual modules of the *Rainbow* system as well as to measure performance of the *Update Manager* , including all of its individual modules of the *Update Manager*. Besides, we also ran experiments to measure the performance of different XML update types namely *Delete, Insert and Replace* in the system. We also ran a comparison between the performance of different types of XML updates. Experiments were also run to compare the reloading of XML documents with the incremental update of XML documents done by our system. Our results confirm that an incremental update of XML documents is always a better choice than reloading XML documents from scratch. The *Update Manager* code and also *Rainbow* system code were written in Java. The *Update Manager* translated update queries into SQL, and used JDBC to communicate with Oracle. All experiments were done on an 500 MHz Pentium with 328 MB of main memory with Oracle installed on a machine on other network and communication with Oracle was done over the internet. In order to ensure consistency, each experiment consisted of set of 5 runs. Thus, each graph point represents the average time for five runs.

8.1 Test Data

Experiments to test the performance of the *Update Manager* were done on relational data obtained as a result of loading XML documents into the Oracle database using XOR system [CR02] using the inline loading strategy. Our experiments were carried out with XML schemas ranging from 'Simple' to 'Very Complex' categorized depending upon the complexity of the schemas in terms of depth of the hierarchy of the document. XML views were defined on relational data as a result of loading XML documents conforming to these schemas. Figures 8.1, 8.2, 8.3 and 8.4 show XQueries we used to define 'Very Simple', 'Simple', 'Complex' and 'Very Complex' XML views respectively. The fanout of the XML documents loaded into the relational database was also taken into consideration while testing the system. The fanout of an XML document is the average number of children per element in the document. For example, a fanout of 800 would mean that the root of the document has 800 children and that on average then every child has 800 children of their own and so on. Since our system handles complete updates only we were primarily interested in the fanout of the root of the document.

```
<books>
FOR $book in document("default.xml")/book/Row
RETURN
  <book>
    <book_iid>$book/IID</book_iid>,
    <bookid>$book/book_bookid/text()</bookid>,
    <title>$book/book_title/text()</title>,
    <year>$book/book_year/text()</year>
  </book>
</books>
```

Figure 8.1: XQuery used to define a 'Very Simple' View

```
<authors>
FOR $authortuple IN document("default.xml")/author/row,
      $booktuple IN document("default.xml")/book/row
WHERE $authortuple/PID = $booktuple/IID
RETURN
  <author>
            <IID>$authortuple/IID/text()</IID>,
            <first>$authortuple/author_first/text()</first>,
            <last>$authortuple/author_last/text()</last>,
            <bookyear>$booktuple/book_year/text()</bookyear>,
            <booktitle>$booktuple/book_title/text()</booktitle>
  </author>
</authors>
```

Figure 8.2: XQuery used to define a 'Simple' View.

8.2 Performance of Rainbow Modules

In our first experiment we compare the performance of different functional modules of the *Rainbow System*. For the sake of simplicity, XML views were constructed

```
<books>
FOR $book in document("default.xml")/book/Row,
      $author in document("default.xml")/author/Row,
      $prices in document("default.xml")/prices/Row
WHERE $author/PID = $book/IID
   AND $prices/PID = $book/IID
RETURN
            <book>
               <IID>$book/IID/text()</IID>,
               <bookid>$book/book_bookid/text()</bookid>,
               <title>$book/book_title/text()</title>,
               <year>$book/book_year/text()</year>,
               <source>$prices/prices_source/text()</source>,
               <value>$prices/prices_value/text()</value>,
               <author>
                        <first>$author/author_first/text()</first>,
                        <last>$author/author_last/text()</last>
               </author>
            </book>
</books>
```

Figure 8.3: XQuery used to define a 'Complex' View.

```
<bib>
FOR $book in document("default.xml")/book/ROW,
      $aname in document("default.xml")/aname/ROW,
      $prices in document("default.xml")/prices/ROW
WHERE $book/book_author_IID = $aname/PID
   AND $book/IID = $prices/PID
RETURN
            <book>
               $book/IID,
               $book/PID,
               <bookid>$book/book_bookid/text()</bookid>,
               $book/book_title,
               $book/book_author_IID,
               <author>
                     $aname/IID,
                     <aname>$aname/aname_PCDATA/text()</aname>
               </author>,
               <prices>
                     $prices/IID,
                     <source>$prices/prices_source/text()</source>,
                     <currency>$prices/prices_currency/text()</currency>,
                     <value>$prices/prices_value/text()</value>
               </prices>,
               <publisher>
                     <pname>$book/book_publisher_pname/text()</pname>,
                     <location>$book/book_publisher_location/text()</location>
               </publisher>,
               <review>
                     $book/book_review/text()
               </review>
            </book>
</bib>
```

