WPI-CS-TR-TBA March 2003

Updating XQuery Views Published over Relational Data

by

Ling Wang
Mukesh Mulchandani and Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Updating XQuery Views Published over Relational Data

Ling Wang, Mukesh Mulchandani, Xin Zhang and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute Worcester, MA 01609
(lingw|mukesh|xinz|rundenst)@cs.wpi.edu

Abstract

Managing XML data using relational database systems, including query processing over virtual XML views that wrap relational sources, has been heavily studied in the last few years. However, in order for such systems to become viable full-featured XML data management systems, they must support not only queries, but also updates of such virtual XML views. XML view update is a challenging problem because of having to address the mismatch between the two distinct data models (the XML hierarchical view model and the flat relational base model) and between the two supported query and update languages (XQuery FLWU updates on the view versus SQL queries on the base).

In this paper, we characterize the XQuery view update problem by presenting a four dimensional problem taxonomy. We then identify the problem space we are tackling along this classification and discuss its updatability. We present our decomposition-based update strategy for solving this update XML problem. Our solution strategy ensures the consistency of the updated database by rejecting invalid updates, checking update translatability for valid updates through analyzing the view definition and view-to-relation mapping, and performing the update translation and propagation following correct translation rules. Our approach is general, and works for XML views over a relational database defined using a core subset of the XQuery language. As evidence of feasibility, we have implemented the proposed strategies for update decomposition and translation within the *Rainbow* query system. We also present our experimental study describing the performance characteristics of our update system in different scenarios.

Keywords: XML, update query processing, XQuery, XML to relational, view updatability

1 Introduction

Motivation. XML [5] has now become the standard for interchanging data between web applications because of its balance of power and flexibility. The database community has focused on combining the strengths of the XML data model with the maturity of relational database technology to provide both reliable persistent storage as well as flexible query processing and publishing, with XML management systems such as EXPERANTO [6], SilkRoute [10] and Rainbow [20]. Such systems typically offer support for XML view creation and for querying against such wrapper views in order to bridge relational databases with XML applications. However, in order for such systems to become viable XML data management systems, they must also support updates of virtual XML views.

The view-update problem in relational databases is a long-standing difficult issue. Much work has been done on defining what a correct translation entails [9] and how to eliminate ambiguity in translation [7, 3]. However, update operations in the XML context have not yet been addressed intensively nor well-supported. Updating of virtual 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 view versus SQL queries on the base). The update translation hence has to ensure consistency of the database and the view in the face of various possibly conflicting constraints in these different schemas and mappings, such as hierarchy constraints, domain constraints, cardinality constraints, and so on.

In this paper, we characterize the **XQuery view update** problem by presenting a four dimensional problem taxonomy. We then identify the problem space we are tackling along this classification and disscuss its updatability. We present our strategy framework named *Rainfall* for solving this update problem. Our update solution ensures the consistency of the updated database by rejecting invalid updates, checking update translatability for valid updates based on analyzing the view definition and view-to-relation mapping, and generating a correct update translation and propagation procedure by following correct translation rules. Due to there not being any standard update language, we have extended the XQuery grammar for supporting XML updates similar to [16]. We have implemented the proposed strategies for update decomposition and translation within the *Rainbow* XQuery system [20]. Experiments are also presented to compare update propagation with re-loading, assessing various performance characteristics of our update solution.

Contributions. In summary, we make the following contributions in this paper:

- We characterize and propose a classification of the XML view update problem, and describe its updatability.
- We provide a decomposition-based update translation algorithm, which correctly translates XQuery updates issued on XML views into a set of SQL-level updates using our *XML algebra tree (XAT)* of the *Rainbow* system, while at the same time maintaining integrity of the underlying relational database.

- We extend the *Rainbow Query engine* [20] to support the XQuery update grammar, and implement the proposed strategies within this system.
- We provide experimental results from the performance study of our proposed update strategies.

Outline. This paper is structured as follows. We briefly introduce the XML data model and XQuery update extension in Section 2. Section 3 describes the taxonomy of the view update problem. In Section 4, we present key concepts used by our update strategy. Section 5 describes our general update strategy, and proposes the decomposition algorithm. We describe our implementation and evaluation of these techniques in Section 6. In Section 7, we review related work while Section 8 concludes our work and points to future directions.

2 Background

We now review the technical background required for this work, in particular, the concept of XML views over relational data and a language extension for XQuery to support update operations.

XML and XQuery. XML (Extensible Markup Language) [5] is used both for defining document markup (and thus information modeling) and for data exchange. XML Schema [17] is a standardized syntax used to represent the structure of XML documents. Figures 1 and 2 respectively show our running example of an XML schema and document representing a book list from an online book store application.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchemax elementFormDefault="qualified">
  <xs:elementname="bib">
    <xs:complexType>
      <xs:sequence>
        <xs:elementname="book" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:elementname="bookid" type="xs:string" nillable="false"/>
              <xs:elementname="title" type="xs:string" nillable="false"/>
              <xs:elementname="author">
                <xs:complexType>
                  <xs:sequence>
                    <xs:elementname="aname" type="xs:string" maxOccurs="unbounded"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:elementname="publisher">
                <xs:complexType>
                  <xs:sequence>
                    <xs:elementname="pname" type="xs:string"/>
                    <xs:elementname="location" type="xs:string"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:elementname="review" type="xs:string" nillable="true"/>
            </xs:sequence>
            <xs:attributename="year" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 1: Example XML schema

```
<bib>
  <book year="1994">
    <bookid>98001</bookid>
    <title>TCP/IP Illustrated</title>
    <author>
      <aname>W. Stevens</aname>
    </author>
    <publisher>
      <pname>Addison-Wesley</pname>
      <location>San Francisco</location>
    </publisher>
    <review>
      One of the best books on TCP/IP.
    </review>
  </book>
  <book year="1992">
    <bookid>98002</bookid>
    <title>Programming in Unix</title>
    <author>
      <aname>Bram Stoker</aname>
    </author>
    <publisher>
      <pname>Addison-Wesley</pname>
      <location>Boston</location>
    </publisher>
    <review>
      A clear and detailed discussion of UNIX programming.
    </review>
  </book>
  <book year="2000">
    <bookid>98003</bookid>
    <title>Data on the Web</title>
    <author>
      <aname>Serge Abiteboul</aname>
      <aname>Peter Buneman</aname>
      <aname>Dan Suciu</aname>
    </author>
    <publisher>
      <pname>Morgan Kaufmann Publishers</pname>
      <location>New York</location>
    </publisher>
    <review>
      A very good discussion of semi-structured database
      systems and XML.
    </review>
  </book>
</bib>
```

Figure 2: Example XML data

Several XML query languages have been proposed by both the document and database communities. We focus our work on the XQuery standard [18], which is a powerful query language designed to query XML data.

book

IID	PID	bookid	title	author_IID	year
1.0	0.0	98001	TCP/IP Illustrated	11.0	1994
2.0	0.0	98002	Programming in Unix	21.0	1992
3.0	0.0	98003	Data on the Web	31.0	2000

author

IID	PID	aname
1.0	11.0	W.Stevens
2.0	21.0	Bram Stoker
1.0	31.0	Serge Abiteboul
2.0	31.0	Peter Buneman
3.0	31.0	Dan Suciu

review

IID	PID	review
1.0	1.0	One of the best books on TCP/IP.
2.0	2.0	A clear and detailed discussion of UNIX programming.
3.0	3.0	A very good discussion of semi-structured database systems and XML.

