Updating XQuery Views Published over Relational Data: A Round-trip Case Study

Ling Wang, Mukesh Mulchandani, and Elke A. Rundensteiner

Department of Computer Science
Worcester Polytechnic Institute Worcester, MA 01609
{lingw, mukesh, 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. Updating such virtual XML views, however, is not well studied, although it is essential for building a viable full-featured XML data management systems. XML view update is a challenging problem because of having to address the mismatch between the two rather different data models and distinct query paradigms. In this paper, we tackle the XQuery view update problem, in particular, we focus on the *round-trip XML view update subproblem*. This case, characterized by a pair of loading and extraction mappings that load XML data into the relational store and extract appropriate XML views, is very common in practice, as many applications utilize a relational engine for XML document storage. We discuss and prove the updatability of such views. We also present a decomposition-based update translation strategy for solving this problem. As evidence of feasibility, we have implemented the proposed strategies within the *Rainbow* XQuery system. Experimental studies are also given to assess the performance characteristics of our update system in different scenarios.

1 Introduction

Motivation. XML [5] has become the standard for interchanging data between web applications because of its modeling 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. Examples of such XML management systems include EXPERANTO [6], SilkRoute [10] and Rainbow [22], which typically offer support for XML view creation over relational data and for querying against such XML wrapper views 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, not just queries of (virtual) XML views.

This view-update problem is a long-standing issue that has been studied in the context of the relational data model. Much work has been done on defining

what a correct translation entails [9] and how to eliminate ambiguity in translation [7, 2]. However, update operations have not been given too much attention yet in the XML context. [18] studies the performance of translated updates executed on the relational store, assuming that the view update is indeed translatable. Updating of virtual XQuery views comes with new challenges beyond those of relational views since we have to address the mismatch between the two data models (the flexible hierarchical XML view model and the flat relational base model) and between the two query languages (XQuery versus SQL queries).

In this paper, we characterize a common sub-case of the *XQuery view update* problem which we call the **Round-trip XML View Update Problem (RXU)**. This is an important case since many XML applications use relational technology to store, query and update XML documents. Such systems require typically a two-way mapping to first load and then to extract XML out of the relational database. Hence, we refer to this as the "round-trip" case. In this paper, we show that the view update operations in this case are always translatable.

We present our framework named *Rainfall* for update translation of this round-trip problem. Due to there not yet being any standard update language, we have extended the XQuery grammar to support XML updates similar to [18]. We have implemented the proposed strategies for update decomposition, translation and propagation within the *Rainbow* XML data management system [22]. Experiments are also presented to compare update translation with the alternative, which would be the re-loading of the updated XML into the relational data store. We also assess various performance characteristics of our update solution.

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

- We characterize a subproblem of the general XML view update called *round-trip XML view update problem (RXU)*, which is a common case in practice.
- We formally describe the view updatability for the RXU case and prove its correctness.
- We provide a decomposition-based update translation solution called *Rainfall* to translate XQuery updates on XML virtual views into a set of SQL-level updates.
- We implement our update solution within the *Rainbow* XML data management system to support the view update extension.
- We present a performance study conducted to assess our update translation strategy.

Outline. This paper is structured as follows. We briefly introduce the XML data model and XQuery update extension in Section 2. Section 3 characterizes the *Round-trip XML view update problem*, and discusses the view updatability in this case. We describe our decomposition-based update strategy and system implementation in Section 5 and evaluate these techniques in Section 6. Section 7 reviews related work while Section 8 concludes our work.

2 Background

XQuery Views of Relational Data. XML (Extensible Markup Language) [5] is used both for defining document markup and for data exchange. XML Schema [19] 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/XMLSchema>
  <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>
```

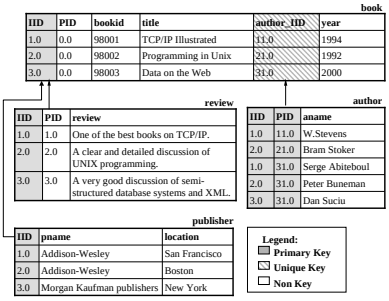
Fig. 1. Example XML schema