Figure 8.4: XQuery used to define a 'Very Complex' View.

in a way as to look alike the XML documnents that were loaded in the database. Different types of XML views were tested ranging from 'Simple' to 'Very Complex'. Figures 8.5, 8.6 and 8.7 depict the performance of individual modules in the case of *Delete, Insert* and *Replace* updates respectively. As seen from the figures, the generation of the view XAT and view analysis take more time than any other module in the system. Thus the time for each update can be significantly decreased if the view analysis could be done once at the time of view definition, and the view XAT is stored persistently thereafter. Updates issued at any point in time can then use the persistent XAT and the persistent data gathered by the analysis of the vieww. Figures 8.8, 8.9 and 8.10 show performance of functional modules in case of each kind of update. The *Generation and Optimization* module includes the generation of the user and view XAT, merging two XATs and rewriting of the merged XAT while the *Update Translation* module includes the decomposition of XML update into SQL updates, update translation using the algorithms discussed in Chapter 6 and execution of the SQL updates in the underlying relational database. Charts show that the percentage of time taken by *Update Translation* from the total time to do an update decreases with an increase in the complexity of the view

whereas the percentage of time taken by *Generation and Optimization* and *View Analysis* increases with an increase in the complexity of the XML view. The results are because of the fact that XATs become more complex with an increase in the complexity of the view and hence optimization also takes more time then. Again, as mentioned earlier, the time for each update can be significantly decreased by persistently storing the optimized XAT for the view and the information gathered during view analysis.

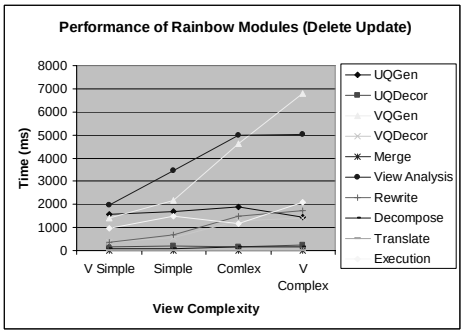

Figure 8.5: Performance of Rainbow Modules in case of Delete update, fixed fanout = 800.

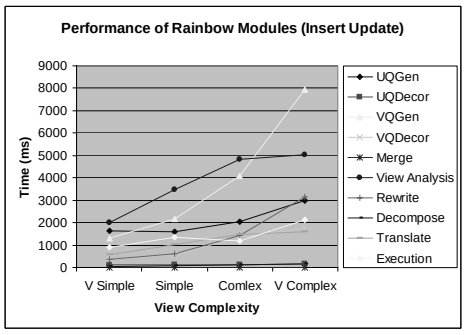

Figure 8.6: Performance of Rainbow Modules in case of Insert update, fixed fanout = 800.

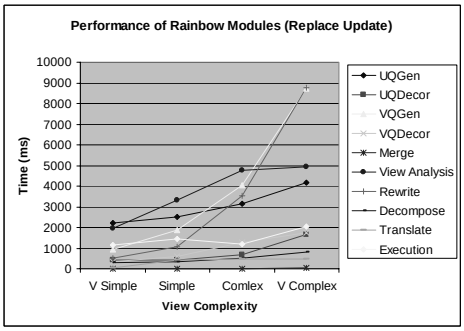

Figure 8.7: Performance of Rainbow Modules in case of Replace update, fixed fanout = 800.

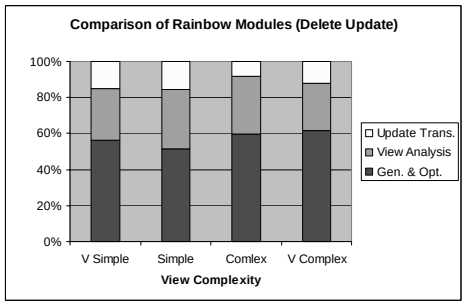

Figure 8.8: Performance of Rainbow Modules in case of Delete update, fixed fanout = 800.