publisher

IID	pname	location
1.0	Addison-Wesley	San Francisco
2.0	Addison-Wesley	Boston
3.0	Morgan Kaufman publishers	New York

Legend:
- Primary Key
- Unique Key
- Non Key

Figure 3: Relations in database

```
CREATE TABLE book
    (IID VARCHAR2(20),
    PID VARCHAR2(20),
    bookid VARCHAR2(20),
    title VARCHAR2(100),
    author_IID VARCHAR2(20),
    year INTEGER,
    CONSTRAINTS AuthorUK UNIQUE (author_IID),
    CONSTRAINTS BookPK PRIMARYKEY (IID))

CREATE TABLE author
    (IID VARCHAR2(20),
    PID VARCHAR2(20),
    aname VARCHAR2(20),
    CONSTRAINTS AuthorPK PRIMARYKEY (IID,PID),
    FOREIGNKEY (PID) REFERENCES Book (author_IID))

CREATE TABLE publisher
    (IID VARCHAR2(20),
    pname VARCHAR2(256),
    location VARCHAR2(256),
    CONSTRAINTS PublisherPK PRIMARYKEY (IID),
    FOREIGNKEY (IID) REFERENCES Book (IID))

CREATE TABLE review
    (IID VARCHAR2(20),
    PID VARCHAR2(20),
    review VARCHAR2(2000),
    CONSTRAINTS ReviewPK PRIMARYKEY (IID),
    FOREIGNKEY (PID) REFERENCES Book (IID))
```

Figure 4: Database schema of Figure 3

```
<DB>
  <book>
    <row>
      <IID>1.0</IID>
      <PID>0.0</PID>
      <bookid>98001</bookid>
      <title>TCP/IP Illustrated</title>
      <author_IID>11.0</author_IID>
      <year>1994</year>
    </row>...
  </book>
  <author>
    <row>
      <IID>1.0</IID>
      <PID>11.0</PID>
      <aname>W. Stevens</aname>
    </row>...
  </author>
  <publisher>
    </row>
      <IID>1.0</IID>
      <pname>Addison-Wesley</pname>
      <location> SanFrancisco</location>
    <row>...
  </publisher>
  <review>
    <row>
      <IID>1.0</IID>
      <PID>1.0</PID>
      <review>
        One of the best books on TCP/IP.
      </review>
    </row>...
  </review>
</DB>
```

Figure 5: Default XML view of database shown in Figure 3

```
FOR $binding1 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 6: XQuery update extension

It uses a **FLWR** expression, which includes *FOR...LET...WHERE...RETURN* clauses.

XQuery Views of Relational Data. In this section we describe the mechanism used in several recent systems such as XPERANTO [6], SilkRoute [10] and Rainbow [20] to define XML views over relational data. An XML query language, such as XQuery [18], can be used both to define such views and also to query them.

The basic XML view, called **Default XML View**, represents each table in the relational database as one XML element and each of its tuples as subelements of this table element. Figures 3 and 4 show an example relational database generated from the XML schema and data of our running example in Figures 1 and 2 using a shared inlining loading strategy [15]. Figure 5 depicts the default XML view of this database. The *default XML view* is a one-to-one mapping to bridge the gap between the two heterogeneous data models, that is the XML and relational model.

A default XML view explicitly exposes the tables and their structure to the end users. However, end users often want to deal with an application specific view of the data. Thus we must extract and then restructure the data. For this reason, many XML data management systems provide a facility to define a user-specific view on

```
<bib>
FOR $book in document("default.xml")/book/row
RETURN{
    <book year=$book/year/text()>
        <bookid>$book/bookid/text()</bookid>,
        <title>$book/title/text()</title>,
        <author>
            FOR $aname in document("default.xml")/author/row
            WHERE $book/author_IID = $aname/PID
            RETURN{
                <aname>$aname/aname/text()</aname>}
        </author>,
        FOR $publisher in document("default.xml")/publisher/row
        WHERE $book/IID = $publisher/IID
        RETURN{
            <publisher>
                <pname>$publisher/pname/text()</pname>,
                <location>$publisher/location/text()</location>
            </publisher>},
        FOR $review in document("default.xml")/review/row
        WHERE $book/IID = $review/PID
        RETURN{
            <review>
                $review/review/text()
            </review>}
    </book>
}
</bib>
```

Figure 7: Virtual XQuery view over default XML view shown in Figure 5 producing the XML data in Figure 2