```
<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>
  ... ...
</bib>
```

Fig. 2. Example XML data

Many XML applications use a relational data store by applying loading strategy such as [17, 8]. Figures 3 and 4 show an example relational database generated from the XML schema and data of our running example using a shared inlining loading strategy [17]. The basic XML view, called *Default XML View*, is a one-to-one mapping to bridge the gap between the two heterogeneous data models, that is the XML (nested) data model and relational (flat) data model. Each table in the relational database is represented as one XML element and each of its tuples as subelements of this table element. Figure 5 depicts the default XML view of the database (Figure 3).

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. For this reason, XML data management systems provide a facility to define user-specific view capabilities on top of this default XML view, called a



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

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.

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

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

Fig. 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))
```

Fig. 4. Database schema of Figure 3

virtual view. Such a *virtual view* can be specified by an XQuery expression, then called a *view query*. Several recent systems such as XPERANTO [6], SilkRoute [10] and Rainbow [22] follow this approach of XML-to-Relational mapping via defining XML views over relational data. An XML query language, such as XQuery proposed by World Wide Web Consortium (W3C), can be used both to define such views and also to query them. Figure 6 shows the view query defining a virtual view identical to the originally loaded XML document in Figure 2.

XQuery Updates. Although W3C is adding update capabilities to the XQuery standard [20], currently no update language for XML has yet been standardized. For our work, we thus adopt an extension of the XQuery language syntax with update operations that follows [18]. XQuery is extended with a *FLWU* expression composed of *FOR...LET...WHERE...UPDATE* clauses (Figure 7). Figure 8 shows an example *Insert* update, which inserts a new *book* element into the (virtual) view defined in Figure 6.

3 Round-Trip XML View Update Problem

3.1 Definition of the Round-Trip XML View Update Problem

The general *XQuery view update problem* can be characterized as follows. Given a relational database and an XQuery view definition over it, can 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 be done.

Given the general problem definition as above, we now focus on one important case which we name the **round-trip XML view update problem**. Given an XML schema and a valid XML document, by using a suitable loading algorithm, such as inlining [17], edge or universal [8], accompanied with a constraint-preserving mapping such as described in [14], assume we built a relational database. We call it a **structured database**. Further we specify an

```
<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>
```

Fig. 5. Default XML view of database shown in Figure 3

```
<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>
```

Fig. 6. Virtual XQuery view over default XML view shown in Figure 5 producing the XML data in Figure 2

XML view query on this *structured database* using an XQuery expression, which constructs an XML view with the content identical to the XML document that had just been supplied as input to the loading mapping. We call this special-purpose view query an *extraction query*. We then can extract a view schema by analyzing the extraction query semantics and the relational database schema. Thus the view has the same content and schema as the original XML document which had just been captured by the relational database. We call this special view a *twin-view*. The problem of updating the database through this *twin-view* is referred to as the *round-trip XML view update problem* (Figure 9).

3.2 Characterization of the XML Loading

As defined above, RXU is closely related with the loading procedure of the XML document and schema into the relational database. To address the influence of the loading strategy on the view updatability, we hence now study the loading strategy characteristics for the RXU case. Many XML loading strategies have been presented in the literature [14, 17, 8]. Not only the XML document, but also the XML schema is typically captured in this procedure, which are called data and constraint information respectively.

```
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.
```

Fig. 7. Update language as extension of XQuery

```
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>
}
```

Fig. 8. Insert update on XQuery view shown in Figure 6

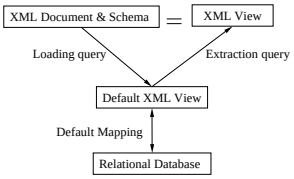


Fig. 9. Round-Trip Update Problem

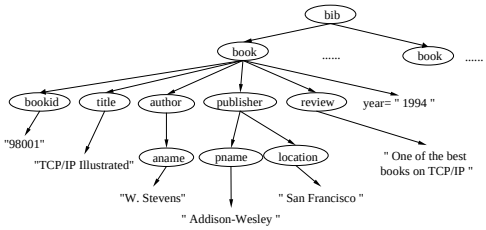


Fig. 10. Tree representation for XML document shown in Figure 2

Data Loading Completeness. The XML (nested) data structure is distinct from the relational (flat) data model. Thus the loading procedure must translate from one model (structure) to the other. The completeness of data loading is important in RXU since the *twin-view* requires exactly the same content as the original document , independent on whatever we may do to the structure.