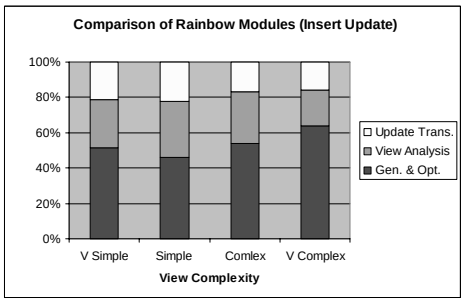

Figure 8.9: Performance of Rainbow Modules in case of Insert update, fixed fanout = 800.

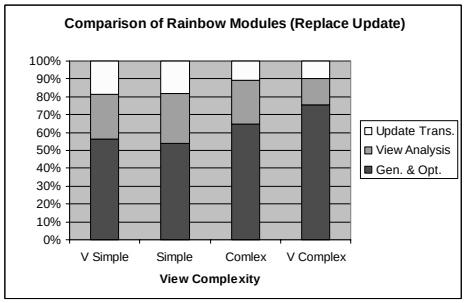

Figure 8.10: Performance of Rainbow Modules in case of Replace update, fixed fanout = 800.

8.3 Comparison between XML Update Types

Our next set of experiments was run to compare costs in terms of time for the different XML update types. Experiments were run to delete a complete element, insert a new complete element or replace an existing complete element with a new complete element. Results of this first experiment is shown in Figure 8.11. We used the definition of a *Complex* view. Documents with a fanout of 800 were loaded into database using the inline loading. In other words, every relation in the relational database that was used to form an XML view had 800 tuples each. The figure shows performance of individual modules of the *Rainbow System* in case of *Delete, Insert* and *Replace* XML updates. It is clear from the chart that the *Replace* update is the most expensive of all update types. This is because as the complexity of the update query increases from *Delete* to *Insert* to *Replace* so does the cost of generation of the user XAT and the rewriting of the merged XAT. Besides, the need to extend tuples in the case of the *Insert* and *Replace* updates makes the update translation time for these updates higher than in the case of the *Delete* update. Also, the time to find old values for relational tuples while decomposing an update in the case of *Replace* makes it more expensive than an *Insert* update.

The chart in Figure 8.12 gives another view of the fact that the cost of the incremental update increases from *Delete* to *Insert* to *Replace* update. It is clear that

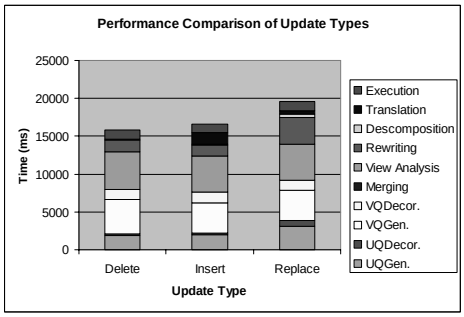

Figure 8.11: Performance Comparison of XML Update Types, *Complex* view, fixed fanout = 800.

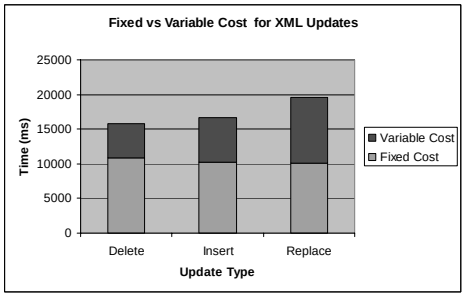

Figure 8.12: Fixed vs Variable Cost of XML Updates, *Complex* view, fixed fanout = 800.

the variable cost of incremental updates increases with an increase in the complexity of the update type. It is evident from the chart that the cost of each incremental update will reduce by at least 50 percentage if we made use of persistent storage to store the optimized view XAT and the data gathered during the view analysis. Namely, the fixed cost portion shown in Figure 8.12 would then be saved.

Figure 8.13 shows results of the experimental run to compare the cost of the XML update types as we increase the size of the XML documents being loaded into the database. In other words, increasing the size of the underlying database in terms of the number of tuples in each relational table that took part in the formation of the view. As expected, the cost of any update type is fairly constant with an increase in size of the underlying database. Each update type takes roughly 15 seconds which in this setup included both the fixed and variable costs of updates.