```
FOR $root in document("view.xml")
UPDATE $root {
  INSERT <book year="1995">
            <bookid>98004</bookid>,
            <title>"Languages and Machines"</title>,
            <author>
              <aname>"Thomas A. Sudkamp"</aname>
            </author>,
            <publisher>
              <pname>"Addison Wesley Longman, Inc."</pname>,
              <location>"Boston"</location>
            </publisher>,
            <review>
              "An Introduction to the theory of Computer Science"
            </review>
          </book>
}
```

Figure 8: Insert update on XQuery view shown in Figure 7

top of this simplistic default XML view, called a **Virtual View**. Such a virtual view can be specified using a view query in the XQuery language, then called a **View Query**. Figure 7 shows the view query which defines a virtual view with exactly the same schema as that of the original XML document in Figure 2.

XQuery Updates. No update language for XML has yet been standarized. For our work, we thus adopt an extension of the XQuery language syntax with update operations, that closely follows the proposal in [16]. XQuery is extended with a **FLWU** expression that includes *FOR...LET...WHERE...UPDATE* clauses (Figure 6). Within the *UPDATE* clause, a sequence of sub-operations can be specified to be applied to the variable bound by the update statement. Figure 8 shows an example *Insert* update, which inserts a new *book* element into the (virtual) view defined in Figure 7. As result, the database will now contain four books instead of three.

3 Problem Definition

We first characterize the **XQuery view update problem** along several problem dimensions, then define the problem within this space that we propose to tackle and discuss its updatability.

3.1 Characterization of Problem Space

There are four dimensions along which to study the XML view update problem: **Information Dimension:** What information is required for solving the view update problem? **Instance Dimension:** Does the view update strategy work for all instances of relational databases or only some instances? **Language Dimension:**

Expressiveness of the language used in the view definition, such as XQuery or subset of XQuery. **Modification Dimension:** What modifications can the view update strategy handle?

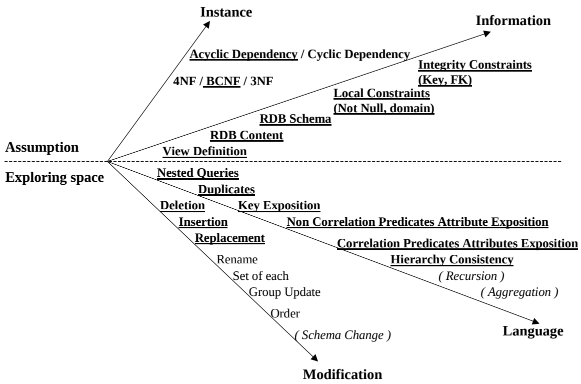

Figure 9: Characterization of XML view problem space

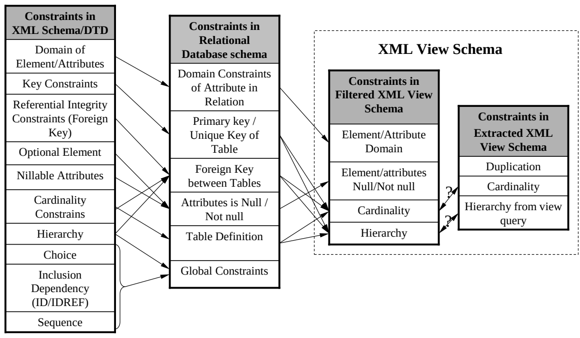

Figure 10: Comparison of constraints of XML schema, relational database schema and XML view schema

As shown in Figure 9, we characterize the XML view update problem based on these four dimensions. Our assumptions for the *Instance* and *Information* dimensions are indicated by the underlined items on these dimensions. Our discussion will explore the *Modification* and *Language* dimensions. The problem space we tackle is defined as: (1) Information dimension: Complete information requirement, that is, we know all information about the underlying relations such as schema, content, constraints and also the virtual view definition. (2) Instance dimension: The relational database is in the BCNF form without any cyclic dependency. This is because we are using functional dependencies of relations and referential integrity constraints to determine the update propagation strategy. (3) Language dimension: No aggregation and recursion in the view definition. These operations make views non-updatable, as enunciated in [11]. (4) Modification dimension: We only focus on content updates and not schema changes, since the former are more common and thus of immediate importance. In Figure 9, the problem space we target is identified by the underlined items.

3.2 Correctness Criteria for Update Translatability

Our problem can be defined as follows: given a relational database and an XQuery view definition over it, how should the system decide if an update against the view can be translated into corresponding updates against the underlying relational database without violating any consistency. And, if it translatable, then how would this translation to be done. A consistent translation has to satisfy three criteria. First, the update must be consistent with the *XML view schema*. Second, the translated update against the relational database must comply with the relational schema and its constraints. Third, the result of XML view reconstruction after performing the translated update on the underlying relational database must be exactly what the user had intended to request, that is, without side-effects.

As shown in Figure 10, an **XML View Schema** is a combination of a *Filtered XML View Schema* and an

Extracted XML View Schema. A **Filtered XML View Schema**, also called *FSchema*, models the implicit constraints in relations referenced by the view query, which include domain constraints, cardinality constraints, Null constraints and hierarchical constraints. An **Extracted XML View Schema**, also called *ESchema*, consists of constraints within the *view query* semantics, which include cardinality constraints, hierarchical constraints and duplication constraints. We will discuss how to extract those two schemas in Section 3.3.2. As shown by question marks in Figure 10, possible conflicts, such as a conflict between their hierarchies, may exist between two schemas and thus have to be detected as early as possible. Various constraints exist and are possibly not exposed along the two mappings, that is, the loading mapping from the *XML Schema* to the *Relational Database Schema* and the view query mapping from the *Relational Database Schema* to the *FSchema*. Only in the *ESchema*, new constraints are being added, such as duplicate representation of relational data in the XML view. These new constraints are not guaranteed to be enforced in the relational database, hence may cause problems in the update translation.

Before studying view updatability, we first define the update operations on nested XML view elements. For this purpose, we introduce a distinction between complete and partial updates issued on a virtual view by extending these terms from [4]. Given a view query Q defined with several nested FLWR expressions, denoted as Q=$E_0(E_1(...(E_n)))$, where R_i is the RETURN clause in the expression E_i, then the element structure defined in R_0 is called the *view-element*. A **Complete Insertion** adds to while a **Complete Deletion** removes from the XML view a complete view-element instance. A **Complete Replacement** replaces a view-element instance with a complete replacing instance. A **Complete Update** is either a complete insertion, a complete deletion or a complete replacement. A **Partial Insertion** adds to while a **Partial Deletion** removes from the XML view a complete *sub-element* or *attribute* of a view-element instance. A **Partial Replacement** replaces a sub-element or attribute of a view-element instance with a replacing instance. A **Partial Update** is either a partial insertion, a partial deletion or a partial replacement.