Definition 1. *Given an XML document D_x, a loading L generates a resulting relational database instance D_r, denoted by $D_x \xrightarrow{L} D_r$. L is a* **lossless data loading** *iff $\exists L'$ such that $D_r \xrightarrow{L'} D_x$ holds true.*

Figure 10 is a tree structured representation of the XML document in Figure 2, while Figure 3 is a structured database resulting from applying the inlining loading to that XML document. The extraction query in Figure 6 will generate the *twin-view* from the *structured database* of Figure 3. Thus this loading is a lossless data loading by Definition 1.

A lossless data loading guarantees to capture all leaves in the XML tree-structured representation (Figure 10). Leaves represent actual data instead of

document structure. Hence we will be able to reconstruct the XML document. While a lossy data loading may not have loaded some of leaves, hence is not sufficient for reconstruction. Most loading strategies presented in the literature, such as Inlining [17] and Edge [8], are all lossless data loadings.

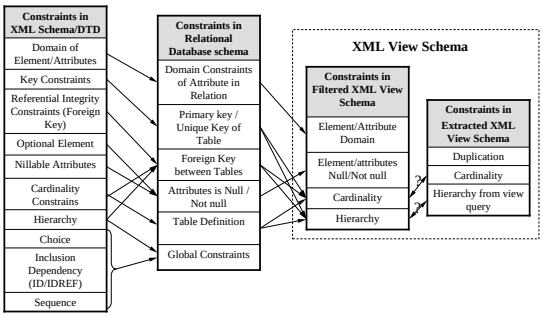


Fig. 11. Comparison of constraints of XML schema, relational database schema and XML view schema

```
<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>
   </book>
}
</bib>
```

Fig. 12. XQuery example

Constraint Loading Completeness. Given a relational database schema S_r and a view query Q, we define the constraints implied by the XML view as *XML View Schema*, which can be extracted by a mapping named *constraint extraction mapping* denoted by $\hat{e}$. As shown in Figure 11, an *XML View Schema* is a combination of a *Filtered XML View Schema (FSchema)* FS_v and an *Extracted XML View Schema (ESchema)* ES_v, thus denoted as $S_v(ES_v, FS_v)$. An *FSchema*, modeling the constraints extracted from the schema of any relation referenced by Q, is inferred by analyzing the relational database schema and filtering this schema using the view definition, hence denoted as $FS_v = \hat{e}(S_r)$. These constraints may include domain constraints, cardinality constraints, null constraints and hierarchical constraints. An *ESchema* consisting of constraints that can be inferred from the *view query* semantics is extracted by analyzing the view query expression, hence represented as $ES_v = \hat{e}(Q)$. They include cardinality constraints, hierarchical constraints and duplication constraints. The constraints implied in the view definition may not be consistent with the constraints imposed by the relational database schema. Hence the ESchema may conflict with the FSchema. This mismatch may cause some problem in the later update translation step. However, in RXU, we assume that the *view schema* is exactly the same as the original XML schema. Hence this mismatch problem will not arise. This assumption relies on the idea of *constraint loading completeness* and *extraction query*.

Definition 2. *Given an XML schema S_x, a loading L generates a structured database with schema S_r, denoted by $S_x \xrightarrow{L} S_r$. L is a* **lossless constraint loading** *iff $\exists Q$ be an extraction query generating an XML view with schema $S_v = (\hat{e}(S_r), \hat{e}(Q))$, such that $S_v = S_x$ holds true.*

An XML to relational database loading is a **lossless loading** iff it is both a lossless data loading as defined by Definition 1 and a lossless constraint loading as defined by Definition 2. Obviously the loading in RXU must be a lossless loading. Most loadings proposed in the literature are all lossless data loading strategies, however few of them are also lossless constraint loading strategies. For example, Edge [8] is a lossless data loading, while it is not a lossless constraint loading. In order for such loading strategies to be usable for RXU, it must accompany a constraint preserving loading such as proposed in [14].