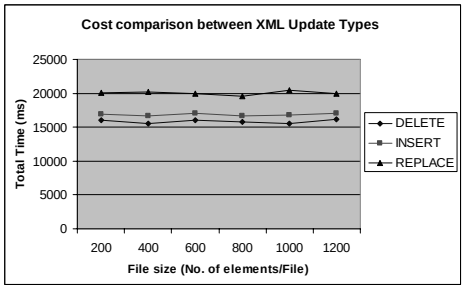

Figure 8.13: Cost Comparison of XML Update Types, *Complex* view, variable fanout.

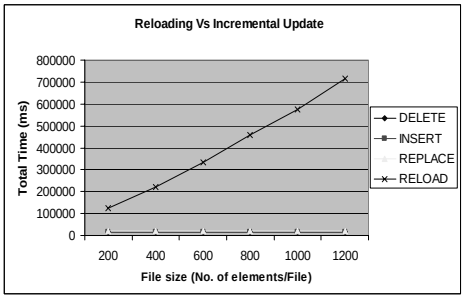

Figure 8.14: Reloading Vs Incremental Updates, *Complex* view, variable fanout.

8.4 Incremental Update Versus Reloading

The last set of experiments compares cost of incremental updates with complete reloading of the XML documents. Experimental results show that in almost all the cases incremental updating of XML documents stored in a relational database is better than reloading of XML documents. Figure 8.14 shows the time taken by different types of incremental updates and the time taken to reload XML documents. The cost of loading XML documents increases with an increase in the size of the XML documents being loaded into the relational database. The fact that the cost of incremental updates does not increase with an increase in the size of XML documents loaded into the database makes incremental update a better choice than the reloading of edited XML documents.

The chart shown in Figure 8.15 shows the time taken by the execution of updates for *Delete* updates. The execution cost includes the time taken to disable and enable database constraints on relevant relations before and after the execution of the delete statements. It also includes the cost of creating temporary tables used to store the results of subqueries referred to by delete statements. Thus with an increase in the size of the XML documents loaded into the relational database (in other words, with an increase in the size of relational tables) the cost of execution increases some what with an increase in the number of elements of an XML view that are expected to be affected by the XML update issued on this virtual XML view. Figure 8.16 shows that the total time taken by the *Delete* update is constant with an increase in the number of elements expected to be affected when compared to the time taken to reload the XML documents. It also shows that incremental update is a better choice even when more than 50 percent of the document is expected to be affected by the XML update.

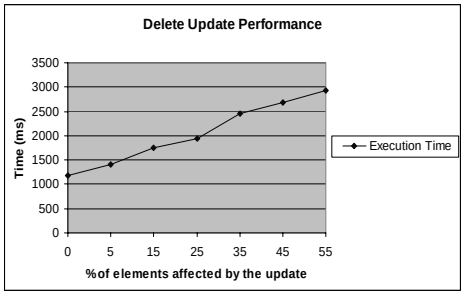

Figure 8.15: Execution time for Delete Update, *Complex* view, fixed fanout = 800.

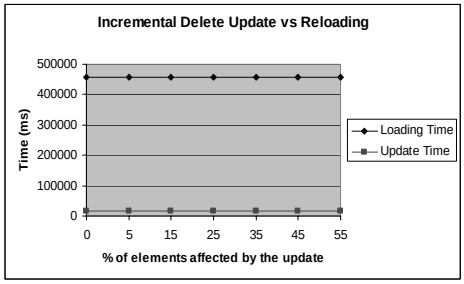

Figure 8.16: Reloading Vs Incremental Delete, *Complex* view, fixed fanout = 800.

Chapter 9

Related Work

While there has been a lot of work to solve the problem of mapping XML documents into and out of relational databases, less research has been done to date on supporting queries or updates on XML documents stored in a relational database. For XML views of relational data, work has appeared in the literature on querying XML views of relational data [SKS+01]. [SKS+01] proposes a method to translate XML queries to SQL in order to evaluate XML queries against XML data stored in a relational database but doesn't support any kind of updates that a user might want to express over the XML data.