In order to permit all possible changes but only in their simplest forms, we require that all candidate update translations satisfy the five criteria as stated in [11]: (1) **No side effects.** Only the desired update is performed on the view. (2) **One step changes.** Each database tuple is affected by at most one step of the translation for any single view update request. (3) **Minimal changes.** There is no valid translation that implements the request by performing only a proper subset of database requests. (4) **Replacement cannot be simplified.** That is, we always pick the simpliest replace operation, e.g, a database replacement that does not involve changing the key is simpler than one where the key changes. (5) **No insert-delete pairs.** We do not allow candidate translations to include both deletions and insertions on the same tuple of the same relation. Instead they must be converted into replacements, which we consider simpler.

Definition 1 *Given a relational database S_r, an XML view V defined by a view query Q over S_r, denoted as $V = Q(S_r)$. Let U be a complete or partial update against V. If $\exists T_u$ is a set of SQL updates that satisfy the five criteria defined above such that $U(V) = Q(T_u(S_r))$, we say U is* **translatable** *for V and V is* **updatable** *by U.*

We now discuss the XML view update problem within our problem space shown in Figure 9.

3.3 Publishing-based View Update Problem

3.3.1 Problem Description

The general update problem within our problem space that we are tackling in this paper is named **Publishing-based View Update Problem (PUP)**. It updates freely constructed XQuery views defined over a relational database. Hence all items along the *language dimension* in Figure 9, named as *Exposition features*, have to be taken into account in this problem.

3.3.2 Exposition Features

(1) Exposition Completeness. In a database in BCNF, primary key exposition is a decisive factor entitling a view user to make modifications on a relation. In addition, for insert update operations, a partial tuple has to be extended to become a complete tuple in order to be insertable into a relation. The primary key attributes have to be specified explictly by an update operation instead of taking the default values, thus they have to be exposed. If the database is a product of having first been loaded in from XML document instead of being native relational schema, then we can get the automatically generated attributes as primary key, named *generated key*. A generated key can be re-generated by system in update translation instead of having to be specified by the user. A *non-correlation predicate* has the form *$VariableBinding θ C*, where $\theta \in \{=, \neq, <, \leq, >, \geq\}$. Assuming the mapping attribute of the $variableBinding in the relational database is a, and C is a constant in the domain of a, then a must be exposed in the view, that is, $variableBinding must appear at least once in the RETURN clauses of the view query. For example, if the view query has a non-correlation predicate $book/title="The six city", then in the RETURN clause, $book/title has to be exposed. Otherwise, in the insert update, you would not know if the inserted element is valid for the view or not. A *correlation predicate* has the form of *$VariableBinding1 θ $VariableBinding2*. For the same reason, the mapping attributes corresponding to all variable bindings have to be exposed.

Definition 2 *Let Q be a view query, $S_r = \{R_1, R_2, ..., R_n\}$ be the set of relations referenced by Q and V=Q(S_r) denote the XML view defined by Q. Then Q is called a* **Complete Exposition** *iff the following hold: (1) All primary key attributes of R_i (1≤i≤n) are exposed in V. (2) All attributes involved in any non-correlation predicate in Q are exposed in V. (3) All attributes in any correlation predicate in Q are exposed in V.*

(2) Construction Consistency. The consistency of the view construction is another important factor in determing view updatability, since the *XML view schema* is a combination of two possibly conflicting schemas, namely the *FSchema* and *ESchema* as described in Section 3.2. Both of them include hierarchical constraints, which may cause problems later in the update translation. So we have to detect hierarchical inconsistency between the two schemas as early as possible. For this purpose, we now first describe the notion of hierarchy in both the XML and the relational context. Then we use it respectively to represent the hierarchical constraints in *FSchema* and *ESchema* by two graphs named *View Relationship Graph (VRG)* and *View Schema Graph (VSG)*. Both are a type of *Annotated Structured Graph* defined in Def 4. Detection of other constraints conflicts is straight-forward.

Definition 3 *We define the implied* **Hierarchy in the Relational Model** *as follows:*

(1) Table vs. attributes. Given a relation R(N,A), where N is the name of the relation and $A = \{a_i | 1 \leq i \leq n\}$ *are the attributes of R, then we say N is the* **parent** *of* a_i *(* $1 \leq i \leq n$ *).*

(2) Key vs. foreign key. Given two relations $R_i(N_i,A_i)$ *and* $R_j(N_j,A_j)$*, with foreign constraints defined as* $K(R_i) \leftarrow FK(R_j)$*, then we say that* N_i *is the* **parent** *of* N_j*.*

Definition 4 *An* **Annotated Structure Graph** *is a graph G(V,E), where V is a finite set of vertices and E is a binary relation on V. Each edge e(v1,v2) ∈ E, with the direction from v1 to v2, represents that v1 is the parent of v2.*

Vertices in *Annotated Structure Graph* are further divided into **leaf nodes** and **internal nodes**, represented by a straight box and a dash box respectivly. The **Closure** of node n in *Annotated Structure Graph* denoted by n^+, is defined as all the leaf nodes which have a path started from n .

We then use an ASG graph to describe the hierarchical constraints in the *ESchema* extracted from the view query. The ASG for this purpose is named the **View Schema Graph (VSG)**, which is a graph G_s(V,E) generated as follows: (1) The set V consists of all elements and attributes exposed in the XQuery RETURN clause. Each leaf vertex v ∈ V is annotated by its XPath and each internal node by its XML tag name. (2) Each edge e(v1,v2) ∈ E represents either an element to sub-element relationship or an element to attribute relationship in the RETURN clause.

Since an XML view corresponds to a hierarchical structure, VSG will be a tree. We can further use an *Annotated Structured Graph* to describe the hierarchy constraints in *FSchema* extracted from the underlying relational database using Definition 3. The Annotated Structured Graph is called a **View Relationship Graph (VRG)** correspondingly. Its generation steps are the following: (1) Take all leaf nodes of *View Schema Graph* as leaves of *View Relationship Graph.* (2) Map leaf nodes back to their corresponding relational tables and attributes. If it is a primary key attribute in relational tables, then name it a *key node* represented by a double circle. (3) Extract the hierarchy in the relational database according to Definition 3. Specially, we define that the *key node* in *View Relationship Graph* has same closure with its parent node.

Figures 11 and 12 are two example view queries defined over our database shown in Figure 3. Figures 13 and 14 depict their corresponding VSGs. Their VRGs are the same, as shown in Figure 15.