4 On the View Updatability in RXU

Basic Concepts. We first review the relational data model and view definition framework. The notation used is shown in Table 1. A *relational database* is a combination of a set of relations and a set of integrity constraints. A *database state*, denoted by s, is an assignment of data values to relations such that the integrity constraints are satisfied. The database *status*, denoted by S, is the set of all possible database states. An *data update* of a relational database with status S is a mapping from S into S, denoted as $\hat{u} : S \rightarrow S$. A *view* V of a given relational database with status S is defined by a set of relations and a mapping f that associates with each database state $s \in S$ a view state $f(s)$. In our case the mapping f is the *view definition mapping* expressed in an XQuery Q. The set $f(S) = \{f(s)|s \in S\}$ is the *view status*. The set of view definition mappings on S is denoted as $M(S)$. A *valid view update* u on view state is an update that satisfies all the constraints of view schema.

S	database status	s	current database state
f	view definition mapping	$M(S)$	a set of view definition mappings on S
$f(S)$	view status	$f(s)$	view state associated with database state s
U^r	set of all database updates	U^v	set of all valid view updates

Table 1. Notation table

Translation Criteria. We now discuss what is the criteria of translating an XML view update. By the **Correctness Criteria,** only the desired update is performed on the view, that is, it is consistent and has no view side effects. Given $u \in U^v$, $\exists u' \in U^r$ such that (a) $u(f(s)) = f(u'(s))$, (b) $\forall s \in S$, $uf(s) = f(s) \Rightarrow u'(s) = s$. In order to permit all possible changes but only in their simplest forms, **the Simplicity Criteria** requires that all candidate update translations satisfy

the following rules [11]: (a) *One step changes.* Each database tuple is affected by at most one step of the translation for any single view update request. (b) *Minimal changes.* There is no valid translation that implements the request by performing only a proper subset of database requests. (c) *Replacement cannot be simplified.* That is, we always pick the simplest replace operation, e.g, a database replacement that does not involve changing the key is simpler than one where the key changes. (d) *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 3. *Given an update $u \in U^v$ on view state $f(s) \in f(S)$, if $\exists u' \in U^r$ that satisfies the correctness criteria defined above, u is called* **translatable** *for $f(s)$ (also can be called f-translatable). $f(s)$ is called* **updatable** *by u. u' is named a* **correct translation** *for u. Further, if u' also satisfies the simplicity criteria defined above, we say u' is an* **optimized translation**.

Updatability of RXU views. We now study the updatability of views in the RXU space. The view complement theory in [2] proposes that if a complementary view, which includes information not "visible" in the view, is chosen and is held constant, then there is at most one translation of any given view update. Although as described in [13], translators based on complements do not necessarily translate all translatable updates. It still provides us with a conservative computation for the set of translatable updates. This fits our RXU case well, since here the complement view always corresponds to a constant. We hence use the view complement theory to prove that any update on a *twin-view* is always translatable.

The complementary theory proposed in [2] can be explained as below.

Definition 4. *Let $f, g \in M(S)$. We say that f is greater than g or that f determines g, denoted by $f \geq g$, iff $\forall s \in S, \forall s' \in S, f(s) = f(s') \Rightarrow g(s) = g(s')$.*

Definition 5. *Let $f, g \in M(S)$. We say that f and g are equivalent, denoted by $f \equiv g$, iff $f \geq g$ and $g \geq f$.*

Definition 6. *Let $f, g \in M(S)$. The product of f and g, denoted by $f \times g$, is defined by $f \times g(s) = (f(s), g(s)), \forall s \in S$.*

Definition 7. *Let $f \in M(S)$. A view $g \in M(S)$ is called a* **complement** *of f, iff $f \times g \equiv 1$. Further, g is the* **minimal complement** *of f iff (i) g is a complement of f, and (ii) if h is a complement of f and $h \leq g$, then $h \equiv g$.*

Definition 4 can be interpreted as $f \geq g$ iff whenever we know the view state $f(s)$, then we also can compute the view state $g(s)$. Definition 6 implies that the product $f \times g$ "adds" to f the information in g. We denote the identity mapping on S as **1** and a constant mapping on S as **0**. In our case, the mapping query used to define the default XML view is mapping **1**. And a XQuery such as $< bib >< /bib >$ is a constant mapping. According to Definition 7, if $f \times g \equiv 1$,

then f, g contain sufficient information for computing the database, and the complementary view g contains the information not "visible" within the view f. For example, assuming the query in Figure 6 define a mapping f, the query in Figure 12 defines a mapping g, then $f \geq g$ and $g \times f \equiv 1$. f is complement of g.