Little work has been published on updating XML views of relational data [ZMLR01, TIHW01]. [ZMLR01] proposes update primitives and their implementation, which can be used to extend an XML query language, provided there is a way to break down the update queries into the update primitives. [TIHW01] specifies extensions to the XML query language XQuery to support data updates over XML documents. [TIHW01] proposes different trigger-based solutions to update the XML data stored in relational databases. They assume however that they already have the knowledge of the tables that are to be updated and other tables that might be affected as a result of these updates. Hence their solution assures completeness at the relational side. They do not focus on the issue of how to translate XML updates through XML

views into the correct SQL updates, which is an essential part for executing XML updates against XML data stored in relational databases and which is addressed by this thesis work. [GVD+01] introduces an algebraic framework for expressing and evaluating queries over XML data. It also defines equivalence rules for algebra rewrites, which can be used for the optimization of queries expressed over XML data. This work doesn't focus on the issue of querying XML views of the relational data.

On the other hand, a lot of work has been done on propagating updates on relational views to the underlying relational database [DB82, Mas84, Kel85, Kel86, A. 86]. Also some work has appeared in the literature on propagating updates on object-views of relational data to the underlying relational database [TBW91].

[DB82] formalizes the notion of correct translatability, and derives constraints on view definitions that ensure the existence of correct update mappings. Their theorems show that there are very few situations in which view updates are possible. They impose a major restriction that a view update should always be translated into the same type of update on the underlying relation which is not realistic in some cases.

[Mas84] gives a semantic approach to design a view update translator for relational database systems. Their translator consists of a translator body and four different types of *semantic ambiguity* solvers. They model a view definition a as tree with the view as the root and its base relations as the leaves. They translate an update issued against the root into updates against the leaves by applying a total of ten local translation rules and a deletion and an insertion modification rule recursively. The modification rules make it possible to update base relations through natural join views. The translation capability in their approach depends on

the solvers available to the translator body and the problem solving capability they offer. Due to ambiguities, the solvers may involve the end-users in resolving the ambiguities. [Mas84] relaxes the restriction imposed by [DB82] that a view update should always be translated into the same type of update on the base relations.

[Kel85, Kel86, A. 86] together consider the problem of updating databases through views composed of selections, projections and joins in general rather than having different translation rules for each type of join as done by [Mas84]. They present five criteria that translations must satisfy, and provide a list of templates for translation into database updates that satisfy the five criteria. They show that there cannot be any other translation that satisfies the five criteria. In addition, they propose that real world semantics can help in resolving ambiguity during translation of view updates and that these semantics can be obtained through dialogs at view definition time.

[TBW91] provide a mechanism for handling update operations on view objects of relational data. Because, a typical view object encompasses multiple relations, a view-object update request must be translated into several valid operations on the underlying relations. Building on the above approach to update relational views [Kel85, Kel86, A. 86] they introduce algorithms to enumerate all valid translations of the various update operations of view objects.

Chapter 10

Conclusions and Future Work

10.1 Summary

Though the XML data management community has recently focused on issues related to querying XML, it is clear that the problem of expressing updates over XML data will become prominent in the near future. An XML update language provides a general-purpose way to express changes to any data that can be respresented in XML - whether the data is actually stored within an XML repository or within a relational database with an XML view. In our work on updating XML views of relational data we have studied the problem of updating virtual XML views of relational data. These XML views built on top of relational data are virtual since the actual data persists within the relational database. Hence, whenever an update is issued on a virtual XML view, it has to be propagated to the underlying relational database. In order to update XML views of relational data we face problems introduced because of the structural differences between relational databases and XML documents besides the challenges already known from the traditional problem of updating relational views. Updating XML views of relational data is a problem more analogous to updating object-based views of relational data [TBW91]. In this work we have presented a solution to this problem by providing a set of XML up-

dates that can be used to do complete element updates to XML documents or views. We have also proposed ways to decompose and translate these XML updates into SQL updates on the underlying database. Our solution is based on Keller's et al's solution of updating object-based views of relational data [TBW91].

In our work, we present the XQuery language support for expressing XML updates. We also present a way of representing XML view definitions and XML updates using XML Algebra Trees (XATs) based on XML algebra designed by Xin et al [ZR02] and heuristics to optimize XQueries by rewriting XATs [ZPR02b]. In our solution to updating XML views we have shown how to decompose and translate given XML update into SQL updates and also a way of executing these translated updates on underlying database to bring out the effect intended by the issued XML update. We have presented a prototype to validate our theory and experimental results to support it. Our experimental results show that a complete *Delete* update is least expensive and a complete *Replace* update is the most expensive update. Incremental update of XML views is much faster than complete reload of XML documents in general. Also, incremental updates are a better choice than complete reloading of XML documents even when more than 50 percent of the document is expected to be affected by the update (Figure 8.16).