```
<bib>
FOR $book IN document("default.xml")/book/row,
        $publisher IN document("default.xml")/publisher/row
WHERE $book/IID = $publisher/IID
RETURN
    <book>
        $book/IID,
        $book/bookid,
        $book/title,
        <publisher_info>
            $publisher/IID,
            $publisher/pname
        </publisher_info>
    </book>
</bib>
```

```
<bib>
      <book>
            <IID>1.0</IID>
            <bookid>98001</bookid>
            <title>TCP/IP Illustrated</title>
            <publisher_info>
                  <IID>1.0</IID>
                  <pname> Addison-Wesley</pname>
            </publisher_info>
      </book>
      ...
</bib>
```

Figure 11: View query example 1

```
<bib>
FOR $book IN document("default.xml")/book/row,
        $publlisher IN document("default.xml")/publisher/row
WHERE $book/IID = $publisher/IID
RETURN
      <publisher_info>
            $publisher/IID,
            $publisher/pname,
            <book>
                  $book/IID,
                  $book/bookid,
                  $book/title
            </book>
       </publisher_info>
</bib>
```

```
<bib>
      <publisher_info>
            <IID>1.0<IID>
            <pname> Addison-Wesley </pname>
            <book>
                  <IID>1.0</IID>
                  <bookid>98001</bookid>
                  <title>TCP/IP Illustrated</title>
            </book>
      </publisher_info>
      ...
</bib>
```

Figure 12: View query example 2

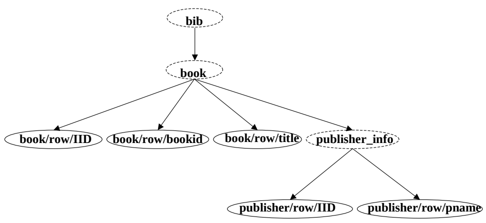

Figure 13: VSG for example 1

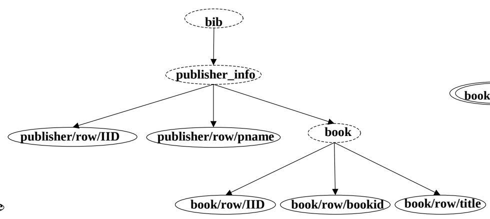

Figure 14: VSG for example 2

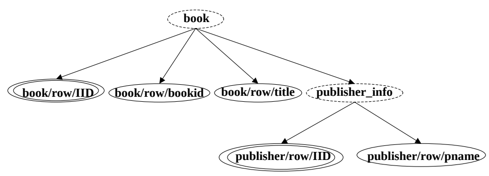

Figure 15: VRG for both queries in example 1 and 2

The VSG and VRG of the same view can be different, since the hierarchy in *FSchema* and *ESchema* can be different. We distinguish between *consistent* and *inconsistent* nodes to describe if an update in VSG will have the same effect in VRG or not. It hence identifies if the hierarchy in ESchema complies with FSchema.

Definition 5 *Given a view query Q with VSG G_s(V1,E1) and VRG G_r(V2,E2). Let $n \in V1$ is an internal node, with its closure in G_s denoted by $n^+ = \{v_{k1}, v_{k2}, ..., v_{kn}\}$, let v_{ki}^+ is the closure of v_{ki} in G_r. Then we say n is a* **consistent node** *iff $n^+ = v_{k1}^+ \cup v_{k2}^+ \cup ... \cup v_{kn}^+$.*

A view query Q is called a **Consistent Construction** iff all the VSG internal nodes are consistent nodes, otherwise, Q is called an **Inconsistent Construction**.

We will find that in the VSG shown in Figure 14, $(book)^+ = \{(book/row/IID), (book/row/bookid), (book/row/title)\}$, while in VRG shown in Figure 15, $\{(book/row/IID)^+ \cup (book/row/bookid)^+ \cup (book/row/title)^+\} = \{(book/row/IID)$, $(book/row/bookid)$, $(book/row/title)$, $(publisher/row/IID)$, $(publisher/row/pname)\}$. They are not equal. Hence *book* is not a consistent node. The view query in Figure 12 is an inconsistent construction, while the view query in Figure 11 is a consistent construction.

Definition 6 *Let Q be a view query defined over a relational database with VRG G_r(V,E), we say Q is* **Duplication Free** *iff the annotations for any two leaves $v_i \in V$ and $v_j \in V$ are not the same.*

3.3.3 View Updatability

Given the above exposition definition, we now divide views into different cases, each with different updatability for *valid updates*. A **valid update** is a *complete update* or *partial update* that never invalidates any semantic constraints in *ESchema*, and also does not violate any *domain constraints* and *Null constraints* in *FSchema*. An invalid update will always be rejected by our proposed updatability checking step. Figure 16 shows the updatability for our problem.

View Features			Update Type	
Exposition complete	Consistency	No Duplication	Complete Update	Partial Update
Y	Y	Y	Y	Y
Y	Y	N	Y	Case 1
Y	N	Y	Y	Case 2
Y	N	N	Y	Case 3
N	Y	Y	N	N
N	Y	N	N	N
N	N	Y	N	N
N	N	N	N	N

Figure 16: View updatability for publishing-based view update problem

	Ownership connection	Subset connection	Referencing connection
$X1 \xleftarrow{FK} X2$	X1=PK/UK(R1), X2⊂PK(R2)	X1=PK/UK(R1), X2=PK(R2)	X1=PK/UK(R1), X2⊆NK/UK(R2)
Cardinality	1:n	1:[0,1]	1:n
Representation	R1 •—— R2	R1 ◄—— R2	R1 ←—— R2

Figure 17: Comparison of connections

Observation 1 *An XQuery view defined by a query with an incomplete exposition is not updatable.*

Observation 2 *An XML view V defined by an XQuery Q with a complete exposition has the following updatability:(1) All complete updates are translatable. (2) For partial updates, we distinguish several cases:*

Case0 *If Q is a consistent construction by Definition 5 and duplication free by Definition 6, the generated views are called Well-formed Views and all types of partial updates are translatable.*

Case1: *If Q is a consistent construction, but not duplication free, any update touching only one part of the duplication is not translatable. For any update touching all parts of duplication, deletion is translatable, while insertion has to be checked in order to make sure we have a consistent value for the duplicates.*

Case2: *If Q is duplication free but has an inconsistent construction, then all types of updates that are applied on any sub-tree rooted at an inconsistent node of VSG are not translatable.*

Case3: *If Q is an inconsistent construction and also not duplication free, then it is a combination of case1 and case2. The updatability is in their intersection.*

3.4 A Special Case: Round-Trip based View Update Problem

The Round-Trip View Update Problem is a special case for our general view update problem. Here we assume that we have an original XML schema and a valid XML document. By using a suitable loading algorithm, such as inlining [15], edge or universal [8], accompanied with a constraint-preserving mapping as described in [13], we built a relational database, whose schema captures all the corresponding XML structural and semantic information. We call this database schema as a **Structured Database Schema**. We define an XML view on this well-structured database using XQuery, which constructs an XML view with a schema identical as had just been supplied as input to this round-trip mapping. We call this special-purpose view query an *extraction query*. Updating of a database through this view is named the *round-trip view update problem*. The implication of this special problem is that the view demonstrates the same structure and semantics as the original XML document which will also all exactly be captured by the relational database. Hence, this special problem falls into case 0 of observation 2, that is a *well-formed view*. Any *valid update* on a view in this problem is hence always translatable.

4 Update Fundamentals

4.1 Structural Model

A structural model of a relational database is used to describe relations and their relationships. The most common relationships between relations are foreign keys. Because of the motivation provided by [4], we have chosen a similar convention to divide foreign key connections into three different types named *ownership connection*, *subset connection* and *reference connection*.

Definition 7 *Given two relations R_1 and R_2 and two subsets of attributes X_1 of R_1 and X_2 of R_2, let PK, UK and NK denote the primary key, unique key and nonkey attributes of relation, connections between R1 and R2 can be divided as:*

An ownership connection *from X_2 of R_2 to X_1 of R_1 specified by: (1) $X_1 = UK(R_1) \vee X_1 = PK(R_1)$, (2) $X_2 \subset PK(R_2)$, and (3) X_2 is a foreign key in R_2, which refers to the key X_1 in R_1.*

A subset connection *from X_2 of R_2 to X_1 of R_1 specified by: (1) $X_1 = UK(R_1) \vee X_1 = PK(R_1)$, (2) $X_2 = PK(R_2)$, and (3) X_2 is a foreign key in R_2, which refers to the key X_1 in R_1.*

A reference connection *from X_2 of R_2 to X_1 of R_1 specified by: (1) $X_1 = UK(R_1) \vee X_1 = PK(R_1)$, (2) $X_2 \subseteq NK(R_1) \vee X_1 \subseteq UK(R_2)$, and (3) X_2 is a foreign key in R_2, which refers to the key X_1 in R_1.*

If there is a connection from R_2 to R_1, then we also say that R_1 has an *incoming connection*, while R_2 has an *out-going connection*. The cardinality between R_1 and R_2 in ownership and reference connection is *1:n*, while *1:[0,1]* in subset connection. Figure 17 compares those three different connections. Their representations in terms of graph edges are also depicted. Note that *m:n* relationships, while not modeled directly in the structural model, can be represented using combinations of connections. If there are several connections between two relations, we follow the precedence subset, ownership and reference connection.

4.2 Relationship Graph for Relational Databases

We first divide relations into three types by their visibilities with respect to the view definition.