Lemma 1. *Given a complement g of f and a view update $u \in U^v$, u is g-translatable iff $\forall s \in S, \exists s' \in S$ so that $f(s') = uf(s)$ and $g(s') = g(s)$.*

This lemma is the complement theory, which implies that given a complement g of the view f and a view update $u \in U^v$, the translation of u that leaves g invariant is the desired translation satisfying our correctness criteria defined above. This is first presented in [9] as the "absence of side effects" feature. For the proof of this lemma, please refer to [2]. We now use this theory to prove that any update on the view of RXU is always translatable, as described below.

Observation 1 *Within the RXU case, given an XQuery view definition f defined over the relational state s, $\forall u \in U^v$, u is translatable by Definition 3.*

Proof. (i) Since the mapping query defining the default XML view is $\mathbf{1}$, according to Definition 5, in RXU, $\forall f$, $f \equiv \mathbf{1}$. This is because we can always compute the default XML view from the view state $f(s)$ by using the loading mapping, that is $f \geq \mathbf{1}$, while $\mathbf{1} \geq f$ always holds true. (ii) Since $\mathbf{0}$ is the complement of $\mathbf{1}$, while $f \equiv \mathbf{1}$, then $\mathbf{0}$ is the complement view of f. (iii) $\forall u \in U^v$, let $f(s') = uf(s)$, then $\mathbf{0}(s') = \mathbf{0}(s)$. Thus, by Lemma 1, u is always translatable.

5 Rainfall — Our Approach for XQuery View Update

Updating through an XML view can be broken into three separate but consecutive processes:

- *Information Preparation.* This process analyzes the XQuery view definition to provide us with a prior knowledge about the relationship of the view with the relational database, that is, extracting the view schema. It also performs pre-checking of updates issued on the view to reject invalid updates using the view schema.
- *Update Decomposition.* This is the key process of the XML update translation to bridge the XQuery model and the relational query model. The given XML update request is decomposed into a set of valid database operations, with each being applied to a single relation.
- *Global Integrity Maintenance.* Because of the structural model of the relational database with its integrity constraints, the database operations resulting from the *decomposition* process may need to be propagated globally throughout the base relations to assure the consistency of the relational database.

We hence call our strategy a *decomposition-based update strategy.* Our update strategy will generate an optimized update translation which follows the simplicity criteria defined in Section 4. For details on our update translation strategy, please refer to [21].

5.1 System Framework

Figure 13 depicts the architecture of our *Rainfall* update system, which is an extension of the base XML query engine *Rainbow* [22]. *Rainbow* is an XML data management system designed to support XQuery processing and optimization based on an XML algebra with the underlying data store being relational.

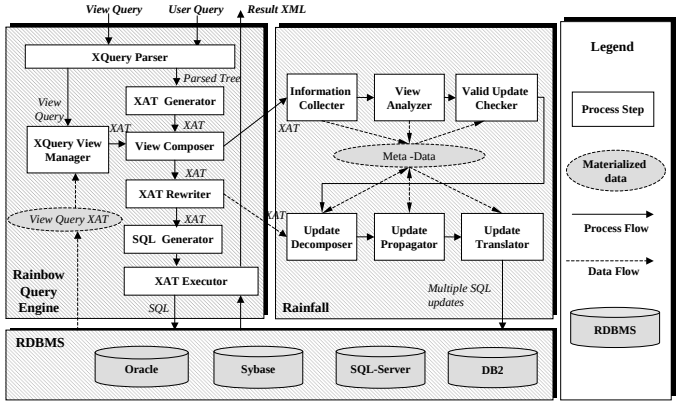


Fig. 13. Architecture of Rainbow query engine with update extension

An XML view or update query is first passed to the *XQuery parser* for syntax checking. We have extended the *Kweelt XQuery parser* [16] to support the update grammar (Figure 7). The *XAT generator* and *View composer* of the Rainbow query engine generates an algebraic representation of XQuery called XML algebra tree (XAT). For a description of XQuery processing in *Rainbow* refer to [23].