10.2 Contributions

Below we list the contributions of this thesis in the area of XML database management systems from our work on updating XML views of relational data.

1. **XQuery support for XML Updates**: We have presented a language for expressing XML updates in XQuery based on Halevy et al's [TIHW01] proposal on XQuery language support for XML updates. We present a format of expressing complete element updates like *Insert, Delete, Rename* and *Replace*.

We have a prototype of our system that supports XML updates expressed in this XQuery update language. The parser for XML updates is build on top of the XQuery parser prototype build by [A. 00].

2. **Framework for correct translation and propagation of XML updates to underlying relations**: With our solution to the problem of updating XML views of relational data we present a framework for the correct translation and propagation of complete XML updates to base relations. From literature we borrow the method applied for updating object-based views of relational data and have extended the solution to be applicable to our problem of updating XML views of relational data. We believe that our work presents first solution to do complete updates on XML views of relational data that addresses expressing XML updates on XML views, the decomposition and translation of XML updates into SQL updates on the underlying database, and the execution of SQL updates on base relations. Our solution is the first solution based on an XML algebra.

3. **Implementation of the system as a proof of concept**: We have a full implementation of our prototype called *Update Manager*. We have incorporated it into a working XML management system called *Rainbow*.

4. **Experimental evaluation**:

10.3 Future Work

While we believe that we have made a fairly comprehensive study of the problem of updating XML views of relational data, several potential areas for future work remain. These potential areas of future work are discussed below:

- **Batch updates**: Execution of relational updates one by one in the database engine surely increases the cost of each XML update. Since there are multi-

ple SQL updates per XML update on the XML view, batching of relational updates will surely increase the performance of the system. Thus from the performance view point investigating how relational updates can be batched per XML update and possibly even across XML updates batch updates becomes a potential area of future work.

- **Support for partial updates**: While support for complete updates is a substantial step towards supporting XML updates, it is evident that support for partial updates is important and this is the next immediate step in our area of research. This will enable users to carry out updates to only a part of the elements of the XML view rather than deleting, inserting or replacing a complete XML view element.

- **Typechecking for updates**: In our work we assume complete and correct updates which removes the need to typecheck updates the system is subjected to. Typechecking of updates is important as a candidate for future work.

- **Ordered XML updates**: Order preserving structure of XML documents definitely demands support for ordered querying and ordered updates. In our work we present the XQuery grammar and language support for ordered XML updates. Providing support for ordered XML updates is thus another important area of future work.

Bibliography

[A. 86] A. M. Keller. The Role of Semantics in Translating View Updates. *IEEE Transactions on Computers*, 19(1):63–73, 1986.

[A. 00] A. Sahuguet. Querying XML in the New Millennium. http://db.cis.upenn.edu/Kweelt, September 2000.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. In *International Journal on Digital Libraries, 1(1)*, pages 68–88, April 1997.

[Bar90] T. Barsalou. View Objects for Relational Databases. Technical Report STAN-CS-90-1310, Stanford University, Computer Science Department, Technical Report, 1990.

[CJS99] S. Cluet, S. Jacqmin, and J. Simeon. The New YATL: Design and Specifications. Technical report, INRIA, 1999.

[CKS+00] Michael J. Carey, Jerry Kiernan, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *The VLDB Journal*, pages 646–648, 2000.

[CR02] Steffen Christ and Elke A. Rundensteiner. X-Cube: A fleXible XML Mapping System Powered by XQuery. Technical Report WPI-CS-TR-02-18, Worcester Polytechnic Institute, 2002.

[CRF00] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *WebDB*, pages 53–62, 2000.

[DB82] U. Dayal and P. A. Bernstein. On The Correct Translation Of Update Operations on Relational Views. In *ACM Transactions on Database Systems*, 1982.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *Proceedings of the Eighth International World Wide Web Conference (WWW-8)*, 1999.

[DFS99] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 431–442, Philadephia, USA, June 1999.

[FK99] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data Using an RDBMS. In *Bulletin of the Technical Committee on Data Engineering*, pages 27–34, September 1999.