Definition 8 *Given a relational database* $S_r = \{R_1, R_2, ..., R_n\}$*, an XML view V defined by an XQuery Q, the* **Core Relations** *of V, denoted by* $CR(V)$ *are a set of relations referenced by Q, that is V=Q(CR(V)). The* **Extended Relations** *of V, denoted by ER(V) specified by:* $\forall R_i \in ER(V)(1 \leq i \leq n)$ *(1)* $(R_i \in S_r) \wedge (R_i \notin CR(V))$ *and (2)* $\exists R_j \in CR(V)$ $(1 \leq j \leq n)$ *such that* R_i *directly connected with* R_j *by subset, ownership or reference connection. The* **Involved Relations** *of V, denoted by IR(V), specified by:* $\forall R_i \in IR(V)(1 \leq i \leq n)$ *(1)* $(R_i \in S_r) \wedge (R_i \notin (CR(V) \cup ER(V)))$ *and (2)* $\exists R_j \in (ER(V) \cup IR(V))$ $(1 \leq j \leq n, i \neq j)$ *such that* R_i *connected with* R_j *by subset, ownership or reference connection (denotation of an indirect connection with core relations).*

Definition 9 *Given an XML view V defined over a relational database* $S_r = \{R_1, R_2, ..., R_n\}$*, the* **XML View Mapping Graph (VMG)***, denoted by* $G_m(V, E)$*, is a maximum directed graph defined as follows: (1) A node* $v \in V$ *is a relation in CR(V), ER(V) or IR(V), and (2) An edge* $e(v_1, v_2) \in E$ *from* v_1 *to* v_2 *is a connection C between the corresponding relations* R_i *and* R_j $(1 \leq i, j \leq n)$ *as defined in Section 4.1.*

Based on our general assumption in Section 3.1, VMG would be acylic, that is, it would have the shape of a *Directed Acylic Graph*. For simplicity reasons, we would like it to be a forest, that is a collection of trees. By replicating subtrees rooted at vertices having multiple incoming edges, we can convert such an acylic-DAG into a forest. VMG indicates all the relations which directly or indirectly are involved in the view query and their relationships. Implicitly, this indicates how many relations could be affected by a view update and how they could be affected. As an example, the VMG of the XQuery view defined by Figure 3 is described in Figure 18.

According to connection categories between relations, we classify the different connections in the relational database into categories using similar notation with [4].

Definition 10 *Given a relation R in VMG* G_m*=(V,E), where* $V = \{R_1, R_2, ..., R_n\}$*, the* **Dependency Island** *of* **R***, represented as DI(R), is the maximal sub-tree which includes all directed paths starting from the root relation* **R** *and contains exclusively incoming ownership and subset connections. The* **Referencing Peninsula** *of* **R***, denoted by RP(R), is a relation* $R_i \in V$ $(1 \leq i \leq n)$ *directly connected to any relation* $R_k \in DI(R)$ $(1 \leq k \leq n)$ *by an incoming reference connection. The* **Referenced Continent** *of* **R***, denoted by RC(R), is a maximal sub-tree of* G_m *which includes all directed paths starting from the root relation* **R***, including all out-going connections.*

The rationale behind the dependency island is that all relations in the dependency island belong to the same entity - the entity that is dependent on the root relation. As a result, any update operation on the view element should have a consistent effect throughout the components of that node's dependency island. Referencing peninsulas, on the other hand, must be identified because of the constraints of referential integrity. In our example VMG in Figure 18, for *book* relation, *author* and *publisher* constitutes its dependency island, while *review* is its referencing peninsula. *book* is a referenced continent of the other three relations.

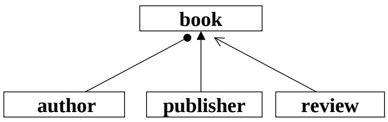

Figure 18: VMG for XQuery view defined in Figure 3

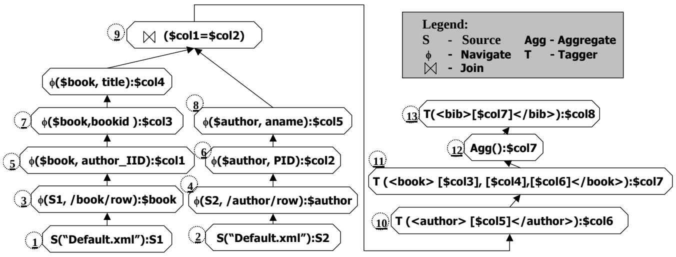

Figure 19: Sample XAT for the view query shown in Figure 11

5 Update Strategy

In this section, we first show the framework of the extension we have build on top of the base system *Rainbow*, and then describe our *decomposition-based update strategy*.

5.1 Rainfall — Update Extension to the Rainbow System

Rainbow is an XML data management system designed to support XQuery processing and optimization using an XML algebra and relational techniques. Figure 20 depicts the architecture of the *Rainbow Query Engine* with our now proposed update extension named *Rainfall*.

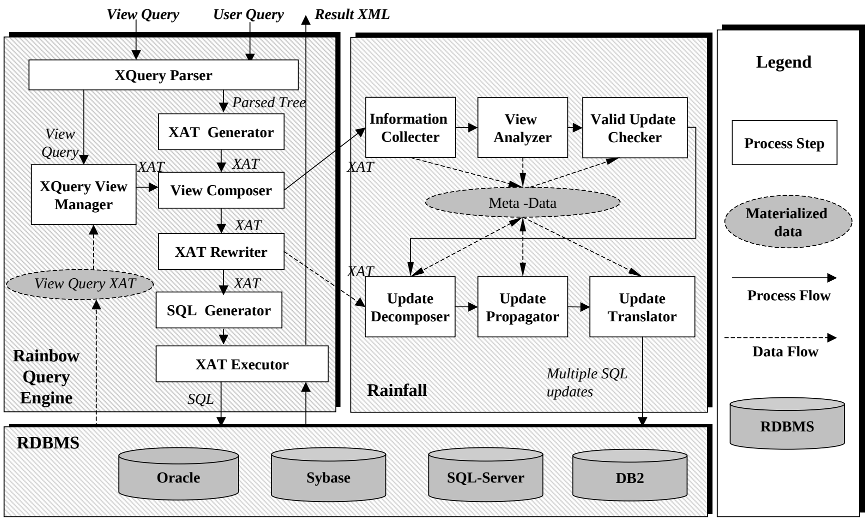

Figure 20: Architecture of Rainbow query engine with update extension

Operator	*Sym*	*Prms.*	*Output*	*Data*	*Description*
Tagger	T	p	col	s	Taggering s according to list pattern p.
Navigate	ϕ	$col1, path$	$col2$	s	Navigate into column col of s based on path $path$.
Select	σ	c	N/A	s	Filter input table s by condition c.
Join	$\bowtie$	c	$col+$	ls, rs	Join two input tables ls and rs under condition c, and output selected columns.
Source	S	$desc$	$col+$	N/A	Identify data source by description $desc$. It could be piece of XML fragment, XML document, or relational table.

Table 1: Related XAT Operators

An XML view query and update query are first passed to the *XQuery parser* for syntax checking. We have extended the *Kweelt XQuery parser* [2] to support the update grammar shown in Figure 6 in *Rainbow*. The *XAT generator* and *View composer* of the Rainbow query engine generates an algebraic representation of XQuery called XML algebra tree (XAT). The operators used in XAT include XML operators, SQL operators and special operators. Table 1 lists some operators used in our examples. Figure 19 is an example XAT for the virtual view shown in Figure 11. For a more general description of XQuery processing in *Rainbow* refer to [21].

Rainfall update system takes the above XAT from the *Rainbow query engine*. An *Infomation Collector* first identifies the core relations, extended relations, and involved relations of the view and their relationships. It then collects their schemas and integrity constraints. This information is stored in a metadata structure. After that, a *View Analyzer* studies the key features of the XQuery view as defined in Section 3.3.2. The *Valid Update Checker* examines if the user update query is valid or not. Invalid updates are rejected, while valid updates will be checked for translatability. Then, a *Update Decomposer* will decompose the translatable update query into several smaller-granularity update queries in that each of them works on a single table and records the decomposed or extended information into a metastructure. The *Update Propagator* then analyzes what type of propagated update should be generated to keep hierarchical consistency, and records the propagated updates into a metadata structure. Finally, the *Update Translator* translates the update information in the metadata structure into SQL update statements. These statements will be submitted to the relational database engine for execution. In some special case like round-trip update problem, since view did not have any abnormal construction, the *view analyzer* will skip the update translatability checking for valid updates, and directly generate the translation.

5.2 Information Collection

The *information collection* step analyzes the XAT generated by the *Rainbow query engine* to provide us prior knowledge about how a particular relation involved in the construction of the view will be affected by updates issued later on the view. Three types of information are collected: names of *core relations* of the view, the metadata information of the core relations, and the VMG.

(1) How to find core relations: Core relations of a view are exposed in the view and will be directly affected by the view update. They will be used in the later decomposition step when generating the translation plan. The *Navigate* operators, which start from *Source* in the view XAT, would provide the name of the core relations. The navigate operators with numbers 3 and 4 in Figure 19 having a *Row* step in their destination tell us the core relation name. In our example there are two core relations namely *book* and *author*.

(2) Collecting meta information about relations: Meta data of the core relations includes keys, foreign keys, names and data types of attributes as well as other constraints on the attributes of the relations. The knowledge of keys and foreign keys of a particular relation help to form the VMG. The information of attributes and their data types help us in setting default values for certain attributes that are not exposed in the view. Such an tuples extending is required, otherwise partial tuples cannot be inserted into relations. In addition, this metadata information is also useful for deciding about the validity of updates.

(3) Analyzing View Mapping Graph: As described in Section 4.2, the VMG identifies all the relations involved in the view update and their relationships. VMG represents the search space for the translation. This information will be used in the update propagation step to decide which relations are affected by the update translation plan and how they will be affected. Figure 21 gives our algorithm for analyzing the VMG.