The *Rainfall* update system takes the above XAT from the *Rainbow query engine*. An *Infomation Collector* first identifies all the relations related to the view and their relationships. It then collects their schemas and integrity constraints. This information is stored in a metadata structure, which will serve as *view schema* as defined in Section 3. After that, a *View Analyzer* studies the key features of the XQuery view definition to prepare a translation policy. The *Valid Update Checker* examines if the user update query is valid or not. Invalid updates are rejected, while valid updates will be prepared for further processing. Then, the *Update Decomposer* will decompose the translatable update query into several smaller-granularity update queries, each defined on a single relational table. The *Update Propagator* then analyzes what type of propagated update should be generated to keep the integrity constraints of the relational database satisfied. It also records the propagated updates into a metadata structure for the next translation step. 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.

6 Experiments

We conducted several experiments on our *Rainfall* update system to assess the performance of our update translation strategy in RXU case. We first show the update translation over XML re-loading to claim that update translation is indeed viable in practice. We then show our update translation strategy is pretty stable and efficient in different update scenarios. In the last experiment, we describe the performance of each translation step. If not stated otherwise, all experiments use the XML 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.

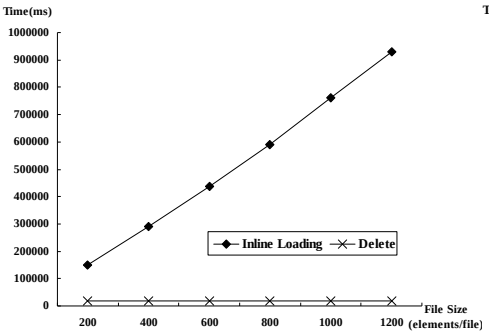


Fig. 14. Performance comparison of re-loading and update translation, shared inlining loading, delete update.

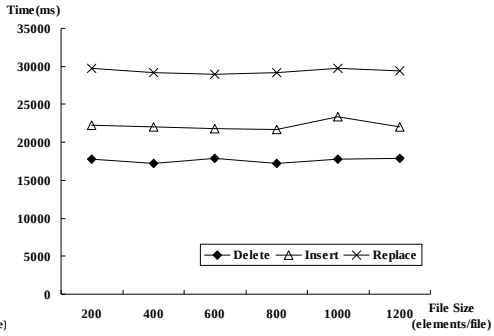


Fig. 15. Performance comparison for different update types, file-size = 800 elements/file.

(1) Experiment on Updating vs. Loading. We evaluate the cost of the translation of an XML update specified against an XML virtual view (Figure 6) into a relational update that then is executed against the relational database. We compare it against re-loading the XML data into relational data store after directly being applied the update on the original XML document (Figure 14). We use a delete update on the view defined in Figure 6. The loading strategy used is shared inlining [17]. 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. The update translation remains fairly steady. Thus the update translation is an efficient mechanism and indeed appears to be viable in practice.

(2) Experiment on Translation for Different Update Types. The performance of different update types is compared in Figure 15. The underlying relational database and view query are the same as in experiment 1. Update operations considered are delete, insert and replace on the view. We find that all

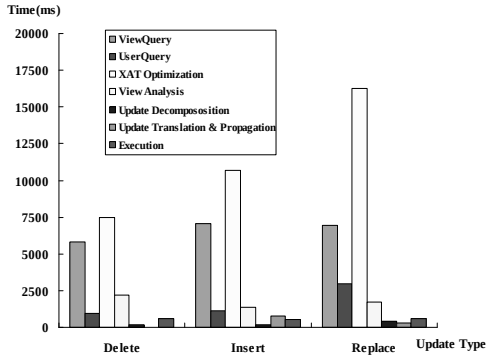


Fig. 16. Comparison of translation steps for different update types

three types of update costs are fairly stable even for increasing file sizes. Delete is the cheapest operation. While replace is the most expensive, it is still cheaper than performing a combination of a delete followed by an insert. This is the reason for the forth simplicity criteria described in Section 4. 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.