[FTS00] Mary Fernandez, Wang-Chiew Tan, and Dan Suciu. SilkRoute: Trading between Relations and XML. http://www.www9.org/w9cdrom/202/202.html, May 2000.

[GVD+01] Leonidas Galanis, Efstratios Viglas, David J. DeWitt, Jeffrey F. Naughton, and David Maier. Following the paths of xml data: An algebraic framework for xml query evaluation. http://www.cs.wisc.edu/niagara/papers/algebra.pdf, 2001.

[IBM] IBM. . www.ibm.com.

[Kel85] A. M. Keller. Algorithms for translating view updates to database updates for view involving selections, projections and joins. In *Proceedings of the Fourth Symposium on Principles of Database Systems, Portland, OR*, 1985.

[Kel86] A. M. Keller. Choosing a view update translator by dialog at view definition time. In *Proceedings of the Twelfth International Conference on Very Large Databases (VLDB), Kyoto, Japan*, pages 467–474, 1986.

[Mas84] Yoshifumi Masunaga. A Relational database view update translation mechanism. In *Proceedings of the Tenth International Conference on Very Large Databases (VLDB), Singapore*, 1984.

[Mic] Microsoft Incorporation. . www.microsoft.com.

[Ora] Oracle. . www.oracle.com.

[SHT+99] J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99), Edinburgh, Scotland, UK*, pages 302–314, September 1999.

[SKS+01] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene Shekita, Catalina Fan, and John Funderburk. Querying xml views of relational data. In *VLDB*, 2001.

[SSC84] C. H. Papadimitriou S. S. Cosmadakis. Updates of Relational Views. In *J. ACM, Vol. 31, No. 4*, 1984.

[TBW91] Niki Siambela Thierry Barasalou, A. M. Keller and Gio Wiederhold. Updating Relational Databases through Object-Based Views. In *ACM SIGMOD Conference on the Management of Data, Boulder, CO*, 1991.

[TIHW01] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S Weld. Updating XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA*, pages 413–424, May 2001.

[W3Ca] W3C. Guide to the W3C XML Specification ("XMLspec") DTD. http://www.w3.org/XML/1998/06/xmlspec-report.htm.

[W3Cb] W3C. XML Schema. http://www.w3.org/XML/Schema.

[W3C98a] W3C. Extensible Markup Language (XML) 1.0 – W3C Recommendation 10-February-1998. http://www.w3.org/TR/REC-xml, 1998.

[W3C98b] W3C. XML^{TM} . http://www.w3.org/XML, 1998.

[W3C00] W3C. XML Path Language (XPath)Version 1.0. W3C Recommendation. http://www.w3.org/TR/xpath.html, March 2000.

[W3C01] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, December 2001.

[WE80] G. Wiederhold and R. ElMasri. The structural model for database design. In *Entity Relationship Approach to Systems Analysis and Design, Holland*, pages 237–257, 1980.

[ZMC+02] Xin Zhang, Mukesh Mulchandani, Steffen Christ, Brian Murphy, and Elke A. Rundensteiner. Rainbow: Mapping-Driven XQuery Processing System. In *Demo Session Proceedings of SIGMOD'02*, page 614, 2002.

[ZMLR01] Xin Zhang, Gail Mitchell, Wang-Chien Lee, and Elke A. Rundensteiner. Clock: Synchronizing Internal Relational Storage with External XML Documents. In *RIDE-DM*, pages 111–118, April 2001.

[ZPR02a] Xin Zhang, Bradford Pielech, and Elke A. Rundensteiner. Honey, I Shrunk the XQuery! — An XML Algebra Optimization Approach. In *WIDM*, pages 15–22, Nov. 2002.

[ZPR02b] Xin Zhang, Bradford Pielech, and Elke A. Rundensteiner. XAT Optimization. Technical Report WPI-CS-TR-02-25, Worcester Polytechnic Institute, 2002.

[ZR01] Xin Zhang and Elke A. Rundensteiner. Rainbow: Bridge over Gap of XML and Relational. Technical report, Worcester Polytechnic Institute, 2001. in progress.

[ZR02] Xin Zhang and Elke A. Rundensteiner. XAT: XML Algebra for the Rainbow System. Technical Report WPI-CS-TR-02-24, Worcester Polytechnic Institute, July 2002.