```
Function analysisVMG ( XAT t )
begin
   new VMG(V,E)
   for ( each core relation Ri∈CR(V) )
        put Ri into V as a node of VMG
        identifyEdge (Ri)
   return VMG
end
```

```
Function identifyEdge ( node R )
begin
   for ( each relation Rj∈DB with Foreign key from Rj →R )
        t ← identify connection type between R and Rj
        if ( Rj∉V )
              put Rj into V as a node of VMG
              e ← identifyEdge (Rj)
        put edge  e(Rj,R) with connection type t into E
   for ( each relation Rj∈DB with Foreign key from R →Rj )
        t ← identify connection type between R and Rj
        if ( Rj∉V )
              put Rj into V as a node of VMG
              e ← identifyEdge (Rj)
        put edge  e(R,Rj) with connection type t into E
end
```

Figure 21: Algorithm for analyzing VMG

```
Class RelationalUpdate {
  String updateType
  String tableName
  Vector updateColumns
  Vector whereClause
}

Class UpdateColumn {
    String columnName
    Object oldValue
    Object newValue
}
```

Figure 22: Data Structure used in update decomposition algorithm

As next two steps, a *View Analyzer* will analyze the view features, while the *Valid Update Checker* will make sure that the issued update is valid. It will reject any invalid update. It also performs translatability checking for valid updates. The definition about validity and translatability of updates has been presented in Section 3.

5.3 Update Decomposition

Update Decomposition is the key step in XML update translation. The component named *Update Decomposer* is responsible for this step. In this step, the XML update query given in the form of an XML algebra tree is decomposed into multiple updates that are relation specific. Each of these updates is issued on an individual relation that belongs to the core relation set of the view.

Our strategy for update decomposition is a bottom up traversal strategy. The intuition is that each leaf at the bottom of the XAT contains information about one relation involved in the formation of the XML view. This relation hence is also a candidate for the update destination. In this update decomposition, the XML update represented by the *update XAT* will be decomposed into several relational updates. The number of such distinct updates will be less than or equal to the number of leaves of this XAT.

(1) Data Structure used in Decomposition: Figure 22 shows the meta data structure used to capture a result update after decomposition. *RelationalUpdate* is designed to keep information about the type of update, the relation name for which the update is being constructed, and knowledge about the attributes to be updated within the relation. Information about the attributes is stored in the form of a vector of *UpdateColumn* that includes the name of the attribute to be updated, and the new value that is going to be inserted, or the old and new values in the case of replace. In addition, the *WhereClause* of the *Relational Update* will store all conditions applicable to the relational update. The *RelationalUpdate*, though being a very simple data structure, is sufficient to store the amount of information required to construct a single relational update.

```
Function updateDecomposition ( XAT t )
begin
  XATleaves ← get all leaf nodes of XAT t
  resultUpdate ← new RelationalUpdate List
  for ( all nodes n in XATleaves )
     up ← new RelationalUpdate
     while  ( n is not null )
        opUpdateDecomp(n, up)
        n ← parent of node n
     decideUpdateType (up)
     put up into resultUpdate
  distinct resultUpdate
end

Function decideUpdateType (XAT t, RelationalUpdate up)
begin
  up.updateType ← same type as root node type of XAT
     if ( up.updateType is insert or replace )
        do nothing
     else
     if ( up.updateType is delete )
        if ( up.updateColumns include all attributes of relation up.tableNan
           up.updateType ← delete
        else
              up.updateType ← replace
end
```

Figure 23: Algorithm for update decomposition