(3) Experiment on Translation Steps for Different Update Types. For the same experimental setup as in 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 16. 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 related *relations* and their relationships and thus prepares for update translation.

7 Related Work

The view update problem has been studied in depth 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 conditions for the existence of such translation in the case of a *clean source*, that is updating a clean source will not generate any view side-effect. An abstract formulation of the update translation problem is given by the *view complementary theorem* in [7, 2] which uses the complement of a view to resolve the ambiguity in mapping between old and new database states. Finally, [11, 12, 1] study the view update problem for SPJ queries on relations that are in Boyce-Codd Normal Form. Our work follows [7] to prove the correct translatability. However, it 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 semantics context, but in addition we have to address the mismatch coming from the two distinct data models. Our constraint mapping in Figure 11 takes this mismatch into account, thus addressing some of the key issues in the XML context.

Closely related to the work of [9], in [3], view update translation algorithms of [11] have been further extended for object-based views. However, an XML model has features in its schema and query language distinct from those in the OO model. The algebraic framework and the update decomposition strategy used in our system bridges the nested XQuery with SQL model gap. 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. [15] introduces the XML view update in SQL-Server2000, based on a specific *annotated schema* and update language called *updategrams*. Different with their work, our update system explores in general the XML view update problem instead of a system-specific solution, though we have also implemented our ideas to check their feasibility. One of the recent work [18] presents an XQuery update grammar, and studies the performance of updates assuming that the view is indeed translatable and has in fact already been translated using a fixed shredding technique, that is inlining [17]. Instead of assuming update always translatable, our work addresses how the updatability infected by XML nested structure. The proposed solution is not limited in specific loading strategy. The most recent work [4] studies the updatability of XML view using nested relational algebra. By assuming the algebra representation of view do not include unnest operator, while nest operator occur last, and won't affect the view udpatability. However, by using XQuery to define the view, the unnested operator is unavoidable. Also, since the order of nest operator will decide the hierarchy of XML view, it will affect the view updatability. Compared to their work, updating XQuery view problem tackled in our paper is more complex.

8 Conclusions

In this paper, we have characterized the round-trip based XQuery view update problem in the context of XML views being published over relational databases. We prove that the updates issued on the view within this problem space are always translatable. A decomposition-based update translation approach is described for generating optimized update plans. A system framework for implementing this approach is also presented. Its performance is studied in various scenarios. Although we base our discussion and have implemented the update strategy in the context of the *Rainbow* XML management system, both the concepts and the algorithms can easily be applied to other systems.

References

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

2. F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. In *ACM Transactions on Database Systems*, pages 557–575, Dec 1981.
3. T. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold. Updating Relational Databases through Object-Based Views. In *10th ACM SIGACT-SIGMOD*, pages 248–257, 1991.
4. V. P. Braganholo, S. B. Davidson, and C. A. Heuser. On the Updatability of XML Views over Relational Databases. In *WEBDB*, 2003.
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. A. M. Keller. Comments on Bancilhon and Spyratos' "update semantics and relational views". *ACM Transactions on Database Systems*, 12(3):521–523, 1987.
14. D. Lee and W. W. Chu. Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. In *ER*, pages 323–338, Oct 2000.
15. 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.
16. A. Sahuguet and L. Dupont. Querying xml in the new millennium, 2002.
17. 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.
18. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proceedings of the ACM SIGMOD International Conference*, pages 413–424, May 2001.
19. W3C. XML Schema. http://www.w3.org/XML/Schema.
20. W3C. XQuery: A Query Language for XML. http://www.w3.org/TR/xquery/, February 2001.
21. L. Wang, M. Mulchandani, and E. A. Rundensteiner. Updating XQuery Views Published over Relational Data. Technical Report WPI-CS-TR-03-23, Computer Science Department, WPI, 2003.
22. 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.
23. X. Zhang and E. Rundensteiner. XAT: XML Algebra for Rainbow System. Technical Report WPI-CS-TR-02-24, Computer Science Department, WPI, July 2002.