```
Function opUpdateDecomp ( XATNode n,  RelationalUpdate up )
begin
 if ( n is a Navigate node )
   up.tableName ← get table name from n if it has one
 else
   if ( n is a Select node or a Join node )
     up.whereClause ← condition in predicate of node n
   else
     if ( n is a Tagger node )
       if ( up.updateType is delete )
           if ( n.DOMpattern matches DOM pattern of element node to be deleted )
             up.updateColum.columnName ← extract names of attributes from Tagger pattern
       if ( up.updateType is insert )
           if ( n.DOMpattern matches DOM pattern of element node to be inserted )
             up.updateColumn.columnName ← extract names of attributes from Tagger pattern
             up.updateColumn.columnValue ←extract attribute values from pattern of inserted element
       if ( up.updateType is replace)
             if ( n.DOM pattern matches DOM pattern of element node to replace )
             up.updateColum.columnName ← extract names of attributes from Tagger pattern
              up.updateColumn.oldColumnValue  ← extract old values from relational database
              up.updateColumn.columnValue ← extract values from pattern of replacing element
     else do nothing
       return up
end
```

Figure 24: Algorithm for update decomposition through XAT node.

(2) Update Decomposition Algorithm: Figure 23 depicts the algorithm for the traversal strategy that we follow to decompose an update. We start with the 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 construct relational updates and ignore all other types of nodes. In particular, we make use of *Navigate*, *Select*, *Join*, and *Tagger* nodes. As shown in the algorithm (Figure 24), the table can be extracted from the *Navigate* nodes. If the node is a *Select* or a *Join* node then we extract the conditions from these nodes to convert them into *where* clauses in the relational update. The attribute and value mappings will be decided by the *Tagger* nodes.

5.4 Update Propagation and Translation

Update decomposition produces a sequence of updates recorded in the metadata structure. As described in Section 4, there are various connections between the tables of a relational database. For those updates, which touch relations connected with others, we need the *update propagation* step to keep the database consistent. For different update types, there are different update propagation strategies as described below.

Algorithm for Delete. Assume the update is to delete a tuple t from a relation R. (1) Isolate the dependency island DI_r of relation R. (2) For each relation in DI_r, delete all tuples connected with t by foreign key connection. (3) For each reference peninsula attached to DI_r, perform a replacement of the foreign keys of each matching tuple. (4) For relations in DI_r that have a dependency island, deletion is propagated (repeatedly, if necessary) to their DI. (5) For those relations involved in a deletion, replacement is executed on their referencing peninsulas.

Algorithm for Insertion. Assuming an update is to insert a tuple t into relation R. (1) Extend the view tuple t as t' with values for the attributes that had not been exposed in the view definition. (2) Case 1: An identical tuple exists in R, reject the update. (3) Case 2: If t' does not match any key of R, perform an insertion into R. Further, for any relation $R' \in$RC(R), a check-insertion operation is done. (4) Case 3: If t' matches the key of an existing tuple, but some values for non-key attributes differ. If R has a dependency island, then reject the update. Otherwise, perform the replacement of t with t'.

We also have designed a similar algorithm for the propagation of a replace operation, but due to space reasons it is omitted here [19]. Finally, the *Update Translator* will translate these intermediate update representations into SQL update statements.

6 Experiments

In this section, we will describe our performance study over our *Rainfall* update system described in Section 5.1.

Experimental Setup. We conducted several experiments on our prototype system to assess the influence of different factors on its performance. We focus on the following factors here: (1) The type of update (insert, delete or replace) issued on the XQuery view. (2) The loading strategies used in building relational databases, such as inline, edge, universal, etc. (3) The size of the original XML files. If not stated otherwise, all experiments use the schema from our running example in Figure 1. The XML data is randomly being generated.

The test system is Intel(R) Celeron(TM) 733MHz processor, 384M memory, running Windows2000 and Java 1.3.0_01. Data connectivity was achieved through JDBC. The database used for our tests (local database and information source) are one installation of Oracle 8i under SUSE LINUX. The times were measured by comparing the system time before and after each step, i.e., they include system, user, and IO time.

Experiment on Updating vs. Loading. We evaluate the cost of the translation of an XML update via an XML view into a relational database and compare it against re-loading the XML data after the update into the database (Figure 25). We use a complete delete on the view defined in Figure 7. The loading strategy used is shared inlining [15]. We observe that as the XML file size increases in the number of elements in the XML document, the re-loading time increases linearly in the size of the file, while update translation remains fairly steady. Thus the update translation is an efficient mechanism and practical for keeping data consistent.

Experiment on Translation for Different Update Types. The performance of different update types is compared in Figure 26. The underlying relational database and view query are the same as in experiment 1. Update operations considered are complete delete, insert and replace on the same view. We find that all three types of update costs are fairly stable even for increasing file sizes. Delete is the cheapest operation. While replace is the most expensive yet is still cheaper than performing a combination of a delete followed by an insert.

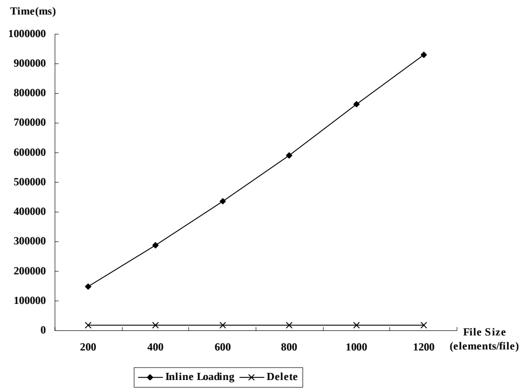

Figure 25: Performance comparison of reloading and update translation, shared inlining loading, delete update.

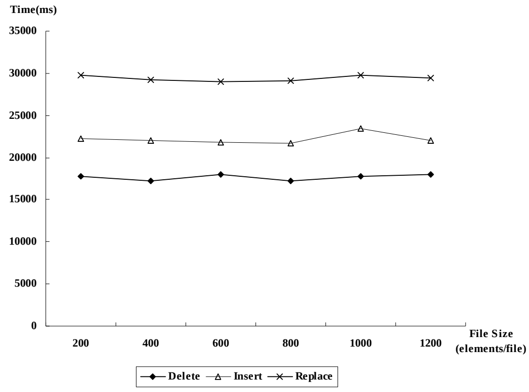

Figure 26: Performance comparison for different update types, file-size = 800 elements/file.

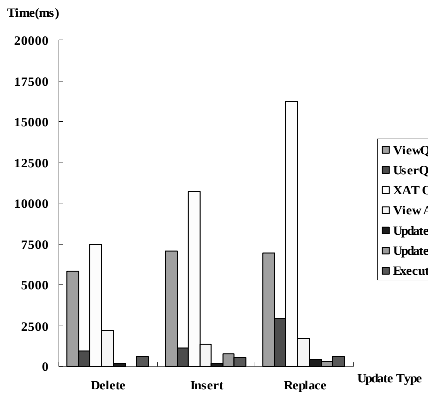

Figure 27: Comparison of all translation steps for different update types, file-size = 800 elements/file.

This is the reason for the fifth criteria in our concept of correctness of translatability described in Section 3.2. Given that only the last step of execution actually touches the relational data, the database size does not have much impact on the overall performance.

Experiment on Translation Steps for Different Update Types. For the same experimental setup as experiment 2, we now break down the costs for the different steps of update translation for the three update types. The result is shown in Figure 27. The result shows that the XAT optimization takes more time compared to XAT generation and update translation. We merge the user XAT and mapping XAT query trees, and optimize the merged XAT before we start the other update translation steps. The reason for this step is to simplify the XAT. Another costly step is the view analysis which analyzes the view structure, finds the *core relations* and their relationships and thus prepares for update translation.

7 Related Work

The view update problem has long been studied for relational databases. [9] is one of the first works dealing with view updates for relational databases. It stipulated a notion of *correct translation*, and described several straightforward conditions for the existence of such translation in the case of a *clean source*. An abstract formulation of the update translation problem is given by the *view complementary theorem* in [7, 3]. [11, 12, 1] study the view update problem for SPJ queries on relations that are in Boyce-Codd Normal Form. Our work is more complex than the pure relational view update problem, since not only do all the problems in the relational context still exist in the XML semantic context, but in addition we have to address the mismatch coming from the two distinct data models, that is the XML hierarchical view model and the flat relational base model. Our constraint mapping in Figure 10 takes this mismatch into account, addressing the new issues in the XML context.

Closely related to the work of [9], in [4], view update translation algorithms of [11] have been further extended for object-based views. In our work, we adapt the strategy now for the XQuery view context. However, an XML model has distinct features in its schema as shown in Figure 10. XQuery FLWU updates on the view are also different with SQL update queries used in the OO model. Our algebraic framework bridges the nested XQuery with SQL, while the decomposition algorithm in the update strategy bridges updates between the hierarchy and the flat model. They thus provide us with a clear solution for view update translation.

The XML view update problem has not yet been much addressed by the database community. The most recent work [16] presents an XQuery update grammar, and studies the performance of updates assuming that the view is indeed translatable and has in fact been already translated somehow using a fixed shredding technique, that is inlining [15]. [14] introduces the XML view update in SQL-Server2000, based on a specific *annotated schema* and update language called *updategrams*. Different with their work, our work in general explores the XML view update problem instead of a system-specific solution. It works for various loading strategies and a majority of relational database instance.

8 Conclusion

In this paper, we characterize the XQuery view update problem in the context of XML views being published over relational databases. A general solution strategy for solving this problem is presented. In particular, a decomposition-based approach is described for generating update plans to perform a translatable update without introducing any side-effects to the view. A system for solving the XQuery view update problems is presented. Its performance is studied in various scenarios.

Although we base our discussion and have implemented the update strategy in the *Rainbow* system, both the concepts and the algorithms can be easily applied to other systems. Future extensions to our work could include considering element order in XML view updates and studying index techniques for optimizing the efficiency in locating the updated nodes in the view level or locating the related tuples at the database level.

References

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

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

[3] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. In *ACM Transactions on Database Systems*, pages 557–575, Dec 1981.

[4] T. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold. Updating Relational Databases through Object-Based Views. In *Proceedings of the 10th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Portland, OR*, pages 248–257, 1991.

[5] E. T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML), 1997. http://www.w3.org/TR/PR-xml-971208.

[6] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *The VLDB Journal*, pages 646–648, 2000.

[7] S. S. Cosmadakis and C. H. Papadimitriou. Updates of Relational Views. *Journal of the Association for Computing Machinery*, pages 742–760, Oct 1984.

[8] F. Daniela and K. Donald. Storing and Querying XML Data Using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[9] U. Dayal and P. A. Bernstein. On the Correct Translation of Update Operations on Relational Views. In *ACM Transactions on Database Systems*, volume 3(3), pages 381–416, Sept 1982.

[10] M. F. Fernandez, A. Morishima, D. Suciu, and W. C. Tan. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Engineering Bulletin*, 24(2):12–19, 2001.

[11] A. M. Keller. Algorithms for Translating View Updates to Database Updates for View Involving Selections, Projections and Joins. In *Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 154–163, 1985.

[12] A. M. Keller. Choosing a View Update Translator by Dialog at View Definition Time. In *VLDB*, pages 467–474, 1986.

[13] D. Lee and W. W. Chu. Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 323–338, Oct 2000.

[14] M. Rys. Bringing the Internet to Your Database: Using SQL Server 2000 and XML to Build Loosely-Coupled Systems. In *VLDB*, pages 465–472, 2001.

[15] J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, pages 302–314, September 1999.

[16] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. pages 413–424, May 2001.

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

[18] W3C. XQuery: A Query Language for XML. http://www.w3.org/TR/xquery/, February 2001.

[19] L. Wang, M. Mulchandani, and E. A. Rundensteiner. Updating XQuery Views Published over Relational Data. Technical report, Computer Science Department, WPI, 2003.

[20] X. Zhang, K. Dimitrova, L. Wang, M. EL-Sayed, B. Murphy, L. Ding, and E. A. Rundensteiner. RainbowII: Multi-XQuery Optimization Using Materialized XML Views. In *Demo Session Proceedings of SIGMOD*, 2003 to appear.

[21] X. Zhang and E. Rundensteiner. XAT: XML Algebra for Rainbow System. Technical Report WPI-CS-TR-02-24, Computer Science Department, WPI, July 2002.